`ePiX` Tutorial and Reference Manual

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

Version 1.2.17, July, 2017

Contents

1 Introduction **7**
- 1.1 Software Dependencies . . . 8
  - 1.1.1 Setting up an Environment Under Windows . . . 9
- 1.2 Installation . . . 9
  - 1.2.1 Development . . . 10

2 Getting Started **11**
- 2.1 Running `ePiX` . . . 11
- 2.2 The Drawing Model . . . 13
- 2.3 Tutorial . . . 13
- 2.4 `C++` Basics . . . 21
  - 2.4.1 File Format . . . 21
  - 2.4.2 Variables and Functions . . . 21
  - 2.4.3 Comments . . . 22
  - 2.4.4 Program Execution . . . 22
  - 2.4.5 Strings and Raw Output . . . 22
  - 2.4.6 Conditionals and Loops . . . 23
- 2.5 Animation . . . 23
- 2.6 Layout Tricks . . . 25
  - 2.6.1 Stereograms . . . 25
  - 2.6.2 Inset Images . . . 25

3 Reference Manual **27**
- 3.1 File Structure . . . 27
- 3.2 Picture Size and Aspect Ratio . . . 29
- 3.3 Color . . . 30
  - 3.3.1 Constructors . . . 30
  - 3.3.2 Color Operations . . . 31
- 3.4 Scene Attributes . . . 31
  - 3.4.1 Angular Mode . . . 31
  - 3.4.2 The Camera . . . 32
  - 3.4.3 Clipping . . . 34

3.4.4 Screens and Page Layout . . . 34
3.5 Drawing Attributes . . . 38
3.5.1 Filled Regions . . . 38
3.5.2 Paths . . . 38
3.5.3 Text Objects . . . 40
3.5.4 Color Declarations . . . 42
3.6 Creating and Drawing Objects . . . 43
3.6.1 Geometric Data Structures . . . 43
3.6.2 Path-Like Elements . . . 46
3.6.3 Coordinate Axes and Labels . . . 48
3.6.4 The Path Class . . . 51
3.6.5 Function Plotting . . . 52
3.6.6 Calculus Plotting . . . 57
3.6.7 Non-Euclidean Geometry . . . 59
3.6.8 Data Plotting . . . 61
3.6.9 Legends . . . 65
3.7 More About C++ . . . 66
3.7.1 Names and Types . . . 66
3.7.2 Functions . . . 67
3.7.3 Mathematical Functions . . . 68
3.7.4 Basics of Classes . . . 69
3.7.5 References and Function Arguments . . . 69
3.7.6 Overloading . . . 70
3.7.7 Scope . . . 70
3.7.8 Headers and Pre-Processing . . . 71
3.7.9 Comparison with LaTeX Syntax . . . 71
3.8 Attribute Quick Reference . . . 72

4 Advanced Topics 75
4.1 Hidden Object Removal . . . 75
4.2 Extensions . . . 76
4.2.1 Header Files . . . 76
4.2.2 Compiling . . . 76
4.2.3 Runtime Linking . . . 78
4.2.4 Using Multiple Versions . . . 78
4.3 Programmer's Guide . . . 79
4.3.1 External Packages . . . 79
4.3.2 User Interface . . . 80
4.3.3 Implementation Classes . . . 81

A Software Freedom 85

B Acknowledgments **89**

Chapter 1

Introduction

ePiX, a collection of batch utilities, creates mathematically accurate figures, plots, and animations containing LaTeX typography. The input syntax is easy to learn, and the user interface resembles that of LaTeX itself: You prepare a scene description in a text editor, then "compile" the input file into a picture. LaTeX- and web-compatible output types include a LaTeX picture-like environment written with PSTricks, tikz, or eepic macros; vector images (eps, ps, and pdf); and bitmapped images and movies (png, mng, and gif).

ePiX's strengths include:

- Quality of output: ePiX creates 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: Objects are described by 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 command-line switches or minor changes to the input file.

- Power and extendibility: 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 a 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.

`ePiX` facilitates logical structuring of mathematical figures, as opposed to visual structuring (or WYSIWYG), analogous to the relationship between LaTeX and a word processor. Stylistic defaults streamline the creation of simple figures, but there are few internal restrictions on the contents or appearance of a figure; aesthetic and practical decisions are left to you.

If you are a:

- Potential user, you may wish to skip immediately to "Software Dependencies" before investing additional time.
- New user, proceed from here until you have enough understanding to run 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.

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 probably have (and can surely install) all the external software needed to use `ePiX`. On Mac OS X, you will need the Apple developer tools and an X server (such as XQuartz), and the free `fink` package manager to build a GNU environment. For Windows, you'll need to install Cygwin and several packages. Detailed instructions are given below.

"Under the hood", an input file is successively converted to a LaTeX picture; `dvi`; PostScript, `pdf` or `eps`; and if desired, to a bitmapped image or movie. Four shell scripts—`epix`, `laps`, `elaps`, and `flix`—automate the various file format conversions.

`ePiX` consists of a `C++` library, header, and shell scripts, and requires GNU `bash` and a compiler *for normal use*. For complete functionality, you need `g++` (Version 3.2 or later), `bash`, a text editor (`ePiX` works particularly well with `emacs`), a LaTeX distribution, `Ghostscript`, `gv` (or your favorite PS/PDF previewer), and `GraphicsMagick`. GNU `grep` and `sed` are good to have. You may need additional "developer packages" (`binutils`, `make`) in order to build `ePiX`. The more up to date your software is, the better your experience is likely to be, but bleeding edge versions are not necessary, or even always desirable.

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 GNU environment.

Jay Belanger's emacs mode allows you to write, compile, and view ePiX figures without leaving emacs. If you use another editor, you'll want to create template source files so you don't have to type boilerplate code each time you write a new figure.

1.1.1 Setting up an Environment Under Windows

Cygwin can be used to run ePiX under Windows. Download setup.exe from www.cygwin.com, then install the packages you need. The following are recommended, and sufficient for the actions described in this manual.

```
(Archive)    bzip2, tar
(Devel)      binutils, coreutils, gcc, gcc-g++, make, sed
(Editors)    emacs, emacs-X11, vim
(Graphics)   GraphicsMagick, ghostscript-base, ghostscript-x11, gv
(Publishing) tetex (all)
(Shells)     bash, bash-completion
(X11)        X-start-menu-icons, X-startup-scripts,
             XFree86-lib-compat, xorg-x11-fscl, xorg-x11-fsrv
```

