`ePiX` Tutorial and Reference Manual

Andrew D. Hwang
Department of Math and CS
College of the Holy Cross

Version 1.0, September, 2004

Contents

Chapter 1

Introduction

`ePiX`, a collection of batch-oriented utilities for *nix, creates mathematically accurate line figures, plots, and movies using easy-to-learn syntax. LaTeX and `dvips` comprise the typographical rendering engine, while `ImageMagick` is used to create bitmapped images and animations. The user interface resembles that of LaTeX itself: You prepare a short scene description in a text editor, then "compile" the input file into a picture. Default output formats are `eepic` (a plain text enhancement to the LaTeX picture environment), `eps`, `pdf`, `png`, and `mng`.

`ePiX`'s strengths include:

- Quality of output: `ePiX` creates mathematically accurate, publication-quality figures whose appearance matches that of LaTeX. Typography may be put in a figure as easily as in an ordinary LaTeX document.

- Ease of use: Figure objects and their attributes are specified by simple, descriptive commands.

- Flexibility: In `ePiX`, an object is described by its attributes and Cartesian location; as in LaTeX, printed appearance is determined when the figure is compiled. A well-designed figure can be altered dramatically, yet precisely, with minor changes to the input file.

- Power and extendability: `ePiX` inherits the power of `C++` as a programming language; variables, data structures, loops, and recursion can be used to draw complicated plots and figures with just a few lines of input. External code can be incorporated in an `ePiX` figure with a command line option or by using a Makefile.

- Economy of storage and transmission: For a document containing many figures, a compressed tar file of the LaTeX sources and `ePiX` files is typically a few percent the size of the compressed PostScript file.

- License: `ePiX` is *Free Software.* 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.

The relationship of `ePiX` to a graphical drawing program is analogous to the relationship between LaTeX and a word processor; `ePiX` facilitates logical structuring of mathematical figures. Though `ePiX` makes a few stylistic defaults to streamline the creation of simple figures, it imposes no internal restrictions on the contents or appearance of a figure; aesthetic and practical decisions are left to the user.

This manual is meant to be read in stages rather than "cover to cover". If you are a:

- Potential user, you may wish to skip immediately to "Software Dependencies" before investing additional time.

- New user, please proceed from here until you have enough understanding to play with the software, then experiment with the samples files while reading Chapter 2, or return to the manual as needed.

- More advanced user, browse at will, probably starting with Chapter 3.

In any case, please don't hesitate to contact the author or join the mailing list if you have questions or comments (good or bad) about the software or manual, or if you are willing and able to join in developmment.

Under the philosophy that people learn most easily when ideas are introduced in context, this manual is relatively conversational, and occasionally redundant (especially between portions meant for readers at different levels of familiarity). Throughout, you are assumed to be familiar with LaTeX and basic linear algebra (the description of points, vectors, lines, and planes in three-dimensional space). Other material, such as `C++` syntax, is introduced as needed.

1.1 Software Dependencies

If you run GNU/Linux, a BSD, or Solaris, you almost surely have (or can install) all the external software needed to use `ePiX`. For Mac OS X, you will need the Apple developer tools, and will want to install an X server and the `fink` package manager to build a featureful *nix environment. Users of other operating systems, most notably Windows, face a challenge in running `ePiX`, though not an insurmountable one. The author's (second-hand) Windows-specific knowledge is summarized below.

"Under the hood", an input file is successively converted to `eepic`, `dvi`, PostScript, `pdf` or `eps`, and if desired, `png` or `mng`. `ePiX` comprises a compiled library written in `C++`, a `C++` header file, and four shell scripts—`epix`, `laps`, `elaps`, and `flix`—that automate the various file format conversions. Each script is written in GNU `bash`. Consequently, there are two absolute necessities: a `C++` compiler (preferably `g++`) and `bash`. The script `epix` uses only these programs. `ePiX` is primarily a pre-processor for LaTeX, but does not absolutely require LaTeX for normal use. However, without LaTeX and Ghostscript (particularly `dvips`), you can't view or print `epix`'s output files, nor can you run `flix` or `elaps`.

A text editor such as `emacs` or `vim` that facilitates formatting `C` code is extremely useful for writing input files. `ePiX` comes with an `emacs` mode, written by Jay Belanger, that allows you to write, compile, and view `ePiX` figures without leaving `emacs`.

In the presence of LaTeX and Ghostscript, only a few standard utilities are needed to run `ePiX`'s "conversion to `eps/pdf`" script `elaps`, namely `grep`, `sed`, `epstopdf`, and `ps2epsi`.

Finally, `flix` uses ImageMagick's `convert` utility to create `png` images, and to assemble `png`s into animated `mng` files. The programs `animate` and `display` are useful for viewing `flix` output.

Aside from their reliance on specific programs, `ePiX`'s shell scripts are written using Unix-style pathnames. Thus, the most straightforward way to use `ePiX` is to install a Unix-like environment.

Alternatives for Windows

Version 0.8.9 of `ePiX` has been implemented in Python 2.2 by Andrew Sterian, making `ePiX` available on any platform that supports Python, and without requiring a `C++` compiler or `bash`. Python is a GPL-ed scripting language, and is available with a Windows installer and detailed instructions. The easiest alternative for Windows users is probably to install Python 2.2 or later (if necessary) and `Pyepix`. The `Pyepix` project home page is:

```
http://claymore.engineer.gvsu.edu/~steriana/Python/index.html
```

Cygwin, and theoretically the de Lorie tools, can be used to run `ePiX` under Windows. The following Cygwin packages are probably necessary and sufficient:

```
[] bash       [] GhostScript [] sh-utils
[] binutils   [] ImageMagick [] teTeX
[] fileutils  [] make        [] textutils
[] gcc        [] sed
```

The `emacs` and `gv` packages are highly desirable, but not absolutely necessary. The author has received scattered reports of success with Cygwin, but is not sufficiently knowledgeable to provide substantial support.

1.2 Installation

`ePiX` is distributed over the World-Wide Web as source code. Packages (stable and development) can be found at

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

In a web browser, `shift`-click on a link to download. The latest stable release is also on the CTAN mirrors, in the `graphics` directory. There are instructions for downloading an entire directory; it is not recommended that you download the files individually. (Some users of Red Hat have reported file permission problems when unpacking the CTAN tarballs. If you encounter this difficulty, please try downloading the sources from the project main page.) Unpack the compressed tar file with the appropriate command:

```
tar -zxvf epix-x.y.z_complete.tar.gz
tar -jxvf epix-x.y.z_complete.tar.bz2
```

(`x.y.z` is the version number) or, if your `tar` doesn't do decompression,

```
gunzip -c epix-x.y.z_complete.tar.gz | tar -xvf -
bzcat epix-x.y.z_complete.tar.bz2 | tar -xvf -
```

cd to the source directory, which is named epix-x.y.z. The INSTALL file contains detailed installation instructions. If you're impatient, the short of it is make [test]; make install. Respectively, these steps build the program, optionally run a test compile on the included sample files, and install the library, header file, and shell scripts.

There is an optional package, contributed by Svend Daugård Pedersen, that supplies extensions for enhanced Cartesian and logarithmic coordinate systems, and for hatching polygons and planar regions. To build this package, do make contrib before make install.

In order to use the contrib package, an input file must contain the line "using namespace ePiX_contrib;" For documentation, please see the directory $INSTALL/share/epix/tutorial/contrib.

By default, ePiX installs in subdirectories of /usr/local; if you want to install elsewhere, see the INSTALL file for detailed instructions. You may also want to consult POST-INSTALL for information on setting your PATH variable so your shell can find ePiX.

To re-iterate, ePiX is not a stand-alone program, but consists of a C/C++ library, header, and a shell script, and therefore requires a compiler *for normal use.* The GNU compiler (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. ePiX is also reliant on the GNU shell bash. Most Unices (and all major GNU/Linux distributions) have bash in /bin/bash. The INSTALL file explains how to cope with bash that is elsewhere. If you port ePiX to another shell or operating environment, or package ePiX for other systems, please notify the author so that your work can be linked from the project page and mentioned in the documentation.

Should it come to this, the command make uninstall will remove installed components of ePiX from your system. You must be in the source directory, and may need to log in as root.

Other Sources

An RPM spec file is maintained by Guido Gonzato. If you run a GNU/Linux distribution that uses RPM 4.x to manage installed packages (e.g., a recent version of Red Hat or Mandrake), you can build and install ePiX with the command (as root)

```
rpm -ta epix_complete.tar.gz
```

An `rpm` source file is available from the project page at Holy Cross. Separately maintained packages exist for FreeBSD and Gentoo:

```
http://www.freshports.org/graphics/epix/
http://packages.gentoo.org/search/?sstring=epix
```

The very latest source code for Versions 1.x and 2.x (a.k.a. The Next Generation) is available by CVS from

```
http://savannah.nongnu.org/cgi-bin/viewcvs/epix/epix/
```

The main project page may migrate to

```
http://savannah.nongnu.org/projects/epix
```

Development

There is a (currently very low-traffic) mailing list for all things `ePiX`-related:

```
ahwang-epix@mathcs.holycross.edu
```

You must be a subscriber in order to post, and a post must have a non-empty `Subject:` line. To subscribe, send an empty message to

```
ahwang-epix-subscribe@mathcs.holycross.edu
```

Please contact the author if you are interested in development.

Chapter 2

Getting Started

This chapter describes the basics of creating figures in `ePiX`, and is written for readers who are familiar with LaTeX but completely new to `C++`. No detailed knowledge of `C++` is needed to use `ePiX`, only a bit of grammar (roughly `C` grammar, without pointers) that is easily absorbed by example. As you read, please study the sample files that are distributed with `ePiX`. If you have installed in the default location, the samples are under `/usr/local/share/epix`. The quickest way to learn is to copy sample files to a convenient location and experiment by modifying them and seeing how your changes affect the figure's appearance.

2.1 Running `ePiX`

The sample and tutorial figures carry the preferred file extension, `.xp`. To convert `file.xp` to a viewable figure, do "`elaps file.xp`" to produce an `eps` file, or "`elaps --pdf file.xp`" for PDF. Like LaTeX, `ePiX` is non-interactive, and run from the command line. Each shell script accepts a `--help` (or `-h`) option that prints a short summary of available options. Under `X`, a graphical environment may be simulated by using `emacs` (with Jay Belanger's `ePiX` mode) to edit and compile files, and a previewer such as `gv` to view figures as they are processed. In `gv`, select "Watch file" from the "State" menu to have images update automatically.

While `elaps` produces output files suitable for immediate previewing, `epix` is far preferable for documents processed with LaTeX (rather than PDFLaTeX). The output of `epix` is `eepic`, an enhancement to the LaTeX picture environment that allows lines of arbitrary length, slope, and width. An `eepic` file is typically a few percent the size of a comparable `eps` file, and is a text file that can be read (and edited) with only knowledge of LaTeX.

An `eepic` file is included directly into a LaTeX document, say `sample.tex`, that contains the line

```
\usepackage{epic,eepic,pstcol}
```

Somewhere in `sample.tex` is the figure itself:

```
\begin{figure}[hbt]
  \begin{center}
    \input{example.eepic}
    \caption{A compiled \ePiX\ figure.}
    \label{fig:example}
  \end{center}
\end{figure}
```

In a text editor, create an `ePiX` input file for the figure, say `file.xp`, then issue the command

```
epix file.xp example.eepic
```

File extensions may be omitted, as can the name of the output file if you want it to be the same as the name of the input (`file.eepic` in this example).

The preferred extension for an `ePiX` file is "`.xp`" (for eXtended Picture), but the source code extensions "`.c`", "`.cc`", "`.C`", and "`.cpp`" are also permitted. If the `emacs` configuration file is properly set up, `emacs` can recognize "`.xp`" files, automatically enter `ePiX` mode, and insert a preamble template. The file `POST-INSTALL` has detailed instructions.

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`. `laps` stands for "LaTeX to Postscript".

Figures in graphical format (as opposed to `eepic`) can be useful if you use `xdvi` to preview (color is available in included files), or if you are tweaking the figures in a large document, and do not want to recompile the entire document to change just one figure. `elaps` also handles `eepic` files (i.e., acts as `eepic2eps` and `eepic2pdf`), even those not produced with `ePiX`.

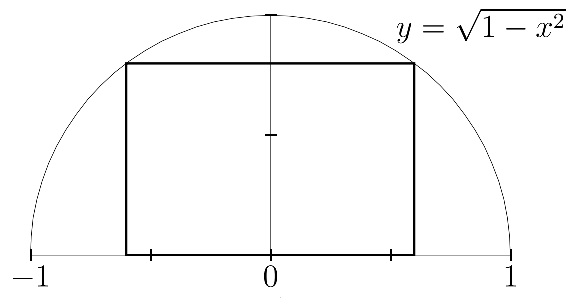


Figure 2.1: A semicircle.