1.2 Installation

ePiX is distributed over the World-Wide Web as source code. Packages may be found at mathcs.holycross.edu/~ahwang/epix/ePiX.html The latest stable release is also on the CTAN mirrors, in the graphics directory. (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.tar.gz
tar -jxvf epix-x.y.z.tar.bz2
```

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

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

cd to the source directory, epix-x.y.z. The INSTALL file contains detailed installation instructions. If you're impatient, the short of it is ./configure [--options]; make; make install. Run ./configure --help for a list of options.

By default, `ePiX` installs in subdirectories of `/usr/local`; if you want to install elsewhere, supply `./configure` with the appropriate `--prefix`. You may also want to consult `POST-INSTALL` for information on setting your `PATH` variable so your shell can find `ePiX`. The manual and sample files are in `/usr/local/share/doc/epix`.

1.2.1 Development

There are two mailing lists, one for user questions, one for development discussion. Please visit `savannah.nongnu.org/mail/?group=epix` to subscribe.

Chapter 2

Getting Started

This chapter describes the basics of creating figures in `ePiX` for readers familiar with LaTeX but completely new to `C++`. No detailed knowledge of `C++` is needed to use `ePiX`, only a bit of grammar that is easily absorbed by example.

Section 2.1 describes the commands (shell scripts) comprising `ePiX`, and explains how to set up a graphical environment using standard *nix programs. Section 2.2 briefly describes figure creation. Section 2.3 presents a few files side-by-side with their output, and should be read at a computer so you can run the exercises.

2.1 Running `ePiX`

An "input file" is a human-written figure specification containing `ePiX` commands; an "output file" is either a LaTeX picture-like environment or a stand-alone vector graphic. Conversion ("compiling" a figure) is accomplished with four shell scripts, `laps`, `epix`, `elaps`, and `flix`. Each script has a preferred extension for its input files, and is invoked with a command of the form

```
<script> [options] <input file(s)>
```

Often, no options are necessary. `<script> --help` describes `<script>`'s options. Figure 2.3 (page 26) diagrams the shell scripts and the file types they process.

By default, output file names are constructed by replacing the input extension with the (preferred) output extension. For brevity, extensions may be omitted. If the script has doubts about your intent, it proceeds with default behavior and prints a warning message.

The author is a great fan of `TAB` completion, under which a shell, based on what has been typed so far, automatically fills in a command when the `TAB` key is pressed. `ePiX` comes with code snippets that complete intelligently when the first part of a command is one of the shell scripts. For example, if command completion is active, typing `epix TAB` prints only names of `epix` input files. To use this feature, you must

install Ian MacDonald's `bash completion` package. The `INSTALL` and `POST_INSTALL` files contain details.

Shell Scripts

laps

`laps` performs LaTeX to PostScript/PDF conversion, and is independent of the rest of `ePiX`. By default, `laps` invokes LaTeX and `dvips`. The option `--pdf` creates a PDF file by post-processing the PostScript with `ps2pdf`. Other TeX-family processors (`pslatex`, `pdftex`, etc.) may be used instead of LaTeX by invoking `laps` with an appropriate option.

epix

`epix` compiles an input file into a LaTeX picture. The input file should have extension `xp` (for eXtended Picture). Jay Belanger's `emacs` mode is Highly Recommended; the installation procedure can be found in the `POST_INSTALL` file. When this mode is active, `emacs` automatically inserts a preamble template when a new `xp` file is created. In addition, you can peruse `ePiX`'s info file, or format, compile, and preview files, all from within `emacs`.

Output files from the script `epix` have extension `eepic`, after Conrad Kwok's enhancements to the LaTeX picture environment. In Version 1.2, however, an output file may contain `PSTricks` or `tikz` macros instead of `eepic` macros. A command such as `\input{myfile.eepic}` incorporates an output file into a LaTeX document. The preamble must contain `usepackage` lines appropriate to the output format.

Format	Required Package(s)
`eepic`	`epic,eepic,xcolor,rotating`
`pst`	`pstricks,rotating`
`tikz`	`tikz,rotating`

elaps

`elaps` creates stand-alone vector images (`eps` or `pdf`) from `epix` input files or `eepic` figures, even those not produced with `epix`. `elaps` automatically loads the LaTeX packages needed for features described in this manual. Additional LaTeX packages and `dvips` options may be specified on the command line. An `elaps` output file is placed in a LaTeX document with an `\includegraphics` command; the preamble must contain `\usepackage{graphicx}` [sic].

flix

`flix` creates bitmapped images and movies, Section 2.5. Input files should have extension `flx`, and must contain a valid `epix` header as well as additional code. Jay

Belanger's `emacs` mode facilitates creation of `flix` files.

2.2 The Drawing Model

To draw a figure manually, you select a sheet of paper of appropriate size and add paths, markers, and labels. These scene elements have attributes (line color, line width, fill color, font size, etc.) affecting their appearance.

`ePiX`'s drawing model behaves similarly. A `picture` command sets the *canvas* (or logical drawing area, a Cartesian rectangle) and the true size of the final figure. A `begin` command initializes the "virtual paper" for drawing. Subsequent *drawing commands* add objects to the scene: lines, curves, function plots, labels, and the like. The printed appearance of scene elements is determined by the current "attribute state" and controlled by *style declaration* which remain in effect until superseded. Finally, an `end` command closes the figure and prints it to the output file.

In this manual, command descriptions follow a few conventions. Optional arguments are enclosed in square brackets. A `len` argument is either a number (representing a length in `pt`) or a double-quoted string containing a number and a two-letter LaTeX length unit, such as `"1.5pt"` or `"6cm"`. A `color` argument is a named primary (`Red()`, `Cyan()`, `White()`, etc.), or a `Color` specified by densities (`RGB(r,g,b)`, `CMY(c,m,y)`, etc.)

- Fill color: `fill(color)`, `nofill()`.
- Line color and width: `plain([color])`, `bold([color])`.
- Text size: `font_size([size])`, returns to `normalsize` if no argument is given.

A complete list of style commands is found in Section 3.8, starting on page 72.

To handle three-dimensional scenes, a `camera` performs point projection from a selected spatial location to the canvas. As a user, you'll need to control relatively few of the `camera`'s parameters. Keep in mind, however, that elements are added to a scene in the same order their commands appear in the output file, and that later elements generally cover earlier ones. Some three-dimensional scenes require manual ordering of the input file; such ordering is dependent on the `camera`'s location.

By default, `ePiX` draws thin, solid, black lines, performs no filling of regions, prints text in a 12 pt Roman font, and looks down the z axis from a large distance, giving orthogonal projection on the (x, y) plane. When the `camera` is moved, the z axis points vertically up on the page.

2.3 Tutorial

This section presents sample input files side-by-side with their output so you can compare what you write with what you'll see on the screen or page. `ePiX` provides

standard drawing capabilities, but like all software has its own idioms and personality. The basic syntax, which comes from C/C++, should be mostly self-explanatory. One-line comments begin with the string "//".

To use the sample files interactively, you'll need working software: bash, emacs, ePiX, g++, gv, and a running X server. To complete your "GUI", start gv and select "Watch file" from the "State" menu. The loaded file will update automatically when it changes.

Create a "scratch" directory, cd into it, and run the command

```
tar -zxf /usr/local/share/doc/epix/sample_src.tar.gz
```

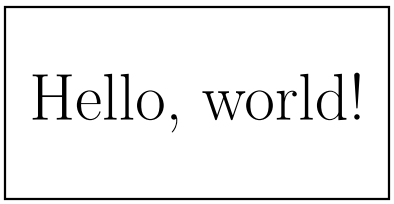
(Change the path as appropriate for your installation.) This unpacks copies of the sample files into your scratch directory, including all the files mentioned in this manual. The README file serves as a table of contents.

Open a sample file in emacs, compile it from the drop-down menu (or with the keyboard command), then open the EPS file in gv. Now you're ready to follow the tutorial interactively. A few suggested exercises are included with each file. Naturally, as you study more files, you'll be able to make more interesting changes on your own.

Basic Drawing

The first sample, hello.xp, contains code needed to specify the figure's size, followed by the classic greeting. The border command draws a box around the figure in the specified color and width, and serves here merely to delimit the output from the surrounding page.