2.2 A Sample File

Figure 2.1 is created from the commented source file `semicirc.xp`:

```
#include "epix.h"
using namespace ePiX;                    // similar to \usepackage

// "double" = double-precision floating point
double f(double x) { return sqrt(1-x*x); } // i.e., f(x) = \sqrt{1-x^2}

double width = 0.6, height = f(width);  // rectangle dimensions
P label_here(0.5, f(0.5));              // position of graph label

int main()
{
  unitlength("1in");
  picture(2.5, 1.25);                   // set printed size, 2.5 x 1.25in
  bounding_box(P(-1, 0), P(1, 1));      // corners; depict [-1,1] x [0,1]

  begin();   // ----- figure body starts here -----
  h_axis(4);                            // mark off 4 intervals, etc.
  v_axis(2);
  // 2 label intervals; shift down 4pt, place below Cartesian location
  h_axis_labels(2, P(0, -4), b);

  arc(P(0,0), 1, 0, M_PI);              // center, radius, start/end angle
  bold();                               // paths only, not fonts
  rect(P(-width,0), P(width, height));  // rectangle given by corners

  // shift label right 2pt, up 4pt, align on reference point (default)
  label(label_here, P(2,4), "$y=\\sqrt{1-x^2}$");
  end();
}
```

ePiX commands are of three types: definitions (of data and functions), attribute setting, and drawing. Like a LaTeX document, an ePiX file contains a *preamble*, which specifies aspects of the figure's appearance, and a *body*, which contains the actual figure-generating commands.

Variables have been used to give logical structure to the figure: The width of the rectangle and the the location of the label are localized to two lines in the preamble, and the rectangle's height is computed from the width. The capacity to structure figures logically is one of ePiX's strengths. In a file as short as this, hard-wired constants are adequate, but the importance of structuring increases rapidly with the size of the input file, and good habits are best formed from the start. The function definition is mostly for illustration, though does provide slightly better organization than

```
double width = 0.6, height = sqrt(1-width*width);
double lab_x = 0.5;
P label_here(lab_x, sqrt(1-lab_x*lab_x));
```

Command Syntax

Attribute-setting and drawing commands are C++ *functions*, blocks of instructions that can be invoked—or *called*—by name. Like mathematical functions, C++ functions accept *arguments* as input and *return* values. Each argument has a *type*, such as integer (int), real number (double, for "double-precision floating point"), or P (ePiX point), and may be hard-coded or specified symbolically. C++ is a "typed" language, meaning that the compiler checks function calls for the proper type and number of arguments, and issues an error if no matching function is found.

C++ allows functions to be given *default arguments*; when a function has defaults, the corresponding argument(s) may be omitted in a call. A command is usually described by listing the types and names of the arguments. For brevity, the type double may be left tacit. Default arguments are denoted by square brackets. The command

```
  line(P pt1, P pt2, [double stretch], [int n]);
```

defines a "line" command that accepts two mandatory arguments of type P, an optional real-valued stretch parameter, and an optional integer n. When a function is called in an input file, arguments' types are not given:

```
  line(P(-2,1), P(1,0), 6.4);
```

The last argument is assigned its default value in this case.

2.3 Basic Picture Concepts

This section outlines the basic typographical and mathematical data used to size and place a figure. The *preamble* of an ePiX input file is everything that comes before the begin() line; the *body* comprises the portion of the file between the begin() and end() lines, inclusive.

Printed Size and Location

LaTeX treats the contents of a picture environment as a single box, aligned by default on the lower left corner. An ePiX file must tell LaTeX how large the printed figure will be, and where to align this "picture box". The commands

```
picture(2.5, 1.25);
unitlength("1in");
offset(0.25,-0.5);
```

set LaTeX's unitlength to 1 in, create a picture environment 2.5 in wide and 1.25 in high, then shift the picture right by 0.25 in and down by 0.5 in. The picture and unitlength lines are mandatory in an ePiX preamble. The offset is optional and defaults to $(0, 0)$.

The argument of a unitlength command is a numerical constant followed by one of the following valid LaTeX length units: bp (big point), cm (centimeter), in (inch), mm (millimeter), pc (pica), pt (point, the default unit), and sp (scaled point). (There are 72 big points per inch, 12 points per pica, and 65536 scaled points per point.) ePiX does not directly support the use of different horizontal and vertical length units; you are expected to perform any conversions manually.

A non-zero offset causes the contents of a picture to appear in a location where LaTeX does not expect them. This may be useful when an eepic file is included into a LaTeX document, but requires visual tweaking. A non-zero offset is risky when compiling a figure into EPS or PDF with elaps, since dvips may crop the figure according to rules of its own.

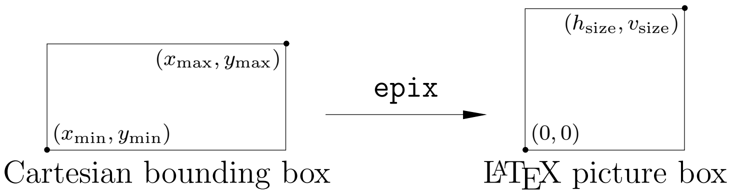
2.4 Logical Size, and Aspect Ratio

An ePiX figure occupies a rectangular Cartesian "bounding box". The lower left and upper right corners of the bounding box are known to ePiX as `(x_min,y_min)` and `(x_max,y_max)`, while the width and height are `x_size` and `y_size`. The bounding box is a virtual, advisory data structure; its dimensions are not directly related to the figure's printed size, and picture elements are not constrained to the bounding box by default.

The bounding box is specified in the preamble by giving a pair of opposite corners; the command

```
  bounding_box(P(-1,0), P(3,2));
```

sets the bounding box to be $[-1, 3] \times [0, 2]$. Either pair of opposite corners may be used, though confusion is less likely if the lower left and upper right corners are given. Affine scaling maps the bounding box to the picture box when the output file is written:



The figure's aspect ratio is controlled by sizing the bounding box. The aspect ratio is "true" if the bounding box and picture box are geometrically similar, e.g., if both boxes are 1.5 times as wide as they are tall.

2.5 Creating and Drawing Objects

Conceptually, `ePiX` builds a 3-dimensional virtual "world", then photographs the world on a 2-dimensional "screen". The screen contains the bounding box, which is affinely scaled to a LaTeX picture environment. By default, the screen is the (x_1, x_2)-plane and the world is renderd by looking straight down the x_3-axis.

Geometric Data Structures

The simplest object in the world is a point, represented by an ordered triple of real numbers (double-precision floats). The function `P(x1,x2,x3)` creates the point (x_1, x_2, x_3). If only two arguments are provided, $x_3 = 0$ by default. This convention allows `ePiX` to treat 2- and 3-dimensional figures uniformly. Often points are given names, so they can be used repeatedly throughout a figure. The (essentially equivalent) commands

```
  P pt(x_min, 2*x_min, 4);
  P pt = P(x_min, 2*x_min, 4);
```

create a point named pt and initialize it to $(x_{\min}, 2x_{\min}, 4)$, using the current value of $x_{\min}$. This value is used each time pt occurs subsequently. In general, an expression involving variables may be used to assign a value to a data structure (not just a point). However, a variable cannot be used until a value has been assigned. For example, the commands above will not work before the bounding box is set, since the value of $x_{\min}$ is unknown. Further, changing the value of a variable does not "update" the values of dependent data structures.

Points can be given in polar, cylindrical, or spherical coordinates; all arguments are numerical.

```
  P pt=polar(r, t);  // (r*Cos(t), r*Sin(t))
  P pt=cyl(r, t, z); // (r*Cos(t), r*Sin(t), z)
  P pt=sph(r, t, phi);
```

The arguments of sph are radius (distance to the origin), longitude (measured from the x_1-axis), and latitude (measured from the equator). By default, angles in ePiX are measured in radians. Two other "angular modes" are available: degrees and revolutions. The angular mode is set with an identically-named command, e.g., degrees(), and all trigonometric operations are affected.

ePiX provides algebraic operations on triples, including addition, scalar multiplication, the dot and cross products, and a few others. These operations are used to express relationships between triples, as in

```
  P p1(2,1), p2(-3,1);
  P q1 = p1-p2, q2 = 2*p1-3*p2, q3=q1*q2;
```

The points $q_1 = p_1 - p_2$, $q_2 = 2p_1 - 3p_2$, and $q_3 = q_1 \times q_2$ could be given hard-coded values. However, defining their values symbolically imbues the figure with logical structure, making the file easier to read, modify, and maintain. The standard basis is available: E_1=P(1,0,0), etc.

Practically, ordered triples of numbers are used to represent both *locations* (points) and *displacements* (vectors). Mathematically, the concepts are distinct; for example, it makes geometric sense to add two displacements (obtaining a displacement), or to add a displacement to a location (to get a location), but two locations cannot be added meaningfully. Algebraic operations act on vectors, not points. `ePiX` does not enforce the distinction between points and vectors, but will in The Next Generation.

`ePiX` implements data structures that represent line segments, circles, planes, and spheres. These structures can be translated, scaled, and intersected. A code snippet illustrates basic techniques:

```
circle C1(P(0,1), P(0,-1), P(0.5,0)); // circle through 3 pts
C1 += P(0.1,0);           // translate center
C1 *= 2;                  // double the radius

sphere S1(P(0,0,0), 1.5); // sphere of radius 1.5 at origin
plane P1(P(0,0,0), E_3);  // (x,y)-plane
circle C2 = S1*P1;        // circle of intersection
```

Use of these data structures is explained in more detail in Chapter 3.

Drawing

The commands of the previous section create data structures but do not write any output. Each type (other than `P`) is drawn with "object-oriented" syntax. For example, if `C1` is a circle, then the command `C1.draw()` draws the circle. The effect of drawing a `plane` or `sphere` is described in Chapter 3. `ePiX` also provides high-level commands that draw polygons, curves, and compound objects, such as arrows and coordinate axes. Drawing commands must come in the body of the file, after `begin()`.

```
line(P p1, P p2);
triangle(P p1, P p2, P p3);   // vertices specified
rect(P p1, P p2);             // coord rect w/opposite corners
quad(P p1, P p2, P p3, P p4); // arbitrary quadrilateral

spline(P p1, P p2, P p3);     // quad/cubic splines given
spline(P p1, P p2, P p3, P p4); // by control points

arc(P center, radius, t_min, t_max);
ellipse(P ctr, P v1, P v2, [t_min], [t_max], [int n]);
```

The arguments of `rect` must lie in a plane parallel to a coordinate plane; the sides of the rectangle are parallel to coordinate axes.

An `arc` has the given center and radius, and lies in a plane parallel to the (x_1, x_2)-plane. Angles are measured from the `E1` direction in the current angle units.

An `ellipse` draws an elliptical arc with the specified center and "axes"; precisely, the curve drawn is parametrized by

$$t \mapsto \mathrm{ctr} + (\cos t)v_1 + (\sin t)v_2, \qquad t_{\min} \leq t \leq t_{\max}.$$

(Note that `ctr` is a location, while `v1` and `v2` are displacements.) As with `arc`s, angles are measured in current angle units. If the parameter bounds are omitted, the entire ellipse is drawn. The final (optional) argument says how many points to use when drawing the arc. This can be omitted safely in most situations.

2.6 The Camera

Art students sometimes practice perspective by tracing on a window with grease pencil, a mathematical transformation called *point projection*. `ePiX`'s default mapping from the world to the screen is similar. Imagine standing before a scene at a *viewpoint*, and possessing X-ray vision, so that objects are transparent. Somewhere in front of you is a plane, the screen. The *target* is the point on the screen obtained by dropping a perpendicular from the viewpoint.

Three mutually perpendicular unit vectors sit at the target: sea, sky, and eye. The sea vector points to the right, sky points upward, and eye points straight from the target to the viewpoint. The bounding box of the figure is defined with respect to the screen's (Cartesian) sea-sky coordinate system.

Given a point p in front of the viewpoint, we want to determine the screen location to which p projects. Join p to the viewpoint by a line; this line intersects the screen plane exactly once, and the point of intersection is where we draw p in the screen.

At the start of a figure, the camera is initialized to lie on the x_3-axis at very large distance from the origin. The resulting view, essentially projection along the axis, is suitable for 2-dimensional figures. The camera is manipulated with object-oriented syntax:

```
camera.at(P posn);               // set viewpoint to posn
camera.look_at(P targ);          // set target
```

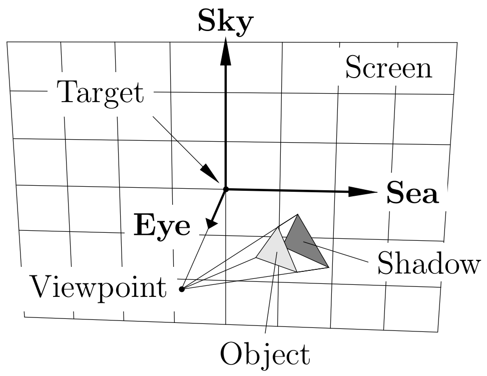


Figure 2.2: Point projection.

```
camera.range(double dist);       // fix target, move viewpoint
camera.focus(double dist);       // fix viewpoint, move target
camera.rotate_sea(double angle); // rotation about an axis
```

These commands must come in the figure body.