```
/* -*-ePiX-*- */
#include "epix.h"      // These lines are analogous
using namespace ePiX;  // to a usepackage command.

int main()
{
  picture(P(-1,-1), P(1,1), "2 x 1in"); // corners, true size

  begin(); // ---- Figure body starts here ----

  border(Black(), "1pt"); // color, line width

  font_size("Huge");  // May be any font size, e.g. "scriptsize"
  label(P(0,0), "Hello, world!");

  end();   // ---- End figure; write output file ----
}
```



- Change the color and width of the border. (`RGB(r,g,b);` creates an RGB color, CMY colors are analogous. Named primaries are available. The densities should be between 0 and 1 for "expected" behavior.)

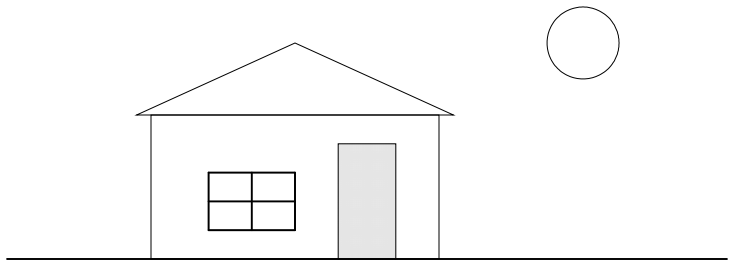
- Add `backing(Cyan());` after the `border` command.

- Put the command `crop_ellipse();` before the `border` command. Permute the `crop_ellipse` command with the `border` and `backing` lines, and note how the attribute (`crop`) affects objects (`border`, `backing`).

Geometric Objects

Our next file uses simple objects to draw a 2-D house-and-sun scene.

```
int main()
{
  picture(P(0,0), P(5,2), "3.75 x 1.5in");

  begin();

  triangle(P(0.9, 1), P(3.1, 1), P(2, 1.5)); // vertices
  circle(P(4,1.5), 0.25); // center and radius

  rect(P(1,0), P(3,1)); // opposite corners

  fill(Black(0.1));     // light gray
  rect(P(2.3,0), P(2.7,0.8)); // the door
  nofill();

  bold();               // draw thicker lines
  grid(P(1.4, 0.2), P(2, 0.6), 2, 2); // corners, number of squares

  line(P(xmin(), 0), P(xmax(), 0));   // endpoints

  end();
}
```



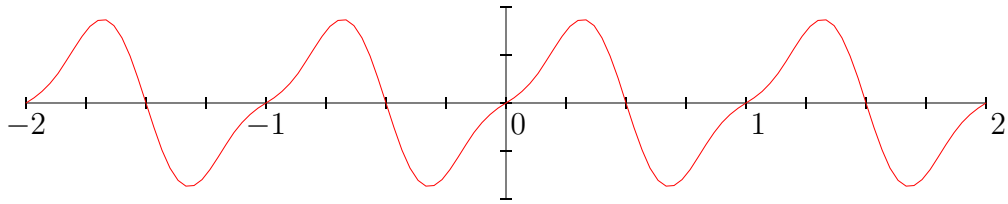
- Add declarations such as `bold(RGB(1,0.9,0.5))` or `fill(Yellow())` to color the scene. (Color and filling are orthogonal attributes.)

The sample file `house.flx` uses loops to draw gradient fills of the lawn, sky, and sun, and animates a sunset.

Function Plotting

Plotted functions must be defined in the "preamble", before `main`. "High-level" elements—coordinate axes and grids, axis labels, and graphs—are drawn with mnemonically-named commands.

```
// double = double-precision floating point
double f(double x) { return 0.75*Sin(x) - 0.25*Sin(2*x); }

int main()
{
  picture(P(-2,-1), P(2,1), "5 x 1in"); // [-2,2] x [-1,1]

  begin();
  revolutions(); // set angle units, [0,1] = one turn

  h_axis(16); // axes w/default endpts
  v_axis(4);
  h_axis_labels(4, P(0,-4), b); // shift down 4pt, align below

  plain(Red());
  plot(f, xmin(), xmax(), 120); // use 120 intervals

  end();
}
```



- Define and plot some different functions; adjust the bounding box as necessary. (Use repeated multiplication for polynomials. The `polarplot` command graphs $r = f(\theta)$.)

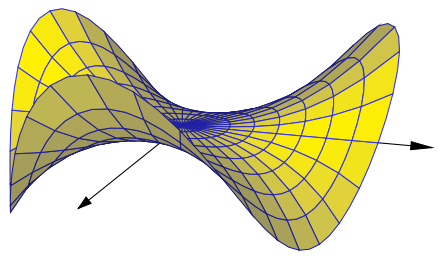
- Change the arguments to `h_axis_labels`. The first specifies the number of intervals to label; the second gives the label offset in `pt`. The last puts each label below (`b`) its Cartesian location.

Multivariable Plotting

Functions of two or three variables are defined just like functions of one variable, but for plotting the return type must be a point (`P`), not a real number (`double`). The `domain` class specifies the set of inputs to plot.

```
P f(double r, double th)
{
  return P(r*Cos(th), r*Sin(th), pow(r, 3)*Cos(3*th));
}

int main()
{
  picture(P(-1,-1), P(1, 1), "2 x 1in");
  begin();
  pst_format();     // use PSTricks macros for output

  // corners ([0,1] x [0,2\pi]) and fineness
  // (8x40 rectangles, plotted at 16x120 resolution)
  domain R(P(0,0), P(1, 2*M_PI), mesh(8,40), mesh(16,120));

  camera.at(P(3,1,2)); // set the viewpoint
  arrow(P(0,0,0), P(1.25,0,0)); // coordinate axes
  arrow(P(0,0,0), P(0,1.25,0));