2.7 Drawing Attributes

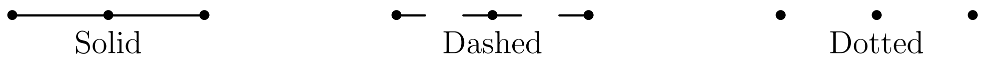
`ePiX` creates line drawings, not pixmaps. In an `eepic` file, objects are either pointlike (LaTeX glyphs, text boxes) or pathlike (everything else). The appearance of a pathlike object depends on the line style, width, filling, and color. Each attribute is a declaration, remaining in effect until superceded or the file ends. When a file starts, paths are drawn solid, black, unfilled, and with `thinlines` (a width of 0.4 pt) in effect. Attribute-setting commands should come in the figure body.

Path Width and Style

The standard path widths are `plain` (`thinlines`) and `bold` (`thicklines`). Other widths are available via a `pen()` command. The argument is a number, interpreted as a length in `pt`, or a number followed by a two-letter LaTeX length as in the `unitlength()` command. Journals discourage line widths smaller than about 0.5 pt, and a multitude of author-specified line widths tends to look cluttered and *ad hoc*. If possible, use the standard widths.

```
plain();         // thinlines, about the same as pen(0.4);
bold();          // thicklines,                  pen(0.8);
pen("0.02in");   // set path width to 0.02 in
```

The path *style* is one of `solid`, `dashed`, or `dotted`. The style is set with an identically-named command, e.g., `dashed()`. Every pathlike object in `ePiX` is drawn using a list of points. When the path style is `solid`, the points are joined connect-the-dots fashion. A `dotted` path is drawn by placing a small dot at each point of the path. When the style is `dashed`, line segments are drawn partway from each vertex to its neighbors.



A few parameters can be adjusted manually: the page distance between consecutive points on a path (for polygons—triangles, quads, etc.), the "dash density" (the percentage of a dashed path filled by dashes), and the dot size:

```
dash_fill(0.7); // dashes fill 70% of point gap
dash_length(6); // path points separated by 6pt
dot_sep(8);     // path points separated by 8pt
dot_size(2);    // dots 2pt in diameter
```

Color and Filling

An `ePiX` output file achieves color through the `pstcol` style, an amalgamation of `color` and `PSTricks` styles. There are three available color models: `rgb`, `cmyk` (cyan, magenta, yellow, and black), and named (primary colors only). An `rgb` color is specified by three numbers between 0 (no color) and 1 (full saturation), each of which represents the density of a primary color (red, green, and blue respectively). A `cmyk` color is similarly set by giving four densities. Finally, a primary color (r, g, or b; c, m, or y; black and white) may be specified by name, with an optional density.

```
rgb(1, 0, 1);   // magenta
cmyk(0,1,0,0);  // same thing
magenta();      // method III
rgb(1, 0.7, 0.7); // light red
rgb(0.4, 0, 0);   // dark red
red(0.4);         // same thing
```

Like all parameters, color densities can depend on variables. Values that lie outside the interval $[0, 1]$ are "clipped"; for example, `rgb(1.4,-0.05,2)` is also magenta.

The `fill()` command causes closed paths to be gray-shaded, using PostScript `specials`. The depth of gray ranges from 0 (white) to 1 (black), and defaults to 0.3. The command `gray(0.4)` sets the depth to 0.4. Filling is deactivated by the command `fill(false)`. The order of filled objects in the source file is significant, because `ePiX` writes its output in the same order, and PostScript has no transparency. Layering, hidden object removal, and color shading are discussed in Chapters 3 and 4.

2.8 Typography

In an `ePiX` file, points are denoted with LaTeX glyphs ("markers") or text boxes ("labels"). A marker occupies a box of zero size, and is placed at a specified Cartesian location. A label has typographical size, so its placement is more involved. When the size or aspect ratio of a figure is adjusted, the font size stays the same. In order to keep a label aligned properly over a range of sizes, a scale-invariant alignment point is attached to each label, and Cartesian coordinates are used to position the alignment point. The alignment point is controlled both by a true-pt offset and an optional LaTeX-style alignment option.

Markers

`ePiX` markers are called by name: ● • • ■ ▪ –○–∘–

```
  spot(P pt);  dot(P pt);   ddot(P pt);
               box(P pt);   bbox(P pt);
  ring(P pt);  circ(P pt);
```

`spot` and `(d)dot` are solid dots that cannot be colored. `box` and `bbox` are solid, colorable squares. A `ring` is colorable and transparent, while a `circ` is uncolorable and opaque. Markers in the same column are the same size, and each column is 1.5 times the diameter of the next. The diameter of a `dot` (hence that of all the above `markers`) is set with `dot_size(diam)`. The argument, 3 by default, is a number of `pt`. The glyphs listed in Table 2.1 are available via the command

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

Table 2.1: `ePiX`'s `marker` types.

```
marker(P pt, <MARKER TYPE>);
```

Labels

A *label* is a typographical box. Since a LaTeX box occupies a rectangle on the printed page, a single location is not enough information to position a label within a figure; an alignment point is needed in addition. By default, the alignment point of a text box is its reference point, the intersection of the left edge and the baseline, which is used by LaTeX to position the box on the page: $\boxed{y = f(x)}$ An alignment point may be shifted manually:

```
label(P(3,2), P(2,-1), "$\\rho=\\sin \\theta$");
```

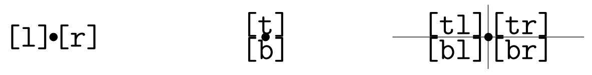
typesets the equation $\rho = \sin\theta$ and places the resulting text box at Cartesian location $(3, 2)$, but shifted right by `2pt` and down by `1pt`. Note that `C++` treats "\\" as a "backslash control sequence", so a double backslash is needed in the source to get a single backslash in the output. The general `label` commands are:

```
label(P posn, P offset, <label text>);
label(P posn, <label text>);
label(P posn, P offset, <label text>, align);
```

The first command prints a label with its LaTeX reference point at the Cartesian location `posn`, offset in true points (on the page) by the specified amount. Offsets of 2, 4, 6, 12, or 18 points work well with a 12-point font.

The second command prints the label text in a LaTeX box centered at `posn`. While this is perhaps the most obvious way of placing a label, it may not be the correct method, since labels often mark a geometric object that should not be covered by the label.

In the third command, the `align` option may be one—or an appropriate pair—of `t`, `b`, `r`, or `l` (top, bottom, right, left), or `c` (center). As with `offsets`, these alignment options specify the position of the label *relative to the Cartesian location p*, namely they work *opposite* to the way they work in LaTeX. For example, the alignment option `br` puts the label below and to the right of p.



Each label command has a corresponding "mask" version (`masklabel`) that draws an opaque white rectangle under the label text. Masking is useful when the text of a label sits in a cluttered part of the figure. An `eepic` file containing `masklabel`s requires the `pstcol` package.

Labels can be rotated; the angle is set in current angle units with the command `label_angle(theta)`. A rotation angle of 90 degrees prints labels suitable for a vertical axis. An `eepic` file containing rotated labels requires the `rotating` package.

When constructing and placing a label, keep in mind that

- Alignment offsets are specified in `pt` (i.e., page coordinates), not in Cartesian units, because the alignment point should not depend on the logical or printed size of the figure.
- The label text is enclosed in double quotes (the single character `"`), and contains the LaTeX code to generate the label. Backslashes are doubled.

2.9 `C++` Basics

An `ePiX` source file is a `C++` program. If you've successfully modified and compiled any of the sample files, you know enough `C++` to use `ePiX`. In the author's experience, `C` grammar suffices for most applications. An excellent introduction to definitions of functions and variables, control statements, and overall program structure is Kernighan and Ritchie's *The `C` Programming Language*, second edition [1].

Jay Belanger's `emacs` mode for `ePiX` inserts a file template when an empty buffer is opened with the extension `xp`. This section explains the purposes served by the template. A few additional remarks may help you avoid basic syntax pitfalls.

Comments

C++ has two types of comments. C-style comments, which may span several lines, are delimited by the strings /* and */. One-line comments, analogous to the LaTeX %, are begun with //. A one-line comment may appear within a multiline comment, but a C-style comment may not; the compiler will mistake the first */ it encounters as the end of the current multiline comment.

Pre-Processing

The compiler ignores nearly all whitespace (spaces, tabs, and newlines), which should be used liberally to make files easy to read. Other punctuation (periods, commas, (semi)colons, parentheses, braces, and quotes) dictates file parsing, and must adhere stringently to grammar.

A C++ file consists of "statements". A statement ends with a semicolon, and conventionally a file contains one statement per line (when possible). For historical reasons, an input file is "pre-processed" before the compiler sees it. The most important use of pre-processing in ePiX is file inclusion. An ePiX file always begins with the lines

```
#include "epix.h" // N.B. pre-processor directive, no semicolon
using namespace ePiX;
```

The first line is analogous to a LaTeX usepackage command: It causes the pre-processor to replace the line with the contants of a file, thereby importing the names of commands provided by ePiX. To avoid name conflicts, ePiX's commands are enclosed in a "namespace". For example, the label command is actually known to the compiler as ePiX::label. The second line above tells the compiler to apply the prefix tacitly.

Variables and Functions

Definitions of variables and functions play the same role in a figure that macro definitions do in a LaTeX document: gathering and organizing information on which the figure depends. A variable is defined by supplying its type, name, and initial value. By far the most common data types in ePiX are double, P, and int. The name of a variable may consist (only) of letters (including the underscore character) and digits, and must begin with a letter:

```
my_var, my_var2,  _MY_var, __, aLongVariableName;  // valid
my-var,    2var, v@riable, $x, ${MY_VARIABLE}; // not valid
```

Variable names are case-sensitive. There are numerous conventions regarding the significance of capitalization. Generally, make names descriptive but not unwieldy, and avoid names that begin with an underscore (unless you know what you're doing).

A function accepts "arguments" and "returns a value". To define a function in C++, you must specify the return type, the name of the function, the types of the arguments, and the algorithm by which the value is computed from the inputs. The code block

```
double f(double x)
{
  return sqrt(1-x*x);
}
```

specifies the double-valued function f of one double variable defined by the formula $f(x) = \sqrt{1-x^2}$. Several sample and source files (especially functions.cc) give more interesting examples. A function definition should be formatted as above for readability.

Program Execution

All the "action" in a C++ program occurs inside the special function main. Running a compiled C++ program is viewed by the operating system as calling the program's main function. The return value (an int) is the program's exit status. In an ePiX file, the main action usually consists of setting the logical and printed size of the figure, then building and drawing the figure, changing attributes when desired. ePiX output proper starts with begin() and terminates with end(). The intervening statements constitute the *body* of the figure.

In C++, a function may not be defined inside another function. Thus, variables may be defined inside main, but functions cannot be.

Raw output

More-or-less verbatim text can be printed to the output file. A single backslash is produced by a double backslash in the input file. Certain letters have special meanings when backslash-escaped, including "\n" (newline) and "\t"

(TAB). Unlike LaTeX, `C++` does not require a space to separate an escape sequence from following text; the string "`\textwidth`" is read "`TABextwidth`" by the compiler.

As an application, a complete LaTeX `figure` environment (with caption and label) can be produced by an `ePiX` file. Newlines must be added explicitly, and all printing statements must occur inside the call to `main()`.

```
#include "epix.h"
using namespace ePiX;
using std::cout;      // C++'s output function

int main()
{
  cout << "\\begin{figure}[hbt]\n";
  unitlength(...);    // picture, bounding_box, etc.
  begin();
  < ... ePiX commands ... >
  end();
  cout << "\\caption{A \\LaTeX\\ figure.}\n" // N.B. line con-
       << "\\label{fig:example}\n"           // tinues, no ";"
       << "\\end{figure}\n%%%%\n";           // LaTeX formatting
} // End of main()
```

Conditionals and Loops

An algorithm's behavior usually depends on some internal state. A *conditional statement* causes blocks of code to be executed depending on some criterion. A *loop* repeatedly executes a code block, usually changing the values of variables in a predictable way, so that the loop exits after finitely many traversals. Figure 2.3 illustrates both conditionals and loops with Euclid's algorithm for the greatest common divisor. Three pieces of notation require explanation: `j%i` means "$j \pmod{i}$", `||` is logical "or", and `==` is "test for equality". (A single "`=`" is the assignment operator.)

2.10 High-Level Picture Elements

`ePiX` implements a miscellany of high-level drawing capabilities: arrows, coordinate axes and axis labels, polar plots, data plotting from files, calculus op-

```
int gcd (unsigned int i, unsigned int j)
{
    int temp=i;
    if (i==0 || j==0)
        return i+j;    // define gcd(k,0) = k

    else {
        if (j < i)     // swap them
            {
                temp = j;
                j=i;
                i=temp;
            }
        // the work is done here...
        while (0 != (temp = j%i)) // assign temp, test for zero
            {
                j=i;
                i=temp;
            }
        return i;
    }
}
```

Figure 2.3: Euclid's division algorithm in `C++`.

erations, vector fields, solutions of differential equations, and non-Euclidean geometry.