  plain(Blue(1.2)); // line color and width
  fill(Yellow());   // shading color
  surface(f, R);
```

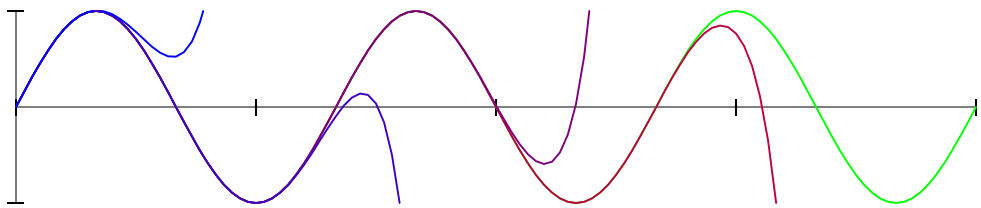
```
  end();
}
```



Loops and Control Structures

A function can be defined by an arbitrary algorithm, and a `domain` may be used to plot a family of functions for several values of one variable.

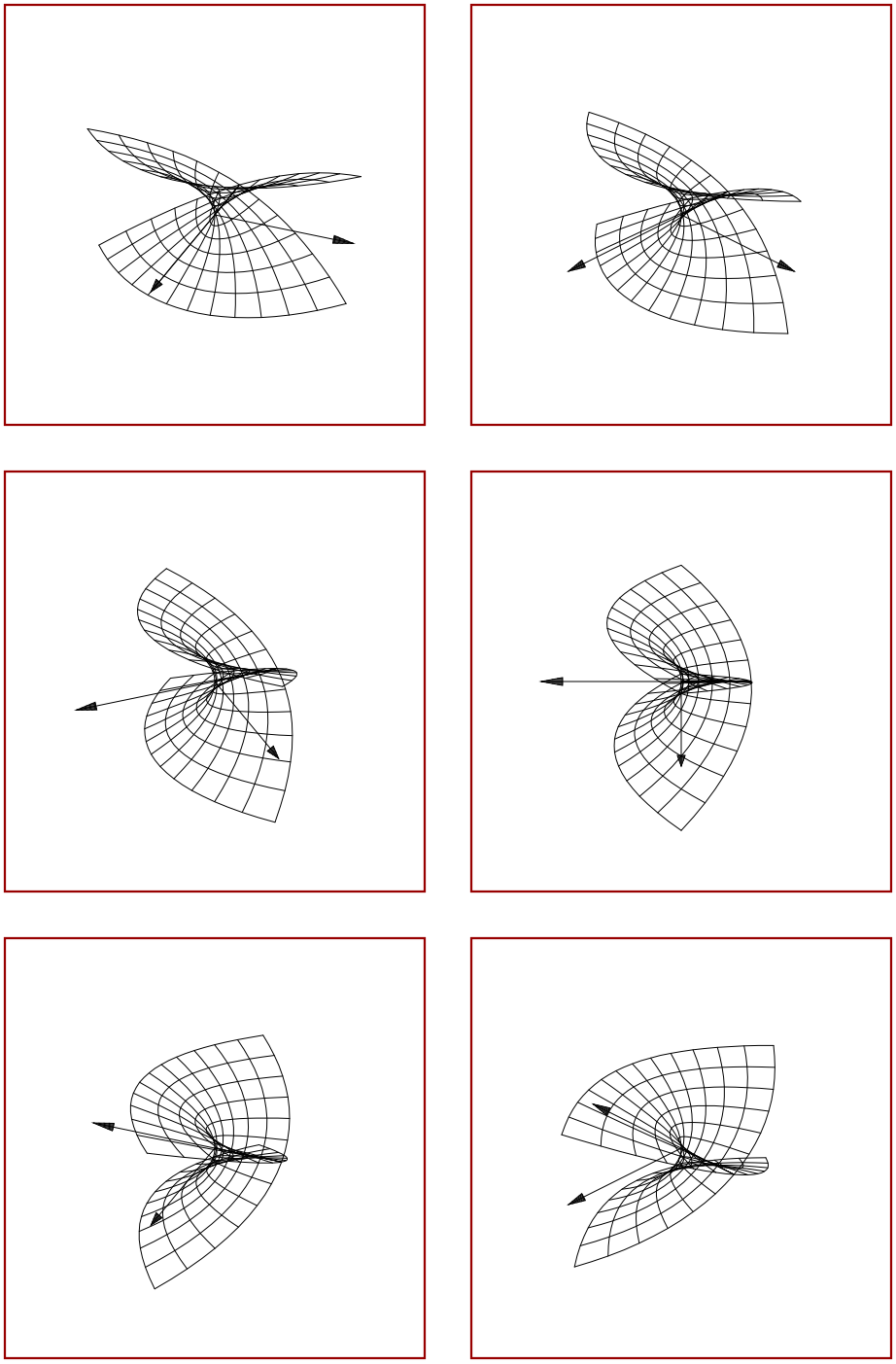
```
P sin_n(double x, double n)     // Taylor polynomial of sin x
{
  const int N((int) floor(n));  // convert n to an index bound
  const double sqx(-pow(x, 2)); // -x^2
  double val(x), summand(x);

  for (int i=1; i <= 2*N+1; i += 2)
    {
      summand *= (sqx/((i+1)*(i+2)));   // (-1)^i x^{2i+1}/(2i+1)!
      val += summand;
    }

  return P(x, val);    // return (x, y)
}

int main()
{
  picture(P(0, -1), P(6*M_PI, 1), "5 x 1in");

  begin();

  set_crop();
  bold(Green());
  plot(Sin, 0, xmax(), 120);

  domain R(P(0,1), P(6*M_PI, 41), mesh(60, 40), mesh(120, 40));
  for (int i=3; 0 <= i; --i)  // print in descending degree
    {
      bold(RGB(0.25*i, 0, 1-0.25*i)); // degree-dependent color
      plot(sin_n, R.slice2(5*i+1));   // plot for n = 5i+1
    }
  end();
}
```



Page Layout

Page layout can be composed from sub-pages with `screen` objects. In the loop body below, objects are added to the "active" `screen`, then `inset` into the "canvas", the screen representing the entire figure.

```
P f(double u, double v)
{
  return P((u-v)*(u+v), 2*u*v, u);
}

int main()
{
  picture(P(0,0), P(2,3), "5x7.5in"); // overall size

  begin();

  domain R(P(-1,-1), P(1,1), mesh(12,12), mesh(24,24));

  for (int i=0; i<2; ++i)
    for (int j=0; j<3; ++j)
      {
        screen my(P(-3,-3), P(3,3));
        activate(my);
        border(Red(0.6), "1pt");
        // frame-dependent viewpoint
        camera.at(sph(10, (2*j+i+1)*M_PI/8, M_PI/6));

        plot(f, R);
        arrow(P(0,0,0), 2*E_1);
        arrow(P(0,0,0), 2*E_2);
        // SW corner at (i, 2-j), padded by 0.05 on all sides
        inset(P(i+0.05,2.05-j), P(i+0.95,2.95-j));
      }

  end();
}
```



2.4 C++ Basics

An ePiX source file is a C++ program. If you've successfully modified and compiled 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 [3].

2.4.1 File Format

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.

A C++ file consists of "statements", analogous to ordinary sentences. Common types include *declarations* (which "register" a function, variable, or type name with the compiler), *definitions* (which assign meaning to declared names), and *function calls* (which cause a named function to execute). Most statements in an ePiX input file are function calls ("commands"). Plain declarations are relatively rare in user files, since a definition serves to declare any new names that it contains.

Every statement ends with a semicolon, and conventionally a file contains at most one statement per line. The compiler ignores nearly all whitespace (spaces, tabs, and newlines), which should be used freely 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.

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 loads the contents of the "header" file epix.h, 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.

2.4.2 Variables and Functions

Definitions of variables and functions play the same role in a figure that macro definitions play 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 (double-precision floating point number), P, and int. The name of a variable may only consist of letters (including the underscore character) and digits, and must begin with a letter:

```
my_var, var2,    MY_var, aLongVariableName;  // valid
my-var, 2var, \v@riable, $x, ${MY_VARIABLE}; // not valid
```

Variable names are case-sensitive, and numerous (non-universal) conventions govern the significance of capitalization. Generally, make names descriptive but not unwieldy, and avoid language keywords (such as `const`, `true`, `double`, `class`, or `public`) and names that begin with an underscore.

A function accepts "arguments" and "returns a value". To define a function in `C++`, 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);
}
```

defines the `double`-valued function f of one `double` variable defined by the formula $f(x) = \sqrt{1 - x^2}$.

2.4.3 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 multi-line comment, but a `C`-style comment may not; the compiler will mistake the first `*/` it encounters as the end of the current multi-line comment.