Arrows

An arrow is specified by its tail and tip. An optional third argument scales the arrowhead.

```
arrow(P tail, P tip, [double scale]);
dart (P p1, P p2);  // same as arrow(p1, p2, 0.5);
aarrow(P p1, P p2); // double-headed arrow <--->
```

Pictorially, an `arrow` consists of a line segment (the shaft) surmounted by a triangle (the arrowhead). In profile, an arrowhead's width is `3pt`, and its height is 5.5 times the width. The actual printed height depends on the `arrow`'s orientation with respect to the camera.

By default, an arrowhead is a hollow triangle, which can be colored. The `fill()` command produces solid, uncolorable arrowheads. `PSTricks` commands can be used for solid colored `arrows`.

Coordinate Axes and Labels

A coordinate axis consists of a line between two points together with a specified number of regularly-spaced tick marks:

```
h_axis(p1, p2, n);    // n subintervals (n+1 ticks)
v_axis(p1, p2, n);
```

The style of tick mark is appropriate for an axis of the given type. If the endpoints are omitted, they default to $p_1 = (x_{\mathrm{min}}, 0)$ and $p_2 = (x_{\mathrm{max}}, 0)$ for a horizontal axis, or to $p_1 = (0, y_{\mathrm{min}})$ and $p_2 = (0, y_{\mathrm{max}})$ for a vertical axis. If the bounding box has integer width and/or height, then omitting the number of points draws tick marks one unit apart.

Labels for a horizontal axis are generated with:

```
h_axis_labels(P p1, P p2, int n, P offset, [alignment]);
```

This puts $(n+1)$ evenly-spaced labels on the segment joining `p1` and `p2`. The labels are automatically generated to match their horizontal location. As for ordinary labels, the `offset` is in `pt`, and the optional LaTeX-style alignment option places the labels using their corners rather than their reference points. Labels for a vertical axis are generated in the obvious way.

As for coordinate axes, the initial and final points may be omitted in an `axis_label` command, with the same defaults. However, the `offset` and number of labels must always be specified.

Coordinate Grids

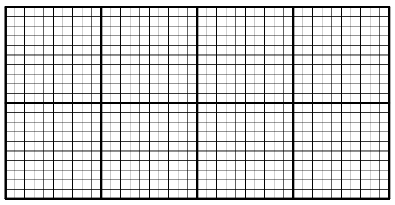
Cartesian grids fill a coordinate rectangle, and have a specified number of lines in each direction. A polar grid has specified radius, rings, and sectors.

```
grid(n1, n2);         // fills the bounding box
grid(p1, p2, n1, n2); // fills the box with corners p1, p2
grid(p1, p2, mesh(n1, n2), mesh(m1,m2));
polar_grid(r, n1, n2);
```

Each command draws an n1 by n2 grid. The third uses an $m_1 \times m_2$ mesh, which is useful only if the camera lens does not map lines in object space to lines on the screen.

Graph paper may be created by superimposing grids:

```
pen(0.25);
grid(10*x_size, 10*y_size);
pen(0.5);
grid(2*x_size, 2*y_size);
pen(1);
grid(x_size, y_size);
```



S. D. Pedersen's `contrib/` package provides enhanced Cartesian graphs. See `contrib/doc` in the source for documentation.

2.11 Basic Plotting

Because `eepic.sty` can draw lines of arbitrary length and slope, curves can be approximated by connect-the-dots paths. ePiX renders curves, polygons, and function graphs in this way.

For the moment, "function" means "function of one variable" (precisely, a `double`-valued function of a `double` variable). A function graph depends on the domain and the number of points to use. Each of the commands

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

graphs the function `f` on the interval `[t_min, t_max]` by dividing the interval into `n` subintervals of equal length. The first gives a Cartesian plot, the second a polar plot with bounds in current angular units, the third shades the region between the graph and the horizontal axis. If two functions are given to `shadeplot`, the region between their graphs is shaded.

Data plotting

Files of numerical data can be manipulated, analyzed, and plotted. The format for a data file is one or more floating-point numbers per line, with the same number of entries per line. A line in a data file that starts with a `%` is a comment.

Roughly, the basic plot command reads numbers from two (or three) columns of a specified file, treats them as coordinates, and plots the resulting points.

```
plot("filename", STYLE, columns, [i_1], [i_2], [i_3], [F]);
```

The first argument is the name of the data file. The `STYLE` may be `PATH`, which joins the points in the order they appear, or any of the marker types in Table 2.1. Next comes the number of columns (entries per line); if there are fewer columns than expected in the file, nothing is plotted, while if there are more columns than expected, a warning is issued but the data is plotted. The remaining arguments are optional. The integers i_k specify columns from which to extract data. The column entries are fed to the `P`-valued function `F` to obtain points. If `F` is omitted, it defaults to the Cartesian point constructor. If i_3 is omitted, the i_1 and i_2 columns are used to create points; by default, $i_1 = 1$ and $i_2 = 2$. For example, if `mydata.dat` contains 6 columns of numbers, then the respective commands

```
plot("mydata.dat", DOWN, 6);
plot("mydata.dat", BOX, 6, 2, 4, 5, sph);
```

plot the first two columns of `mydata.dat`, putting a "$\bigtriangledown$" at each point; and extract the second, fourth, and fifth columns of the file, treat them as spherical coordinates, and put a " ▪ " at each point.

For more elaborate (e.g., user-defined) analysis, data may be read into a `FILEDATA` structure. A `FILEDATA` is a `C++` vector of columns, each column having as many entries as there are lines of data in the file. The snippet below reads data from a file, then plots the result.

```
FILEDATA my_cols(6);          // vector of 6 columns
read("mydata.dat", my_cols);  // read data from file
...                           // code to mangle data
plot(my_cols, BOX, 2, 4, 5, sph);
```

The number of columns needn't be specified in the `plot` command, since it was provided when the data was read. The post-`TYPE` options are identical to the earlier plot command. The jth entry of the ith column is called `my_cols.at(i).at(j)`.

`ePiX` implements simple numerical functions of a `FILEDATA` structure:

```
  avg(my_cols, i_1);      // arithmetic mean of the i_1 column
  var(my_cols, i_1);      // variance of the i_1 column
  covar(my_cols, i_1, i_2);
  regression(my_cols, i_1, i_2);  // draw regression line
```

The covariance of two columns is obtained by subtracting the respective averages entry by entry, then taking the dot product. The regression line is the least-squares best fit for predicting column i_2 from column i_1.

Chapter 3

Reference Manual

This chapter covers the design and use of `ePiX`, assuming you've thoroughly digested the material in Chapter 2. Remaining features are documented, and the implementation described. If a feature isn't explained here, please consult the source code or contact the author.

3.1 More About `C++`

A textbook or similarly detailed reference is essential for serious study of `C` or `C++`. Go with the masters: *The `C` Programming Language*, second edition, by Brian Kernighan and Dennis Ritchie [1], is an excellent, manageable resource for the basics of procedural programming, while *The `C++` Programming Language*, by Bjarne Stroustrup [4], is a definitive, encyclopedaic reference.

`C++` is a powerful, complex language whose syntax is similar to that of `C`, or to the scripting languages of Maple and Mathematica. An `ePiX` input file is source code for a `C++` program that writes an `eepic` file as output. `ePiX` may be viewed as an extension to `C++`; in the same way that LaTeX furnishes a high-level interface to TeX, `ePiX` provides a high-level bridge between the computational power of `C++` and the LaTeX `picture` environment.

Like all high-level programming languages, `C++` provides variables, functions, and control structures. Variables hold pieces of data such as numerical values and geometric locations, while functions operate on data. A control structure, such as a loop or conditional statement, affects the program's course according to the program's current state. A source file is composed primarily of "statements", which perform actions ranging from defining variables and functions to setting figure attributes, performing calculations, and writing objects to the output file.

Names and Types

Names of variables and functions may consist (only) of letters, digits, and the underscore character. The first character of a name must not be a digit, and the language standard reserves names starting with underscore for library authors. Names are case-sensitive, but it's usually a bad idea to use a single name capitalized and uncapitalized in a single file. Numerous capitalization conventions are used informally; this document uses uncapitalized words separated by underscores for variables and functions, and occasionally uses all capitals for constants. As with names of LaTeX macros, primary considerations are clarity (of meaning), readability, and consistency.

Every variable in `C++` has a "type", such as integer (`int`), double-precision floating point (`double`), or Boolean (`bool`, true or false). `ePiX` provides additional types, the most common of which is `P`, for point. The construct `P(x,y,z)` creates (x, y, z), while `P(x,y)` gives $(x, y, 0)$, which is effectively the pair (x, y). A variable is defined by giving its type, its name, and an initializing expression.

In `C` and `C++`, a *pointer* is a variable that holds the memory address of another variable. Though more subtle than ordinary variables, pointers are useful in expressing certain algorithms, such as sorting. In Japan, buildings' addresses are assigned chronologically, rather than according to street location. A building is analogous to a variable, while the address is a pointer. If the Japanese parliament passed a law mandating that buildings be addressed consecutively along the street, there would be two ways to proceed: Dig up and relocate each physical building (move variables), or re-number the buildings in place (sort pointers). For similar reasons of efficiency, `C++`'s sorting algorithms work with pointers.

`C++` also provides *reference* variables, which allow a variable to be given an additional name. Their use arises because of the way `C++` functions treat their arguments.

Functions

In a programming language, the term "function" refers to a block of code that is executable by name. A `C++` function takes a list of "arguments", and has a “return value”. This information, together with the function’s name, must be provided when a function is defined. A function may not be defined inside another function. However, a function may call other functions (including itself) as part of its execution:

```
int factorial(unsigned int n)
{
  if (n == 0) return 1;
  else return n*factorial(n-1);
}
```

The special type `void` represents a “null type”. A function that performs an action but does not return a value has return type `void`. A function that takes no arguments may be viewed as taking a single `void` argument.

Every `C++` program has a special function `main()`, which is called by the operating system when the program is run. The arguments of `main()` are command-line arguments, and the return type is an integer that signals success or failure. User-specified functions must be defined before the call to `main()` or in a separately-compiled file.

Functions in `C++` may be as simple as an algebraic formula or as complex as an arbitrary algorithm. Greatest common divisors, finite sums, numerical derivatives and integrals, solutions of differential equations, recursively generated fractal curves, and curves of best fit are a few applications in `ePiX`. Several sample files contain user-level algorithms, which do not require knowledge of `ePiX`’s internal data structures. The source file `functions.cc` contains simple functions defined by algorithms, and `functions.h` illustrates the use of `C++` templates. Other source files, such as `plots.cc`, may be consulted for Simpson’s rule, Euler’s method, and the like.

An error, such as division by zero or an attempt to intersect parallel lines, may occur when a function is executed. In this situation, a `C++` function can “throw an exception”, or return an error type that the caller “catches” and handles. If an uncaught exception is thrown, the program terminates. `ePiX`’s intersection operators throw exceptions when certain conditions are not met.

Mathematical Functions

`C++` knows several familiar mathematical functions by name:

```
sqrt   exp   log   log10   ceil   floor   fabs
```

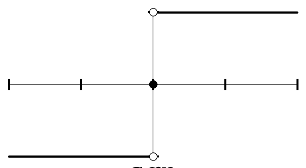
(`fabs` is the absolute value for a floating-point argument.) `ePiX` provides trig functions and their inverses that are sensitive to angular mode:

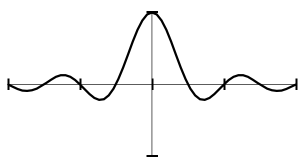
`Cos`	`Sin`	`Tan`
`Sec`	`Csc`	`Cot`
`Acos`	`Asin`	`Atan`

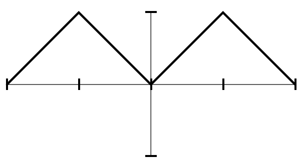
The inverse trig functions are principle branches.

The function `pow(x,y)` returns x^y when $x > 0$, and `atan2(y,x)` (N.B. argument order) returns $\mathrm{Arg}(x + iy) \in (-\pi, \pi]$, 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. `ePiX` defines a few additional functions:

`recip` `sgn` `zero` `sinx` `cb` `id` `proj1` `proj2`


`sgn`


`sinx`


`cb`

`recip` is the reciprocal, defined to be 0 at 0; `sgn` is the signum function; `zero` is the constant function; `sinx` is the function $x \mapsto \sin(x)/x$ with the discontinuity removed; `cb` (for "Charlie Brown") is the period-2 extension of the absolute value function on $[-1, 1]$; `id` is the identity mapping, defined for an arbitrary data type; the `proj` functions return their first and second variable, regardless of type.

The GNU `C++` library defines other functions, including inverse hyperbolic functions (`acosh`, etc.), `log` and `exp` with base 2, 10, or arbitrary b (`log2`, etc.), the error and gamma functions (`erf` and `tgamma` [sic], respectively), and Bessel functions of first and second kind: `j0`, `j1`, `y0`, etc. Use, e.g., `jn(5, )` to get higher indices. The GNU `C` library reference manual [2] describes these and other functions in detail.