2.4.4 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. The contents of the output file start with `begin()` and terminate with `end()`. Intervening statements constitute the *body* of the file.

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

2.4.5 Strings and Raw Output

In `C++`, a `string` is a sequence of characters. Most `string`s in `ePiX` input files are *literals*, double-quoted `string`s whose value is read from the input. In a string literal, backslash is an escape character; 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

```
#include "epix.h"
using namespace ePiX;

int main()
{
  picture(P(-1,-1), P(1,1), "10cm x 3in);
  begin();
  pre_write("\\begin{figure}[hbt]"); // comes before the picture

  post_write("\\caption{A \\LaTeX\\ figure.}"); // and after
  post_write("\\end{figure}");

  < ... other ePiX commands ... >
  write("%% A comment near the end, but inside the picture.");
  end();
} // End of main()
```

Figure 2.1: Generating a self-contained figure in `ePiX`.

a space to separate an escape sequence from following text; the string "`\\textwidth`" literally represents a LaTeX command, while "`\textwidth`" is read "`TABextwidth`" by the compiler.

Though not commonly needed, raw text can be printed to the output file. The functions `write`, `pre_write`, and `post_write` accept `string` arguments. `write` prints its argument where the call appears in the input file. The other functions print their arguments before or after the completed picture, respectively. These commands must be in the file body. As an application, a complete LaTeX `figure` environment (with caption and label) can be produced by an `ePiX` file, Figure 2.1.

2.4.6 Conditionals and Loops

An algorithm's behavior usually depends on internal state. A *conditional statement* causes blocks of code to be executed according to criteria. 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 2.2 illustrates 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.)

2.5 Animation

`ePiX` is well-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

```
int gcd (int i, int j)
{
  int temp(i);  // initialization syntax
    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 2.2: Euclid's division algorithm in `C++`.

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 `pngs` and assembling these frames into a `mng` or `gif` animation. GraphicsMagick 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 command line 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 `samples` directory contains a handful of `flix` files that may be consulted for ideas.

By default, `flix` creates movies with 24 frames, in which `tix()` runs from 0 to 1, and animates at 0.08 sec/frame. These and other parameters can be changed with command-line options.

A "typical" `.flx` file may take 30 seconds to a few minutes to compile, depending on the number of frames and the complexity of each frame. 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.

2.6 Layout Tricks

2.6.1 Stereograms

Stereograms are created by drawing a single scene twice from slightly different points of view. To create a stereogram with `ePiX`, write a "scene" function containing the necessary drawing commands, then set the camera appropriately and call the scene function twice, once for each frame.

Crossed-Eyes Stereograms

The sample files `lorenz.xp` and `twisted_cubic.xp` use page layout to draw crossed-eyes stereograms. For each frame, create and activate a `screen`, set the camera, and call the scene function. The frames are inset side by side in the final picture. For a crossed-eyes stereogram, the frame from the "rightmost" camera position is placed on the left.

Bi-Color Stereograms

The sample files `cube.flx` and `mirrorball.flx` contain animated red-cyan stereograms, suitable for viewing with 3-D glasses. The technique works best with black-and-white line drawings, but the basic approach is the same as for crossed-eyes stereograms: Create a scene function and call it twice, setting the camera appropriately. Layout is unnecessary since the frames are superposed. Pen colors should be close to red and cyan, but "optimal" choices depend on one's eyes and 3-D glasses.

Stereographic Movies

Either form of stereogram can be animated in a `.flx` file, but keeping one's eyes properly crossed requires practice while most people can instantly see depth with 3-D glasses. As always when using `flix`, debug the scene with `elaps` before compiling a movie. If a stereoscopic effect is difficult to see in a still image, it will be even harder to perceive in a movie.

2.6.2 Inset Images

`ePiX` can place external images into a figure, similarly to Rolf Niepraschk's `overpic` package. You provide the Cartesian center point, the file name, and optionally the true height and/or width of the image:

```
label(P(a,b), "\\includegraphics[width=w,height=h]{file}");
```

This effect requires the `graphicx` package. When compiling a stand-alone graphic containing an external image, you must specify the `graphicx` package on the command line: `elaps -p graphicx <file>`

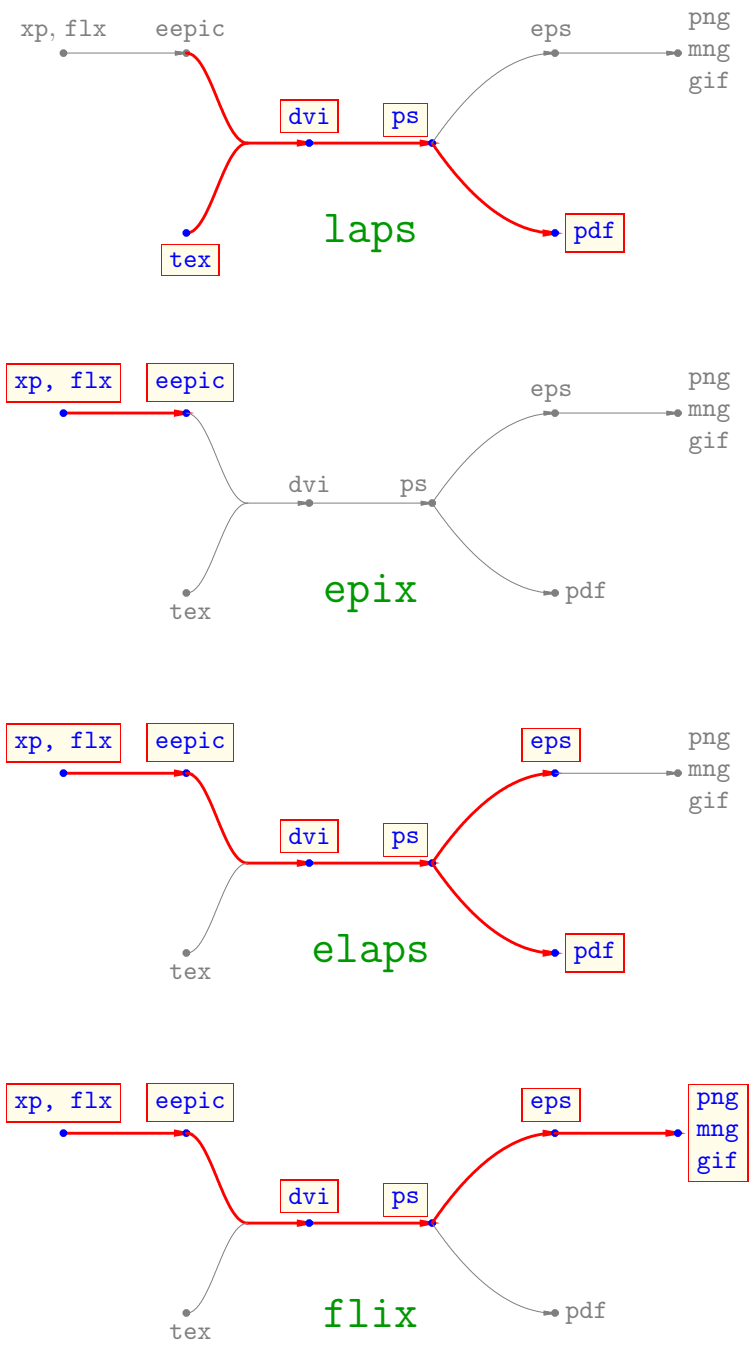


Figure 2.3: Visual guide to `ePiX`'s shell scripts

Chapter 3

Reference Manual

This chapter details ePiX's capabilities, discussing attributes and data types, and listing available commands in the form

```
label(P posn, [P offset], string text, [align]);
circle(P ctr, double rad=1, normal=P(0,0,1));
```

As in Chapter 2, function arguments are given by type (P, double, string, etc.) and name, or by name alone if the type is clear. Optional arguments are enclosed in square brackets. A few argument types, such as [align] above, admit only a small number of values; these cases are explained when the corresponding command is first introduced.

A name followed by an equals sign and a value indicates a default argument; if omitted in an input file, the compiler substitutes the default value. Only trailing arguments may be specified this way. For example, if the rad argument of the circle command is omitted, the normal argument *must* be omitted as well.

In an input file, only an argument's value is given, not the type:

```
label(P(0,0), P(0,-4), "Hello world", b);
circle(P(0,-0.25), sqrt(2)); // use default normal
```

Generally, len signifies either a number (representing a length in pt) or a double-quoted string containing a number and a two-letter LaTeX length unit, such as "1.5pt" or "6cm". color represents a Color object constructed from a named primary (Blue(), Magenta(0.7), etc.), a color specification (RGB(r,g,b), CMY(c,m,y), etc.), or an operator applied to an existing Color.

3.1 File Structure

An ePiX input file constitutes a short C++ program. When this program is compiled and run by one of the shell scripts, it creates a figure file suitable for inclusion in LaTeX.

Like a LaTeX document, an `ePiX` file contains a *preamble*, which sets up a drawing environment, and a *body*, which contains actual figure-generating commands. The minimal file has the form

```
#include "epix.h"  // N.B. no semicolon
using namespace ePiX;

int main() {
  picture(P(a1,b1), P(a2,b2), "n1 [unit1] x n2 unit2");
  begin(); // end of preamble, start of body
  end();   // end of body
}
```

`ePiX` commands are of four general types: drawing, attribute setting, definitions (of data and functions), and operations on existing objects. Except as noted below, drawing and attribute commands must appear in the body, between `begin()` and `end()`. Function definitions must come in the preamble, before `main()`. Data definitions may appear in the preamble or body.

Output Format

The `end()` command writes the output file to `stdout`, using `eepic` macros by default. The shell scripts redirect `stdout` to an appropriate disk file.

The attribute-setting command `pst_format()` causes the output file to be written using `PSTricks` macros when `end` is called. Similarly, `tikz_format()` causes the file to be written using `tikz` macros, and `eepic_format()` causes the file to be written using `eepic` macros. These commands may appear anywhere in the figure body. There is no reason for a file to contain more than one such command.

The output format may be selected on the command line, overriding any explicit request in the input file. Supplying `epix`, `elaps`, or `flix` with one of the options `--pst`, `--tikz`, or `--eepic` is tantamount to issuing a `pst_format()` (etc.) command just before the end of the file. These options are listed in decreasing precedence. If more than one is given, the "strongest" applies, regardless of the command line order.

A figure may be written directly to a specified disk file, in a format unaffected by the command line flags above. The command

```
  print_eepic("file.tex");
```

writes the figure to the named file using `eepic` macros. Analogous commands exist for the formats `pst` and `tikz`. These commands perform an immediate action. Consequently, the command's location in the input file is significant, and multiple commands may appear in a single file, so long as distinct file name arguments are provided. Applications include writing the same figure in multiple formats, or creating successive "snapshots" of a lengthy computation.

3.2 Picture Size and Aspect Ratio

The `picture` command specifies a figure's logical and true sizes. In the "minimal file" snippet above, the *canvas* is the Cartesian rectangle $[a_1, a_2] \times [b_1, b_2]$ whose corners are given. Either pair of opposite corners is acceptable, but confusion is less likely when the SW and NE corners are given, in this order.

The true (printed) width and height are read from `picture`'s third argument, a "size `string`" such as `"4cm x 1in"`, containing a number and optional LaTeX length unit, an `x`, another number, and a mandatory length unit. Spaces may be used for legibility. The mandatory length unit becomes the LaTeX `unitlength`. The argument `"4cm x 1in"` creates a picture 4 cm wide and 1 in high, but internally converts 4 cm to inches. Recognized length units are `pt` (points, the default), `cm` (centimeters), `in` (inches), `mm` (millimeters), `pc` (picas), and `bp` (big points). (`1in = 2.54cm = 72bp = 72.27pt`, `1pc = 12pt`.)

The logical and true sizes may be defined separately with the commands

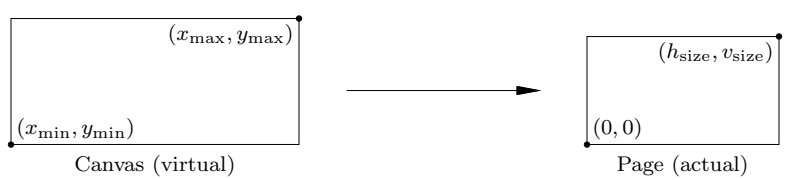
```
bounding_box(P(a1,b1), P(a2,b2));
picture(n1, n2);
unitlength(len); // len a length string, not a double
```

One way or another, the logical and true dimensions *must* have positive values when the `begin()` command is issued.

LaTeX treats the contents of a picture environment as a single box, aligned by default on its lower left corner. An `offset` command accepts two `double` arguments or a size `string` and shifts the page location accordingly. The command `offset("0.25 x -0.5cm")` shifts the picture right 0.25 cm and down 0.5 cm. If the `unitlength` is 1 cm, the command `offset(0.25, -0.5)` has the same effect.

A non-zero `offset` causes a picture's contents to appear in a location where LaTeX does not expect them. This can be useful in a LaTeX document, but should be avoided when compiling a stand-alone image, since `dvips` may crop according to rules of its own.

The canvas's corners are `(xmin(),ymin())` and `(xmax(),ymax())`, while its width and height are `xsize()` and `ysize()`. The canvas is a virtual, advisory data structure; its dimensions are not directly related to the figure's printed size, and picture elements may lie outside the canvas. Affine scaling maps the canvas to the page when the output file is written.



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

3.3 Color

ePiX provides a Color data type. Four models are implemented: RGB, CMY, CMYK, and Gray. A Color holds primary color "channels", each carrying an intensity between 0 (no color) and 1 (full saturation). "No color" means black in RGB and Gray, white in CMY(K). RGB, CMY, and CMYK colors are written to the output file as commands in the corresponding model. Gray shades are written in RGB.

Operations on red-green-blue colors are described below. Functionally, colors are converted to RGB, operated upon, then converted back to the original model. Conversions are as described in Uwe Kern's xcolor manual [2].

3.3.1 Constructors

Each color model has a "constructor" creating a color of specified densities. The RGB and CMY models have named "primary" constructors; the density argument is optional and defaults to 1.