Functions may be used in subsequent definitions, and 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)*Sin(t); }
```

Basics of Classes

Unlike `C`, `C++` supports "object-oriented programming". In a nutshell, a *class* is an abstraction in computer code of some concept, such as a point, a sphere, a mapping that can be plotted, or a camera. Implementationally, a class consists of *members* (named data elements) and *member functions* (functions that belong to the class and have free access to members). The idea is to encapsulate objects and the operations that make sense for them in a single logical entity. In simple programming, classes may be treated like built-in types.

`C++` classes enforce access permissions on their members, protecting data from being manipulated except in controlled ways. A class's member functions (and "public" members) constitute its *user interface.* The concept of class separates logical aspects of a data type from a particular implementation.

Each "instantiation" of a class has its own member functions. A member function therefore knows "which object" it was called on, and the call syntax differs from standard function calls:

```
circle C1(P(1,0), 1.5); // circle of given center and radius
C1.draw();              // member function circle::draw();
```

Naturally, this call draws the circle `C1`.

A few short paragraphs cannot do more than scratch the surface of classes and object-oriented programming. Please consult a book, such as Stroustrup [4], for details.

References and Function Arguments

`C` and `C++` are "call by value" languages. Actual variables are not passed to a function; instead a copy of the value is made, and the function operates only on the copy. Though this feature causes occasional inconvenience, it prevents a variable from having its value changed by a function.

A function may accept arbitrary data structures as arguments. If a data structure is complicated and a function is called frequently (thousands or millions of times), implicit copying becomes a source of inefficiency. References bypass this problem: If a variable is passed by reference, only a pointer need be copied.

A variable that is passed by reference may be altered by the calling function. This technique of updating variables is often touted as a feature in C++ texts; however, such trickery circumvents the data encapsulation of calling by value. Passing a variable by reference so that a function can modify the value of a variable is considered less than ideal programming practice.

Overloading

C++ provides "overloading": Multiple functions can be given the same name, so long as the number and/or type of their arguments differ. (It is *not* enough for the return types alone to differ. The compiler must be able to select a function from its calling syntax.) To the user, the appearance is that a single function intelligently handles multiple argument lists. Naturally, overloaded names should refer to functions that are conceptually related. Overloading tends to be most useful in library code; ePiX provides numerous plot functions, for instance.

Scope

A C++ statement ends with a semicolon. A collection of statements enclosed by curly braces is a "code block", and may be viewed as a single logical statement. Curly braces determine a "scope", inside of which variable names may be re-used without ambiguity. A variable defined between curly braces is said to be *local* (to the scope in which it is defined); its value cannot be used out of scope.

Function bodies are code blocks, as are the alternatives associated to control statements. The compiler is not picky about spaces, tabs, and newlines, so an input file should be organized in a way that makes the file easy to read. Indentation signifies levels of nesting within code blocks, but specific details are the focus of passionate debate. As with variable naming, clarity and consistency are the important criteria.

Headers and Pre-Processing

A C++ source file is compiled in multiple stages that occur transparently to the user. The first step, pre-processing, involves simple text replacement for file inclusion, macro expansion and conditional compilation. Next, the source is compiled and assembled: Human-readable language instructions are parsed, then represented in assembly language. Finally, the object files are linked: Function calls are resolved to hard-coded file offsets, possibly involving external library files, and the program instructions are packaged into an executable binary that the operating system can run.

Pre-processing is used much less in `C++` than in `C`; the language itself supports safer and more featureful alternatives to macros, such as `const` variables and inline functions. File inclusion and conditional compilation are the chief uses of the pre-processor. Lines of the form

```
  #include <cstdlib>
  #include "epix.h"
```

cause the contents of a *header file* to be read into the source file. A header file contains variable and function *declarations*, statements that specify types and names but do not define actual data. Declarations tell the compiler just enough to resolve expressions and function calls without knowing specific values or function definitions.

Conditional compilation is similar to conditional LaTeX code, and is best explained by example. A file might be used to produce both color and monochrome output as follows:

```
//#define COLOR  // uncomment for color
#ifdef COLOR
  ...  // code for generating color figure
#else
  ...  // monochrome code
#endif // COLOR
```

The “compiler symbol” `COLOR` is an ordinary `C++` name; compilation is controlled by commenting or uncommenting the `#define` line. Multiple decisions on the same symbol may appear in a file. An `#else` block is optional, but every `#ifdef` must have a matching `#endif`. Commenting the `#endif` is a good habit; in a realistic file, the start and end of a conditional block may be separated by more than one screen.

Comparison with LaTeX Syntax

As a programming language, `C++` provides certain features common to all languages (such as LaTeX, Metapost, Perl, Lisp. . .) and adheres to rules of grammar. Salient differences between LaTeX and `C++` include:

1. Every C++ statement and function call must end with a semicolon. An omitted semicolon may result in a cryptic error message from the compiler. Pre-processor directives, which start with a #, do not end with a semicolon.

2. Backslash is an escape character in C++:

   ```
   // 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
   ```

3. Variable and function names may contain letters (including underscore) and digits *only*, are case sensitive, and must begin with a letter.

4. Variables in C++ must have a declared *type*, such as `int` (integer) or `double` (double-precision floating point). If a variable has global scope and its value does not change, the definition should probably come in the preamble or at the beginning of `main`. Local variables should be defined in the smallest possible scope. Unlike C, C++ allows variables to be defined where they first appear.

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

6. C++ has single- and multi-line comments. Everything between a double slash and the next newline is ignored, while the strings `/*` and `*/` delimit multi-line comments. A single-line comment may appear within a multi-line comment, but the compiler does not nest multi-line comments.

Between them, C and C++ have about 100 reserved keywords which **cannot** be used as function or variable names. The script `keywords` packaged with `ePiX` is a simple lookup utility, meant to help you avoid name clashes. To find keywords containing the string `type`, for example, do “`keywords type`”.

3.2 The Camera

`ePiX` depicts a Cartesian world by projecting mathematically to a screen plane, then affinely scaling to a printed page. The camera, which maps the world to the screen, consists of a *body* (data that determines the position and orientation of the camera) and a *lens* (the actual mapping to the screen plane).

The Body

The camera's spatial orientation is described by a triple of mutually perpendicular unit vectors. In memory of happy days at the beach, these vectors are called *sea*, *sky*, and *eye*. The screen plane is parallel to the sea-sky plane; the sea vector points horizontally to the right, sky points vertically upward. The eye is their cross product, which points directly at the viewer.

The sea-sky-eye basis is located at the *target*, so the target is the origin of the screen plane. The *viewpoint* lies on the line through the target in the direction of the eye vector, and is the center of projection for the default lens. The distance from the viewpoint to the target is the *range*. The orientation, viewpoint, target, and range completely (and redundantly) determine the camera's geometric situation in the world.

The Lens

The *lens* is a mapping from the world to the screen. `ePiX` comes with three lenses: *shadow* (the default), *fisheye*, and *bubble*. Each lens simulates the appearance of world objects as seen by an observer at the viewpoint. The shadow lens is point projection from the viewpoint to the screen plane. Each of the other lenses performs radial projection to a sphere centered at the viewpoint, then maps the sphere to the screen plane; the fisheye lens does orthogonal projection (so the entire image lies inside a disk, and positions behind the camera are inverted) while the bubble lens does stereographic projection from the point directly behind the viewpoint.

Other lenses may be defined; the file `camera.cc` may be consulted as a template. Syntactically, a lens is a `pair`-valued function of a single `P` argument. To ensure expected behavior a lens should respect the meaning of the camera body.

Manipulating the Camera

The camera is a `C++` class, manipulated by member functions. The `begin()` command initializes the camera, so all changes of camera must occur in the body of the figure. By default, the viewpoint is at large distance on the positive x_3-axis, looking down on the (x_1, x_2)-plane. This provides the expected behavior for 2-dimensional figures.

The viewpoint and target may be set individually. In general, either operation changes the direction of the eye vector, which forces `ePiX` to re-determine the sky vector. When possible, the x_3-axis projects parallel to the sky; otherwise the x_2-axis is used. The viewpoint and target can be moved along the eye axis, changing the range while preserving the orientation: `range()` fixes the `target`, and `focus()` fixes the viewpoint. Each command re-sizes the image; note that increasing the focus *enlarges* the image.

The camera may be rotated about any of its axes. Axes of rotation pass through the target, so rotations about the sea or sky vectors change the viewpoint. For best control of the camera, set the target first, then the viewpoint. If desired, perform eye rotations last.

3.3 Clipping and Cropping

`ePiX` provides two masking operations to handle figure elements that lie far from the target: clipping (in the world) and cropping (in the screen). The "clip box" may be regarded as a set of "walls". When clipping is active, objects outside the walls are not visible. By default, the clip box is a large cube centered at the origin.

The "crop box" is a rectangle in the screen plane. When cropping is active, objects that project outside this rectangle are not visible. By default, the crop box is the bounding box. Since the figure is drawn (on the page) by affinely scaling the bounding box to a specified LaTeX box, default cropping ensures that a figure lies inside the printed region allocated by LaTeX.

By default, clipping and cropping are off. The command `clip(bool)` (de)activates clipping. The argument defaults to `true`. The clip box may be set with the commands

```
clip_box(P pt1, P pt2); // opposite corners
clip_box(P pt);         // opposite corners pt and -pt
clip_to (P pt);         //                  pt and P(0,0,0)
```

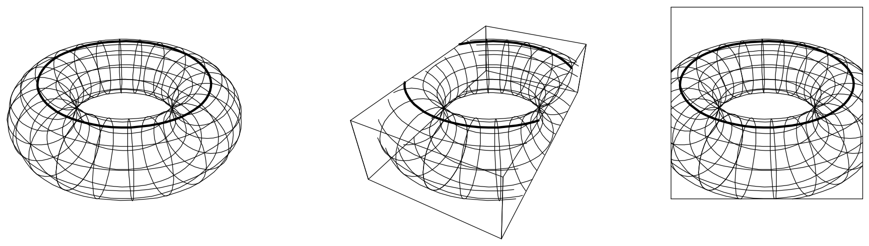


Figure 3.1: Clipping and cropping a torus mesh (boxes added).

Analogously, cropping is (de)activted with `crop(bool)`. The crop box is given by a pair of opposite corners, `crop_box(pt1,pt2)`; third coordinates are discarded. With no arguments, the command `crop_box()` re-sets the crop box to the bounding box.

3.4 Attributes

At a minimum, `ePiX` must be told how large the printed figure will be, and how large a Cartesian rectangle to allocate. The preamble must contain enough information to create a working state. In the body of an input file, the "drawing state" determines the figure's appearance. Attributes are declarations, set by commands that accept arguments of the stated type, possibly `void`. Color and path width are controlled by writing LaTeX commands to the output file immediately; the remaining attributes are managed internally.

- Angular mode: `radians()`, `degrees()`, or `revolutions()`.
- Path thickness: `plain()`, `bold()`, `pen(double)`.
- Path style:
  - `solid()`
  - `dashed(double)`. Optional argument is the dash density, the fraction (0.05–0.95) of the path taken up by dashes.
  - `dotted(double)`. Optional argument is the diameter (in `pt`) of a `dot`.

  The commands `dash_length(double)` and `dot_sep(double)` set the distance (in `pt`) between vertices of a dashed or dotted path.

- Color: `rgb(densities)`, `cmyk(densities)`, `primary(density)`.
- Filling: `fill(bool)`, argument defaults to `true`.
  - Gray depth: `gray(double)`, 0=white, 1=black.
  - PSTricks (q.v.) filling style, fill color.
- Text rotation: `label_angle(double)`
- Clipping and cropping (q.v.)
- Camera (q.v.)

Angular Mode

`ePiX` has three angular modes: `radians()` (the default), `degrees()`, and `revolutions()`. These modes affect all trigonometric operations, including camera rotations, the drawing of arcs and ellipses, polar plotting, label angle, and the trig functions themselves. Angle-sensitive trig functions are capitalized, e.g., `Cos`, `Tan`.

Color and Shading

`ePiX` provides color output via the `pstcol` package, using the `rgb` and `cmyk` models. Gray shading of regions is supported through `eepic.sty` (without requiring `pstcol`). Colors are best previewed by converting the document to Postscript or PDF. Alternatively, EPS files can be previewed in `xdvi`.

An `rgb` color is determined by three floating-point densities between 0 (no color) and 1 (full saturation). A `cmyk` color is similarly specified by four floats. Densities outside the range $[0, 1]$ are "clipped". Like line style, a color remains in effect until superseded. Seven primary colors and white are available by name. (Drawing in `white` can be used like correction fluid to remove pieces of a figure accurately.)

```
red();             // rgb(1,0,0);
magenta(0.6);      // cmyk(0,0.6,0,0);
rgb(0.2,0.7,0.8); // custom color
```

PSTricks

PSTricks is a powerful collection of macros, by Timothy van Zandt [5] and others, for incorporating PostScript in LaTeX. `ePiX` uses PSTricks primarily for color filling, but does not yet work seamlessly. PSTricks should be used in a file only when absolutely necessary. **The discussion below assumes PSTricks is needed in the file**. The command

```
use_pstricks(bool);
```

sets an internal flag that determines whether a path is drawn as an `eepic path` or as a PSTricks `psline`. When issued before `begin()`, this command also synchronizes the PSTricks `unitlength` and line width with `ePiX`, and sets the fill style to solid. PSTricks may be activated and deactivated freely, but *the first activation must occur in the preamble.*

While `pstricks` is active, commands of the form

```
fill_color("<color name>");
psset("<pstricks command>");
```

are used to set attributes such as the style and color of paths and filling. The default fill color is `"white"`. This snippet, taken from the sample file `contour.xp`, shows how to define a new color and use `psset()`.

```
use_pstricks();      // synchronize length, etc.
begin();
use_pstricks(false); // temporarily disable
...
std::cout << "\n\\newrgbcolor{orange}{1 0.7 0.2}";
psset("fillcolor=orange, linecolor=green, linewidth=1.5pt");
```

The PSTricks manual should be consulted for information.

Two major incompatibilities involve and filling and color. Unless the fill style is set explicitly to `none`, PSTricks fills all paths, even if they are not closed. Second, PSTricks manipulates colors by named strings, so raw output is required to exploit the full power of PSTricks from `ePiX`.

3.5 The Path Class

A `path` data structure is `ePiX`'s low-level ordered list of points that can be cropped, clipped, mapped, concatenated, and drawn. Raw path data is useful for complicated paths built in pieces. Available constructors are:

```
path(p1, p2);          // line (endpoints)
path(p1, p2, p3);      // quadratic spline
path(p1, p2, p3, p4);  // cubic spline
path(p1, v1, v2, t_min, t_max); // cf. ellipse arc
path(f, t_min, t_max); // graph or parametrized path

polyline(n, &p1, ...); // n points, followed by pointers
polygon(n, &p1, ...);  // same, but marked as closed
```

The first argument of the parametrized path constructor is a real- or vector-valued function of one variable. In each of the first five constructors, an optional final argument may be supplied to specify the number of points used.

The polyline and polygon constructors accept an unknown number of arguments; consequently, their arguments must be passed as "pointers", or memory addresses:

```
polyline(2, P(0,0), P(1,1)); // wrong; object arguments

P p1 = P(0,0), p2=P(1,1);
polyline(2, &p1, &p2);       // right; pointer arguments
```

This makes polylines and polygons inconvenient for quick-and-dirty use, but imposes little burden if the vertices have been defined elsewhere, as they might be in a logically structured file.

Compound paths may be built by concatenation. If `path1` and `path2` are paths, then the commands

```
path1 += path2;
path1 -= path2;
```

replace `path1` with the result of traversing `path1` "forward", then following `path2` in the forward or reverse direction (respectively). The notation is meant to suggest 1-dimensional homology chains. The sample file `contour.xp` illustrates path creation and manipulation. Finally, note that a `path` is a data structure, not a drawing command. The `path::draw()` function must be called explicitly to create visible output.

Path-Like Objects

Path-like objects comprise polygons with a fixed number of vertices (lines, triangles, quadrilaterals) and objects built from them (including coordinate axes, grids, and arrows); curves with a variable number of points (ellipses, arcs, splines); and plots of functions of one variable.

Fixed-data polygons are drawn with high-level commands.

```
line(p1, p2);
triangle(p1, p2, p3);
quad(p1, p2, p3, p4);
rect(p1, p2);              // coordinate rectangle
spline(p1, p2, p3);        // quad spline with control pts
spline(p1, p2, p3, p4);    // cubic spline
```

A `line()` accepts an optional numerical argument that acts as an expansion parameter: `line(p1,p2,t);` draws a segment with midpoint at the midpoint of `p1` and `p2`, but having length scaled by $2^{t/100}$. (That is, $t = 100$ doubles the length, while $t = -100$ halves the length.) The arguments of `rect()` must lie in a plane parallel to a coordinate plane.

Internally, `ePiX` marks paths as closed or not; triangles, quadrilaterals, polygons, ellipses, and arrows are closed. If a solid, closed path is drawn while filling is active, the path is filled with the current shade of gray. If filling is deactivated, the path is drawn but not filled. Dashed and dotted paths cannot be filled in one step; instead, fill the solid path, then draw the dashed/dotted boundary.

The shade of gray is a number between 0 (white) and 1 (black), and defaults to 0.3. Shading is opaque in PostScript, so the order of figure elements is significant in the input file when filling is active.

Color filling is available only through PSTricks. The features of PSTricks that are relatively well-supported in `ePiX` are illustrated in the sample file `contour.xp`. In principle, any PSTricks features can be obtained through raw output. However, this approach is generally discouraged since, for example, PSTricks color declarations are not recognized by `eepic`, and *vice-versa.* The Next Generation of `ePiX` will work more seamlessly with PSTricks.

Elliptical arcs are specified by their center, a pair of vectors, an angular range, and an optional number of points:

```
ellipse(ctr, v1, v2, t_min, t_max, n);
```

uses $n+1$ points to draw the parametrized path

$$t \mapsto \mathrm{ctr} + (\cos t)v_1 + (\sin t)v_2, \qquad t_{\min} \leq t \leq t_{\max}$$

If omitted, the number of points defaults to 80 for a full turn, with proportionally fewer points for an arc. When the angular range subtends one or more full turns, the ellipse is marked as closed, and will be filled if filling is active.

The circular arc parallel to the (x_1, x_2)-plane, having center p_1 and radius r, and subtending the angle (counterclockwise, in current angle units) from θ_1 to θ_2 is drawn with

```
arc(p1, r, theta1, theta2);
arc_arrow(p1, r, theta1, theta2); // same, with arrowhead
```

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.

3.6 Geometric Data Structures

Figure elements have an abstract description and a typographical appearance. This section describes the former. `ePiX` provides classes, `P`, `segment`, `circle`, `plane`, and `sphere`, that can be used for Euclidean geometry constructions. Each class is specified by a small amount of data (e.g., a center, radius, and unit normal vector for a `circle`) and provides constructors, intersection operators, affine transformations, and a `draw()` function.

`ePiX` also provides features for spherical and hyperbolic geometry, especially the ability to draw lines in the half-plane and Poincaré disk models of the hyperbolic plane, and to draw latitudes, longitudes, and parametrized curves on a sphere.

Triples and Frames

Points and displacements in space are represented as ordered triples. The name of the type is "`P`", though "`triple`" may be used for backward compatibility. Spherical and cylindrical/polar constructors are provided. Standard (and non-standard) algebraic operations—addition/subtraction, scalar multiplication; dot, cross, and componentwise products, and orthogonalization—can be performed on triples. When forming symbolic expressions involving triples, scalars must be collected together at left, triples at right. If necessary, use parentheses to force a particular association.

```
P pt(x,y,z);         // define pt = (x,y,z)
double u=pt.x1();    // first coordinate of pt, etc.
P(x,y);              // same as P(x, y, 0);
polar(r, theta);
cis(t);              // polar(1, t), aka Cos(t) + i*Sin(t)
sph(r, theta, phi);  // theta=longitude, phi=latitude
P(a,b,c)|P(x,y,z);   // dot product, ax+by+cz
P(a,b,c)&P(x,y,z);   // componentwise product, P(ax,by,cz)
p*q;                 // cross product, p x q
J(p);                // quarter turn about the x3-axis
p%q;                 // orthogonalization, p (mod q)
```

Explicitly, `p%q` is the unique vector `p+k*q` that is perpendicular to `q`.

A `frame` is a set of three mutually perpendicular unit vectors. The standard `frame` is the set `E_1`, `E_2`, `E_3`. The constructor turns three non-coplanar vectors into a `frame`:

```
frame(); // the standard frame
frame(P v1, P v2, P v3); // orthonormalize (v1, v2, v3)
```

The third vector of the new frame is positively proportional to `v3`, the second vector is positively proportional to `v2%v3`, and the first vector is the cross product. Thus, a `frame` is right-handed, and does not depend on `v1`.

Intersection

The concept of *genericity* is central to understanding intersections of geometric data structures in `ePiX`. For a working definition, two objects that are disjoint, tangent, or coincident intersect "non-generically". (Geometers will note that this differs substantially from the usual definition.) `ePiX`'s intersection operators throw exceptions when the operands are non-generic. If a run of `epix` terminates with an error message, check that you are not trying to intersect badly-formed or situated objects.

In `ePiX`, a `segment` is extended into a line for purposes of intersecting. Table 3.1 lists types of (generic!) intersections in `ePiX`. Intersection is commutative, so only the top half of the table is shown. Not all intersections are defined.

*	segment	circle	plane	sphere
segment	P	segment	P	—
circle		segment	segment	—
plane			line	circle
sphere				circle

Table 3.1: `ePiX`'s intersection types.

Segments

A `segment` is an *unordered* pair of points. A segment may be translated by a `P`, and the midpoint taken:

```
segment L1 = segment(P(0,0), P(2,4));
P mid = L1.midpoint();
segment L2(mid, P(-2,3)); // form segment
L2 += P(1,0);             // translate L2 by (1,0)
L1.draw(); L2.draw();
dot(L1*L2);               // point of intersection
```

Circles

A `circle` data structure consists of a center, radius, and a perpendicular unit vector. Three constructors are provided:

```
circle(P center, double radius, P normal);
circle(P center, P point);
circle(P p1, P p2, P p3);
```

Unspecified trailing arguments in the first constructor take the following defaults: If `normal` is not given, it is set to `E_3`; if in addition, the radius is unspecified, it is unity; finally, the center is the origin if no arguments are supplied. As usual in `C++`, only trailing arguments may be left implicit; the call `circle(center, normal)` does *not* create a unit circle with the given center and normal.

The second creates a `circle` parallel to the (x_1, x_2)-plane with the given center and passing through the given point. The third returns the `circle` passing through the given points. Exceptions are thrown if the `center` and `point` do not lie in a plane parallel to the (x_1, x_2) plane, or if the three points p_i are collinear.

A `circle` may be translated by a `P` with the $+$ operator, or scaled by a `double` using the $*$ operator:

```
circle C1=circle();      // unit circle
circle C2 = C1+P(1,0.5); // translate up and right
C2 *= 1.5;               // multiply radius by 1.5
C1.draw(); C2.draw();    // view handiwork
(C1*C2).draw();          // draw segment of intersection
```

Spheres

A `sphere` is specified by a point and a radius, by default the origin and unity. Constructors are provided for a `sphere` of specified center that passes through a given point, and for a `sphere` given by a pair of antipodes:

```
sphere(center, point);
poles(north, south);
```

As for `circle`s, translation and scaling operators are provided. Capabilities specific to geography and spherical geometry are described in Section 3.8.

The `draw()` function of a `sphere` draws the horizon visible from the current viewpoint. While this horizon is a circle in object space, its image in the screen plane is generally an ellipse. Further, antipodal points are not generally mapped to points that are symmetrically placed with respect to the center of this ellipse. These effects are most pronounced when the viewpoint is close to the `sphere` and the center is not close to the `target`.

Lines and Planes

`ePiX` does not have a separate data structure for lines, but it does provide a function, `Line`, that draws the line joining a pair of points. Naturally, only a segment can be drawn. `ePiX` always `crop`s a `Line`, and in addition removes the half-line that lies behind the observer. Thus, a `Line` appears as a printed segment, either with ends on the bounding box, or with at most one end in the interior, representing a point on a visible horizon. The Next Generation (a.k.a. `ePiX3d` or "Version 2.x") will supply a line data structure.

A `plane` is specified by a point and a unit normal vector, or by three non-collinear points. The `draw()` function renders the lines of intersection of the `plane` with the faces of the clip box. Unless the clip box has been set manually, these lines of intersection will almost surely avoid the bounding box, and therefore be invisible.

3.7 Domains and Plotting

To fix terminology, the noun "map" will be used for a `C++` function that accepts one or more `double` arguments or a `P` argument, and returns a `double` or a `P`. Maps can be depicted in two mathematically useful ways: graphing (which retains information about the domain), and drawing the image as a parametrized curve or surface (which discards domain information). Usually, a `double`-valued map is graphed, while a `P`-valued map is viewed parametrically. The verb "plot" is used generically to mean either sort of depiction, and the noun "plot" refers to the result of plotting.

The "domain" of a map is the set of allowed input values. In `ePiX`, a `domain` is a coordinate box of dimension one, two, or three. For uniformity, a 2-dimensional domain is actually a 3-dimensional domain whose "thickness" along the x_3-axis is 0. Similarly, a 1-dimensional domain is a "highly degenerate" 3-dimensional domain. Domains facilitate the plotting of families of mappings, and the selective plotting of parts of a mapping; `domain` operators for these purposes are described below.

A plot depends on a map and a domain. For example, suppose `f` is a `P`-valued map of two (`double`) variables, and `R` is a two-dimensional domain. To construct a plot, subdivide `R` with a "coarse" rectangular mesh, say having $m_1 \times m_2$ sub-rectangles. For each "grid line", plot the parametric curve obtained by evaluating `f` along the line. The result is the (wiremesh) plot of `f` over `R`.

A detail was omitted in the preceding paragraph: the number of points plotted in each parametric curve. Many graphing programs use as many points as dictated by the coarse mesh, so that the surface is made of quadrilaterals (Figure 3.2, left). In `ePiX`, a domain possesses a "fine" mesh, say that divides `R` into $n_1 \times n_2$ subrectangles (Figure 3.2, right). The individual parametric curves comprising the surface are drawn using the fine subdivisions. A plot can conform to the "true" surface without requiring many quadrilaterals.

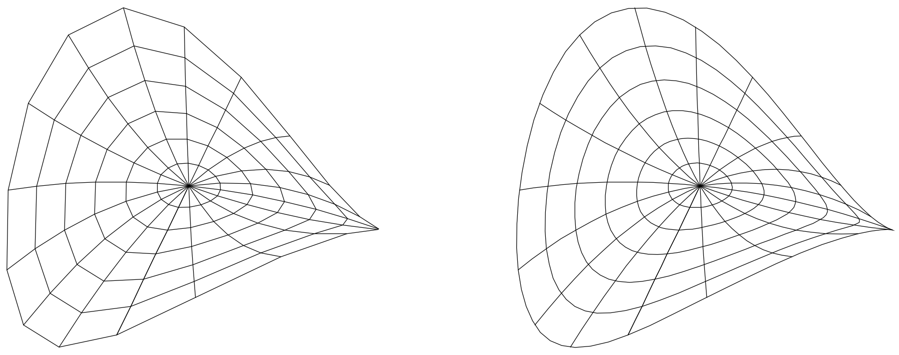


Figure 3.2: A surface with coarse (6×18) and fine (12×60) meshes.

A `domain` is defined by giving a pair of opposite corners, the coarse mesh, and the fine mesh. A domain's meshes are independent; the fine mesh need not be a "multiple" of the coarse mesh. Indeed, the "fine" mesh may be *more coarse* than the "coarse" mesh. Usually, however, the fine mesh is a "small multiple" of the coarse mesh.

A domain can be resized in any coordinate direction for which the thickness is positive, using object-oriented syntax. For the domain `R` above,