```
// red-green-blue colors
RGB(double r=0, double g=0, double b=0);
Red(d=1);   Green(d=1);   Blue(d=1);
White(d=1);   Black(d=1);

// cyan-magenta-yellow colors
CMY(double c=0, double m=0, double y=0);
Cyan(d=1);   Magenta(d=1);   Yellow(d=1);
CMY_White(d=1);   CMY_Black(d=1);

// gray
Gray(double d=0); // equivalent to RGB(d, d, d)
```

Each CMY constructor has a corresponding CMYK function, e.g. CyanK() or CMYK_White().

Though color densities lie between 0 and 1, ePiX's primary color constructors take arguments mod 4, viewed as elements of $[-2, 2]$. Consider Red(d), "red with density d". For integer values of d, the constructor has the following meanings: Red(0) is black, Red(1) is red, Red(2)=Red(-2) is white, and Red(-1) is anti-red, or cyan. For non-integer d, the constructor interpolates between the bracketing integer values, Figure 3.1. Other primary constructors work analogously.

Let $\delta : [-2, 2] \to [0, 1]$ be the piecewise-linear function that converts real numbers to primary color densities, clip : $\mathbf{R} \to [0, 1]$ the clipping function. The non-primary constructor RGB(r, g, b) computes the density of the red channel as $\mathrm{clip}\big(\delta(r) + \delta(-g) + \delta(-b)\big)$; the green and blue channel densities are computed similarly. Every color can be created with arguments between 0 and 1, but the constructor accepts arbitrary real arguments and returns colors varying "continuously and periodically".

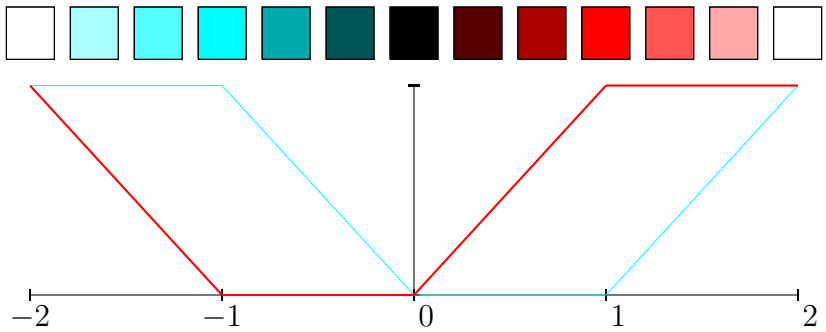


Figure 3.1: The red primary constructor.

Colors possess an "alpha-channel" for transparency. Except as supported by the output format, this feature is a stub.

3.3.2 Color Operations

Colors can be scaled, inverted, blended, superposed, and filtered. In the code below, `tint` is a `Color`. The effect of each operation is described in the `rgb` model; the visual result is the same in all models.

```
tint *= double c; // multiply channels by c, rebuild
tint.invert();    // (r, g, b) -> (1-r, 1-g, 1-b)
tint.blend(Color col, double t);  // (1-t)*tint + t*col
tint.superpose(col); // add channels, then clip to [0,1]
tint.alpha(double d); // set alpha channel to clip(d)

tint.filter(col); // return min density in each channel
```

Except for `filter`, these operators modify their object. `filter` simulates the effect of viewing `col` through a transparent sheet of `tint`, and returns a new `Color` object having the same model as `tint` without modifying `tint` itself.

3.4 Scene Attributes

Objects in a scene sit in 3-dimensional space. A *camera* maps objects to the *active screen*. The active `screen` may, in turn crop its contents. Each drawing command creates an object, "photographs" it, and adds the image to the active `screen`. This section describes the `Camera` and `screen` classes and a few associated concepts.

3.4.1 Angular Mode

By default, angles are measured in `radians`. Two other angular modes are available: `degrees` and `revolutions`. The angular mode is set with a named command,

`radians()`, `degrees()`, or `revolutions()`. The current angular mode affects all trigonometric functions and operations.

3.4.2 The Camera

Three-dimensional scenes are drawn on flat paper by applying a mathematical transformation. By default, `ePiX` uses *point projection*, the technique used by art students when they trace on a window with grease pencil, Figure 3.2.

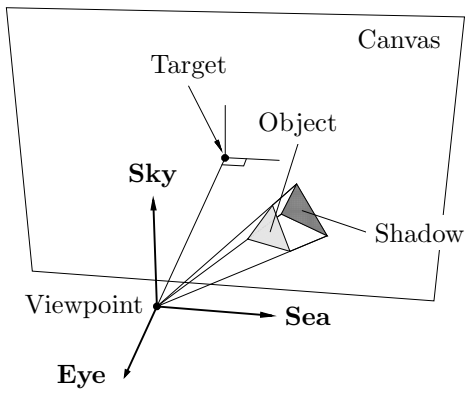


Figure 3.2: Point projection.

`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), a *lens* (the actual mapping to the screen plane), and a *filter* (a color through which the scene is viewed).

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 *viewpoint*, the camera's spatial location. The *target*, the origin of the screen plane, lies on the line through the viewpoint in the direction opposite the eye vector. The distance from the viewpoint to the target is the *range*. The *camera sphere* is centered at the viewpoint and passes through the target. The orientation, viewpoint, target, and range completely (and redundantly) determine the camera's geometric situation in the world.

The Lens

A *lens* maps the world to the screen. `ePiX` comes with four lenses: *shadow* (the default), *orthogonal*, *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. The orthogonal lens projects from infinite distance. Each of the other lenses performs radial projection to the camera sphere, then maps the sphere to the screen plane; the fisheye lens does orthogonal projection (so the entire image lies inside the disk centered at the target whose radius is the range) while the bubble lens does stereographic projection from the target's antipode.

An input file normally uses a single camera, so a global object named `camera` is defined automatically. At the start of a figure, the `camera` looks down on the (x_1, x_2)-plane from a distant point on the x_3-axis. The resulting view, essentially projection along the axis, is suitable for 2-dimensional figures.

The `camera` is manipulated similarly to a real camera:

```
camera.at(P posn);          // set viewpoint to posn
camera.look_at(P targ);     // set target to targ
camera.range(double dist);  // fix target, move viewpoint
camera.focus(double dist);  // fix viewpoint, move target
camera.tilt(double angle);  // rotate about the sea axis
camera.pan(double angle);   // rotate about the sky axis
camera.roll(double angle);  // rotate about the eye axis
camera.clip_range(dist);    // cull scenery closer than dist
```

Explicitly setting the viewpoint or target makes the sky vector parallel to the projection of the x_3 axis when possible; otherwise the x_2-axis is used. Setting the range or focus moves the `camera` parallel along the eye line. Each command re-sizes the image; note that increasing the focus *enlarges* the image. The three rotation operations fix the viewpoint, but only roll fixes the target.

Filtering and Color Separation

The `camera` has a *filter* through which all scene `Color`s pass. For uniformity, the filter should only be set at the start of the file body, before any visible elements have been placed in the scene. The filter has two primary uses—model conversion and color separation—but can also be used for special effects.

There is a `Neutral` color, for which filtering has no effect at all. The `camera`'s default filter is `Neutral`. In addition, each color model has a *neutral* representative, `RGB_Neutral`, etc. Passing `Color`s through a neutral filter preserves their appearance but converts them to the neutral `Color`'s model.

Filtering can be used to "split" a `Color` into primary constituents. For `RGB` channels, the primaries themselves are suitable filters. For `CMYK` channels, there are special

process filters, named `C_Process`, etc.