```
domain R_new = R.resize2(0.5,0.75);
```

defines a new domain having opposite corners $(0, 0.5)$ and $(1, 0.75)$. To the extent possible, resizing attempts to preserve absolute grid sizes. In this example, the new domain has 6×4 coarse mesh ($18/4 = 4.5 \to 4$) and 12×15 fine mesh. It's best to choose meshes so that resizing does not cause integer truncation.

A domain can be sliced by setting one variable to a constant; the result is a domain whose dimension is one smaller than the original:

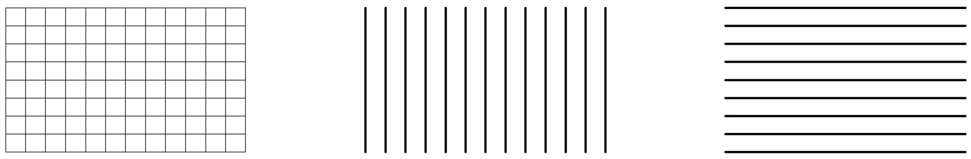
```
domain R1 = R.slice1(0.3);
```

creates the 1-dimensional domain with endpoints $(0.3, 0)$ and $(0.3, 1)$, and having coarse mesh 18 and fine mesh 60. Resizing and slicing commands may be used directly in a `plot` command:

```
plot(f, R.slice1(0.3));
```

draws the parametric curve obtained by slicing `R` at $x_1 = 0.3$, then applying `f`.

For plotting families of maps, a `domain` has "`slices`" operators that return the list of `domain` slices obtained by setting one variable to components of the coarse mesh:



When plotting a function of one variable, constructing a `domain` is possible but tedious. The command

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

plots `f` over the interval `[t_min, t_max]` using `n` subintervals (`n+1` points).

The ability to slice or resize makes `domain`s more useful when plotting functions of two or three variables. The commands below typify plot commands; f is a function of one variable, and F is a function of 2 or 3 variables. If the function is real-valued, the graph is drawn; otherwise the image is drawn.

```
// two ways to plot F on [a,b] x [c,d]
plot(F,  P(a,c), P(b,d), mesh(N1,N2), mesh(n1,n2));

domain R(P(a,c), P(b,d), mesh(N1,N2), mesh(n1,n2));
plot(F, R);

plot(F, R.slice1(a));      // plot F on part of the boundary
plot(F, R.slices1());      // plot F over all slices x1=const
```

A function of 3 variables can be plotted over a 1- or 2-dimensional domain, while a function of 2 variables cannot be plotted over a 3-dimensional domain.

Utility Functions

In this section, `f` and `g` are `double`-valued functions of one variable. `ePiX` defines numerical functions that return the maximum or minimum value on an interval, approximate the location of roots, and perform calculations with derivatives and definite integrals.

```
  sup(f, a, b);     // max/min of f on [a,b]
  inf(f, a, b);
  newton(f, g, x0); // find approximate crossing point
```

Newton's method returns the crossing point of the given functions, starting from the specified seed. If a critical point is hit or 20 iterations pass, a warning is issued and the current result (probably incorrect) is returned. The second function g defaults to the zero function if omitted.

The classes `D` and `I` are used to calculate values of derivatives and integrals, and to plot these functions.

```
  D df=D(f);   // data structure representing the derivative
  df.eval(t);  // return f'(t)
  df.left(t);  // deriv from left at t:  (f(t)-f(t-dt))/dt
  df.right(t); // deriv from right at t: (f(t+dt)-f(t))/dt

  I prim=I(f,a); // representation of the integral from a
  prim.eval(b);  // numerical integral of f over [a,b]
  double val=I(f).eval(1);   // \int_0^1 f
```

The lower limit on an integral is 0 by default.

Tangent lines and envelopes (families of tangent lines) are drawn with

```
  tan_line(f, t);                // f real- or vector-valued
  envelope(f, t_min, t_max, n);  // family of tangent lines
  tan_field(f1, f2, t_min, t_max, n); // field of tangents
```

The sample files `conic.xp` and `lissajous.xp` illustrate these features.

Calculus Plotting

`ePiX` can plot the derivative or definite integral of real-valued functions, solve ODEs in two or three variables, and graph slope- or vector fields. Let `f` be a real-valued function of one variable, `F` a `P`-valued function of two or three variables.

```
  plot(D(f), a, b, n);  // plot f' over [a,b]
  plot(I(f, x0), a, b, n);

  ode_plot(F, p_0, t_min, t_max, n);
  flow(F, p_0, t_max, n);
```

The second command graphs the definite integral $x \mapsto \int_{x_0}^{x} f(t)\,dt$ over $[a,b]$. As above, x_0 defaults to 0.

The third command plots the solution curve of the initial-value problem $\dot{x} = F(x)$, $x(0) = p_0$, over the specified time interval. If $t_{\min}$ is omitted, its value is 0, so the curve starts at p_0. With manual calculation to rotate a planar field a quarter turn, `ode_plot` can be used to draw level curves (isobars) of a function; see the sample file `dipole.xp`. The fourth command returns the result of starting at p_0 and flowing by F for time $t_{\max}$, using Euler's method with n timesteps. This is useful for placing markers or arrowheads precisely along a flow line. A vector field itself may be drawn in three ways:

```
vector_field(F, p, q, n1, n2); // proportional length
dart_field  (F, p, q, n1, n2); // constant length
slope_field (F, p, q, n1, n2); // directionless, const length
```

The field is sampled at the grid points of the coordinate rectangle whose corners are specified.

Recursive Fractal Curves

Consider a path made up of equal-length segments that can point at any angle of the form $2\pi k/n$ radians, for $0 \le k < n$, like spokes on a wheel. A 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 approximation starts with such a "seed" then recursively (up to a specified depth) replaces each segment with a scaled and rotated copy of the seed. The seed above generates the standard von Koch snowflake fractal. In code:

```
const int seed[] = {6, 4, 0, 1, -1, 0};
fractal(P(a,b), P(c,d), depth, seed);
```

The first entry of `seed[]` (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 likely to exceed the capabilities of LaTeX and/or PostScript.

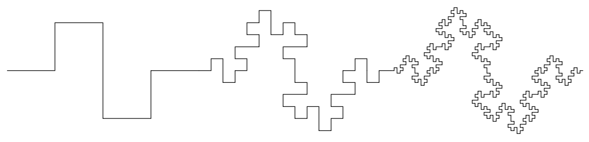


Figure 3.3: Successive iterations of {`4,8,0,1,0,3,3,0,1,0`}

3.8 Non-Euclidean Geometry

Hyperbolic line segments are specified by their endpoints in the upper half space or ball (Poincaré) models. In each case there is no output if either endpoint lies outside the model.

```
hyperbolic_line(p, q);
disk_line(p, q);
```

For compatibility with 2-dimensional hyperbolic space, the half-space model is the set $\{(x_1, x_2, x_3) \mid x_2 > 0\}$.

A `frame` determines geographical coordinates on a `sphere`: the first element points toward longitude 0 on the equator, the third element points to the north pole. A latitude line depends on a `sphere`, a `frame`, the numerical latitude, and a range of longitudes. A longitude line is described similarly.

```
latitude(lat, long_min,long_max, sphere S, frame coords);
longitude(lngtd, lat_min, lat_max, sphere S, frame coords);
```

By default, `coords` is the standard `frame` and `S` is the unit `sphere`. These commands draw only the portion of the curve that is visible from the current `viewpoint`. The function `back_latitude` draws the invisible portion of a latitude line.

Parametrized paths on the unit sphere can be specified either by radial projection of a space curve, or by stereographic projection of a plane curve:

```
  plot_R(phi, t_min, t_max, n);    // radial
  plot_N(f1, f2, t_min, t_max, n); // from north pole
  plot_S(f1, f2, t_min, t_max, n); // from south pole
```

Attempts to perform radial projection on a path through the origin will generate division-by-zero errors. Stereographic projection maps the equatorial plane $\{x_3 = 0\}$ to the unit sphere by projection from the corresponding pole: $N = (0, 0, 1)$, $S = (0, 0, -1)$.

Each spherical plot command accepts a prefix `front` or `back` that prints only the portion of the path visible or invisible (respectively) from the current `viewpoint`.

Because of the way `ePiX` layers output, it is generally best to put hidden portions of the input before visible portions, with line width and/or style that suggests hidden lines.

3.9 Animation

`ePiX` is ideally suited to the creation of mathematically accurate animations: If a figure depends suitably upon a "time" parameter, then a loop can be used to draw the entire figure for multiple time values, yielding successive "snapshots" of the figure as time progresses. The shell script `flix` automates the process of compiling a suitable input file into a collection of `png`s and assembling these frames into a `mng` animation. ImageMagick is the image-handling engine.

A `flix` file is an `epix` file with two restrictions:

- The `double` variable `tix` is used as "clock".
- `main` accepts two commandline arguments and sets `tix` accordingly.

Jay Belanger's `emacs` mode recognizes the file extension `.flx` and inserts template code if an empty buffer is opened. Creation of `flix` files is as easy as creation of `epix` files. The directory `samples/extras` contains a handful of `flix` files that may be consulted for ideas.

A "typical" `.flx` file may take 30 seconds to a couple of minutes to compile. To present the impression that work is being done, `flix` prints a progress bar, counting the number of `eps` files that have been created. There will be a delay of a few seconds after the last frame is produced, during which ImageMagick's `convert` utility creates `png` files from `eps` files, then assembles the movie.

To facilitate debugging, `elaps` can be run on a `flix` file. `elaps` runs in a fraction of the time, and if `elaps` can't produce a viewable image, `flix` will surely fail.

By default, `flix` creates movies with 24 frames, in which `tix` runs from 0 to 1, and animates at 0.08 sec/frame. Command-line options change these and other parameters; please use `flix`'s built-in help for details.

3.10 Troubleshooting

Seven files comprise `ePiX`: a header (`epix.h`), compiled library (`libepix.a`), and five shell scripts (`epix`, `elaps`, `laps`, `flix`, and `keywords`). Four of these files are generated at compile time, using variables found in the `Makefile`. In addition to the executable portions of the program, there are a few dozen sample files, configuration files, `README`s, and miscellaneous documentation. These components are placed into standard locations where they can find each other.

Installation Problems

Public installation (in `/usr/local`) is the default. Unpack the tar file, then `cd` to the source directory (`epix-x.y.z_complete`). If you are installing in `/usr/local`, your `C++` compiler is `g++`, and `bash` is in `/bin/bash`, then do

```
make
make contrib (optional)
make test    (optional)
make install
```

(the last as `root`, if necessary). If `bash` is not in `/bin/bash`, edit the script `newbash.sh` and run it in the source directory. If the install directory or compiler are not as above, edit the `Makefile` accordingly, then build. If a failed partial compile creates the header file `epix.h`, you *must* do `make clean` before re-compiling.

If you have installed in `$HOME`, please see the `POST-INSTALL` file for additional instructions.

The directory in which you install `ePiX` is denoted `$INSTALL`. A successful `make install` creates the following files:

```
  $INSTALL/bin/{epix,elaps,laps,flix,keywords}
  $INSTALL/include/epix.h
  $INSTALL/lib/libepix.a
  $INSTALL/man/man1/epix.1
```

Sample files, notes and README files, configuration snippets, and (if the `complete` package is installed) this tutorial are installed in subdirectories of `$INSTALL/share/epix`.

Known Issues

If you install `ePiX` on MacOS X with the December, 2002 Apple Developer Tools, you must manually run `ranlib` on the compiled library:

```
sudo ranlib /usr/local/lib/libepix.a
```

Buggy versions of the conversion script `ps2epsi` have been distributed with recent versions of RedHat and Mandrake. The author has seen two *different* one-byte errors that cause `elaps` to fail with a `sed` error. If `elaps` generates a `sed` error, you have a buggy `ps2epsi`. Please google for the specific error message or, if you're good with regular expressions, just fix the script yourself. (In both prior instances, the problem was an unescaped special character.)

Runtime Errors

Typos frequently cause errors when `epix` is run; the compiler generally issues a helpful message, naming the troublesome line of the input file and the type of error.

Two common sources of error in a syntactically correct input file are camera mis-placement and non-generic intersections. If the camera is too close to objects in the scene, the lens may try to divide by zero (or by a small epsilon), resulting in `nan` errors (or very large coordinates) in the output file. In this situation, `epix` will succeed, but `elaps` will hang. By contrast, an error message of the form

```
/usr/local/bin/epix: line 275: 27333 Aborted ...
```

signifies an uncaught exception, hence a non-generic intersection.

Though the shell scripts are extensively tested, they are a possible source of runtime problems. (For example, `elaps` tries to pass options to the compiler while reserving other options for its own use, all while trying to deal intelligently with malformed commands.) If a script does something unexpected, it's probably a Feature, not a Bug. In any case, please notify the author of anomalous behavior.

Hanging scripts By default, the conversion scripts run silently. Because LaTeX does not use separate channels to return output and error messages, `laps` can hang if there is a problem with the LaTeX file. If this happens, try typing "`s`" to put LaTeX into scroll mode. If this fails to return the prompt after a few seconds, type `<ctrl>-C` to kill `laps`, and re-run in verbose mode:

```
laps -vv <filename>
```

This will cause LaTeX's error messages to be printed normally.

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.

Permission denied This is unlikely, but conceivable. For each component of the program, do a long listing, e.g. (with the appropriate install directory)

```
  ls -l /usr/local/bin/epix
```

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,elaps,laps,flix,keywords}
chmod 0644  include/epix.h  lib/libepix.a  man/man1/epix.1
```

If you still cannot get `ePiX` to run, please send email to the author. Generally, it is helpful to specify the operating system (e.g., RedHat 8.2, or Debian Potato on a G4 Powerbook), the version of the `C` compiler (e.g. `gcc-3.2`), the version of `ePiX`, and where you downloaded the sources (CTAN, the project home page, etc.). Specific commands you typed and error messages may also be helpful. If you don't know what an error message means, feel free to paste or attach it to your email as plain text. The author does not have access to an enormous variety of systems, and sometimes reported errors never get explained, but to date no error has failed to be worked around.

LaTeX Errors

A few things 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 has been addressed; please send the author a bug report if you encounter this behavior in `ePiX` code. 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 in LaTeX 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.

Chapter 4

Advanced Topics

This chapter covers *ad hoc* tricks and open-ended techniques that require relatively more programming sophistication. You will almost surely need an external `C++` reference if you do not speak the language.

4.1 Hidden Object Removal

`ePiX` writes the output file in the same order that objects appear in the input. The order is significant because PostScript builds a figure in layers: Objects are drawn over objects that come earlier in the file. Shaded polygons can be used to obtain surprisingly effective hidden object removal in surface meshes. The techniques are still provisional, however; this section describes a method that works for the author, though is not sufficiently refined for inclusion in the source code.

The basic idea is to create a shaded quadrilateral class that knows its distance to the camera. To draw parametrized surfaces, quadrilaterals are written in any convenient order to a `C++` vector, then sorted by distance to the camera and printed to the output file in decreasing order of distance. The gray density of a mesh element depends on the cosine of the angle between the normal vector and the vector from the camera to the element. The method works acceptably well in practice: Several surfaces can be drawn, and the effect is fairly realistic. If the surface meshes are fine enough, intersections are accurate and not jagged. However, the output file tends to be enormous, and is not effectively human-readable.

The quadrilateral class might look like this:

```
class mesh_quad
{
private:
  P pt1, pt2, pt3, pt4;
  double distance;

public:
  mesh_quad()
  {
    pt1=pt2=pt3=pt4=P();
    distance=camera.get_range();
  }

  mesh_quad(P f(double u, double v), double u0, double v0)
  {
    pt1=f(u0+EPS,v0+EPS);       // EPS = "shrinkage"
    pt2=f(u0+du-EPS,v0+EPS);
    pt3=f(u0+du-EPS,v0+dv-EPS);
    pt4=f(u0+EPS,v0+dv-EPS);

    P center = 0.25*(pt1 + pt2 + pt3 + pt4);
    distance = norm(center-camera.get_viewpt());
  }

  double how_far() const { return distance; }

  void draw()
  {
    P normal = (pt2 - pt1)*(pt4 - pt1);
    normal *= 1/norm(normal);

    double dens  = 0.75*(1-pow(normal|LIGHT, 2)/(LIGHT|LIGHT));
    gray(dens);
    quad(pt1, pt2, pt3, pt4);
  }
};
```

To utilize C++'s sorting algorithm, a class is defined to measure the distance to a mesh element:

```
class by_distance {
public:
  bool operator() (const mesh_quad& arg1, const mesh_quad& arg2)
  { return arg1.how_far() > arg2.how_far(); }
};
```

Assuming `f1` and `f2` are parametric surface maps, the class above can be used as follows in the body of the figure:

```
std::vector<mesh_quad> mesh(0); // list of mesh_quads

for (int i=0; i<N1; ++i)
  for (int j=0; j<N2; ++j)
    {
      mesh.push_back(mesh_quad(f1, -1+du*i, -1+dv*j));
      mesh.push_back(mesh_quad(f2, -1+du*i, -1+dv*j));
    }

sort(mesh.begin(), mesh.end(), by_distance());

for (unsigned int i=0; i<mesh.size(); ++i)
  mesh.at(i).draw();
```

For a complete file, please see `samples/extras/tori.xp`.

4.2 Extensions

Thanks to a suggestion of Andrew Sterian, `ePiX` is extensible. The preferred method is to create external modules for use at run time, analogous to a LaTeX macro/style file. It is also possible to modify the `ePiX` source code itself before compiling; this is inflexible (and requires administrator privileges), but can be useful for changing system-wide defaults. User extensions span a spectrum, from header files that require only basic knowledge of `C++` to separately compiled libraries that substantially extend the capabilities of `ePiX`.

Header Files

A `C++` header file conventionally has suffix `.h`, as in `myheader.h`. To use this custom header, put a line `#include "myheader.h"` in your source file.

User definitions can be easily and robustly implemented with “inline functions”. Inline functions are superficially similar to macros, but are far more safe and featureful (since they are handled by the compiler rather than by the pre-processor). Examples are

```
inline void Bold(void) { pen(1.5); }
inline void purple(void) { rgb(0.5, 0, 0.7); }
```

```
  inline void draw_square(double s) { rect(P(-s,-s),P(s,s)); }
  inline double cube(double x) { return x*x*x; }
```

The keyword `void` signifies a function that does not return a value, or (when used as a parameter) a function that does not accept arguments. Inline function definitions are syntactically exactly like ordinary function definitions, but *must* occur in a header file or in the source file where they are used. The examples above might be used in an input file as follows:

```
  Bold();
  draw_square(cube(1.25));
```

Compiling

The next few sections outline the creation of a "static library" on GNU/Linux, and explain how to incorporate custom features at runtime. For more details, the source code, `Makefile`, and your system's documentation should provide a good start.

A small library is usually written as a *header* file, which contains function declarations (also called "prototypes"), and a *source* file, which contains the actual code. Conventionally (under *nix), these files have extension `.h` and `.cc` respectively. Header and source files may "include" other header files, to incorporate additional functionality.

```
/* my_code.h */
#ifndef MY_CODE
#define MY_CODE
#include <cmath>   // standard library math header
#include "epix.h"  // ePiX header
using ePiX::P;

namespace Mine {   // to avoid name conflicts
  // functions for special relativity
  double lorentz_norm(P);
  bool   spacelike(P);
} // end of namespace
#endif // MY_CODE
```

This file exhibits two "safety features". The three `MY_CODE` lines prevent the file from being included multiple times. In a file of this size, inclusion protection is overkill, but as your code base grows and the number of header files increases, this protection is essential. Second, the header introduces a "Mine" namespace. Inside this namespace, two functions are declared as prototypes, giving the function's return type, name, and argument type(s). A header file should be commented fairly liberally, so that a year or two from now you'll be able to decipher the file's contents. For a longer file, version and contact information, an overall comment describing the file's features, and license information are appropriate.

Next, the corresponding source file; definitions are also placed into the namespace, and must match their prototypes from the header file exactly.

```
/* my_code.cc */
#include "my_code.h"
using namespace ePiX;

namespace Mine {
  double lorentz_norm(P arg)
  {
    double x=arg.x1(), y=arg.x2(), z=arg.x3(); // extract coords
    return -x*x + y*y + z*z;
  }
  bool spacelike(P arg)
  {
    return (lorentz_norm(arg) > 0); // true if inequality is
  }
} // end of namespace
```

Copies of these files are included with the source code so you can experiment with them. Next, the source file must be "compiled", "archived", and "indexed". In the commands below, the percent sign is the prompt.

```
% g++ -c my_code.cc
% ar -ru libcustom.a my_code.o
% ranlib libcustom.a
```

Please see your system documentation for details on command options and what each step does. For linking (below), the name of the library file must begin "lib" and have the extension `.a`. Once these steps are successfully completed, put the library `libcustom.a` and header file `my_code.h` in your project directory. You're ready to use the code in an `ePiX` figure.

Runtime Linking

The script `epix` allows input files to be linked with external libraries at run time, when the input file is compiled into a temporary executable.

`epix` recognizes command line options and passes them verbatim to the compiler. The most commonly used options are those of the form

```
-I<include>     -L<libdir>     -l<lib>
```

There must be no space between the option flags `-I`, `-L`, and `-l` and their arguments. For example, to link `figure.xp` against `mylibs/libcustom.a`, run the command

```
epix -Lmylibs -lcustom figure
```

The options `-I. -L.` tell the compiler to look in the current directory for header and library files. Compiler options may appear in any order, but must come before the name of the input file; options that come after the input file are silently discarded.

Compiler options may be placed in the configuration file `$HOME/.epixrc`, with syntax as above. A line in the config file that contains a pound sign (`#`) is a comment, no matter where in the line the `#` appears. If any non-comment line fails to start with a dash, the rest of the file is silently discarded. Command-line options are read before the config file.

Appendix A

Software Freedom

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, users should be granted the four freedoms laid out in the GNU General Public License:

`Free0`: To run a program for any purpose
`Free1`: To study how the program works, and adapt it to your needs
`Free2`: To redistribute copies of the program
`Free3`: To improve the program, and release improvements to the public

Just as theorems are not restrictively licensed, I believe that software we use in our academic work should be licensed in a way that encourages openness and sharing. 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, Richard Stallman, and the many people who have contributed to the authorship of LaTeX and its 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.

Appendix B

Acknowledgements

`ePiX` is built on the work of many people (unfortunately, most of whom I am unaware). The following people have contributed, sometimes unknowingly but always generously:

Infrastructure Donald E. Knuth, Conrad Kwok, Leslie Lamport, Tim Morgan, Piet van Oostrum, Sunil Podar, Richard Stallman

Enhancements Jay Belanger, Robin Blume-Kohout, Guido Gonzato, Svend Daugård Pedersen, Andrew Sterian

Debugging, advice, and other assistance Jay Belanger, Felipe Paulo Guazzi Bergo, Robin Blume-Kohout, Patrick Cousot, Stephen Gibson, Dov Grobgeld, Bob Grover, Jim Hefferon, Jacques L'helgoual, Yvon Henel, Hartmut Henkel, Herng-Jeng Jou, Walter Kehowski, Ross Moore, Thorsten Riess, Neel Smith, Michael Somos, Andrew Sterian, Ryszard Tanas, Kai Trukenmueller, Torbjorn Vik, Wenguang Wang, Gabe Weaver, Mariusz Wodzicki

Bibliography

[1] B. Kernighan and D. Ritchie, *The C Programming Language*, Second Ed., Prentice-Hall Software Series, 1988

[2] S. Loosemore, R. M. Stallman, et. al., *The GNU C Library Reference Manual*, GNU Press, 2004.

[3] K. Reckdahl, *Using Imported Graphics in LaTeX2e*, Version 2.0, whitepaper, Dec. 15, 1997

[4] B. Stroustrup, *The C++ Programming Language*, Special Ed., Addison-Wesley, 1997

[5] T. van Zandt, *PSTricks: PostScript Macros for Generic TeX*, Version 0.93a, whitepaper, Mar. 12, 1993

Index