```
camera.filter(CMY_Neutral());  // convert all colors to CMY
camera.filter(Gray_Neutral()); // black and white photography
camera.filter(Green());        // the green channel
camera.filter(M_Process());    // the magenta channel
camera.filter(Red(1.4));       // la vie en rose?
```

3.4.3 Clipping

Two operations handle elements lying far from the target. *Cropping* culls elements based on their screen location, and is described later. *Clipping* removes objects whose spatial location lies outside the "clip region".

Initially, the clip region is a very large box centered at the origin. Commands are provided to resize this box, keeping the faces parallel to coordinate planes.

```
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)
clip_box();             // very large box
```

Additional "clippers" may be added manually. Individual clipping planes are specified by a point and inward-pointing normal vector. Parallel planes cutting a "slice" or "slab" are described by a location, direction, and a distance. The location lies halfway between the clipping planes, and the planes' normals point toward the location.

```
clip_face(P loc, P perp); // perp points inward
clip_slice(loc, perp);    // very close parallel planes
clip_slice(loc, perp, dist); // planes separated by dist
clip_restore();  // remove manually-added clip planes
```

Like `clip_restore`, the `clip_box` and `clip_to` commands remove all user-specified half spaces.

3.4.4 Screens and Page Layout

By default, drawing occurs in the Cartesian coordinate system of the `canvas`. However, insets and subfigures are most naturally composed in "local" coordinates, then placed into their final location as a unit, a process called *layout*. `ePiX` implements layout with the `screen` class.

Screens

A `screen` is a Cartesian plane containing a distinguished coordinate rectangle, its *bounding box*. The `canvas` of a figure is a `screen`, as is the internal representation of the printed page. At any point of a file body, some `screen` is *active*, or "open for drawing". A new `screen` is created from a pair of opposite corners, but is not used until explicitly activated.

```
screen subfig(P(a1,b1), P(a2,b2)); // [a1, a2] x [b1, b2]
activate(subfig);   // open subfig for drawing
// commands to draw in subfig
deactivate(subfig); // re-activate the previous screen
```

The `canvas` is automatically drawn at the `end` of the input file. The contents of other `screen`s must be incorporated into the `canvas` explicitly with an `import` or `inset` command. `import` affinely scales the bounding box of the "child" `screen` to the `canvas`. `inset` specifies corners in the "parent" where the child is placed.

```
import();           // active screen to canvas
import(subfig);     // subfig to canvas
inset(P sw, P ne);  // active screen to specified rectangle
inset(subfig, sw, ne); // subfig to specified rectangle
inset(subfig);      // inset to subfig's corners
```

Extracting

The portion of a `screen` enclosed by a specified rectangle can be *extracted*. Each of the commands

```
subfig.extract(P sw, P ne);
canvas().extract(P sw, P ne);
```

returns the `screen` having stated corners and containing part of `subfig` or the `canvas`, respectively. Elliptical and diamond-shaped portions of a `screen` may be extracted. The contents occupy the ellipse or diamond inscribed in the given rectangle.

```
subfig.extract_ellipse(sw, ne);
canvas().extract_diamond(sw, ne);
```

The `screen` returned by `extract` (or its variants) may be used just like a manually-constructed `screen`, for example to construct a magnified inset.

True-Size Drawing

On occasion, it is necessary to draw at known page size, independently of the unit length. For 2-dimensional drawing in the (x_1, x_2)-plane with the `camera` in its default location, the coordinates (horizontal, vertical, or both) of a `screen` may be interpreted as true pt offsets relative to a specified point rather than as Cartesian coordinates.

```
inlay(subfig, P loc);
inlay_horizontal(subfig, loc);
inlay_vertical(subfig, loc);
```

Each command maps the origin of `subfig` to the specified location in the active `screen`. The first performs affine scaling so that one Cartesian unit in `subfig` maps to one true pt on the page. The second and third commands perform this scaling only in the named direction.

The `inlay` functions are suitable only for 2-dimensional effects. For 3-dimensional true-size drawing, the function `pt_to_screen` converts a true length of 1 pt into Cartesian coordinates in the active `screen`. For example, the command `line(P(0,0), P(pt_to_screen(12),0))` draws a line segment 12 pt long with its left endpoint at the Cartesian origin. True size drawing is unusual in an input file, but has definite uses in library code; `ePiX`'s right angle marker is a typical application.

Either type of true-size drawing works as described when drawing in the `canvas`, and *only* when drawing in a `screen` imported to the `canvas` at "true Cartesian" size (one `screen` unit maps to one Cartesian unit in the `canvas`). In other situations, the true size coordinate(s) will be scaled by additional affine transformation(s).

Cropping and Decorations

A `screen` has a *crop mask*: rectangular (default), elliptical, or diamond-shaped. If cropping is switched on in the active `screen`, added elements are cropped to the crop mask. For example, cropping by the default crop mask in the `canvas` ensures the figure lies inside the printed region allocated by LaTeX.

The `crop` commands below affect the active `screen` only. Cropping is not a "global" attribute, but must be set or unset explicitly for each `screen`. By default, cropping is switched off.

```
set_crop();      // activate cropping
set_crop(false); // deactivate cropping
crop_diamond();  // <>-shaped mask, set crop state
crop_ellipse();  // etc...
crop();          // crop active screen's contents
```

Normally the crop mask is inscribed in the `screen`'s bounding box. The commands `crop_diamond` and `crop_ellipse` also accept a pair of `P` arguments, which are treated as corners of the crop box.

The active `screen` may be decorated with a `border` (outline of specified color and line width) and `backing` (solid fill color). The crop mask determines the shape of the `border` and `backing`.

```
border(color, len);
backing(color);
```

Identically-named member functions can be applied to an arbitrary `screen` whether or not it is active:

```
scr.crop_rectangle().backing(Blue(1.8));
canvas().crop();
```

Affine Maps

A *plane affine map* has the form $T(x) = Ax+b$ for some invertible 2×2 matrix A and a constant vector b. `ePiX`'s `affine` class allows affine maps to be built and applied to `screen`s' contents.

An affine map is uniquely determined by the images of three non-collinear points. The constructor returns the affine map sending the points $(1,0)$, $(0,1)$, and $(0,0)$ to `pt1`, `pt2`, and `pt0`, respectively.

```
affine af(P pt1, P pt2, P pt0=P(0,0));
```

To emphasize, the arguments are locations, not displacements, and the image of the origin comes last. There is also a `void` constructor (taking no arguments) which returns the identity map.

To facilitate construction of affine maps, an existing `affine` may be post-composed with a variety of "elementary" affine transformations. In the commands below, `th` is an angle (in current units), `sc` is a non-zero `double`, and `ctr` is a point fixed by the composing transformation. In each command, `ctr` defaults to $(0,0)$.

```
affine af; // the identity map
af.shift(P arg);    // translate by arg
af.rotate(th, [ctr]);  // counterclockwise rotation about ctr
af.reflect(th, [ctr]); // reflect across line through ctr

af.h_scale(sc, [ctr]); // horizontal scaling
af.v_scale(sc, [ctr]); // vertical scaling
af.scale(sc, [ctr]);   // dilatation

af.h_shear(sc, [ctr]); // shear preserving horizontals
af.v_shear(sc, [ctr]); // shear preserving verticals

af.invert();           // the inverse
```

```
af.postcomp(affine f); // post-compose with f
af(f);                 // pre-compose, af not modified
```

A non-invertible `affine` can be created *only* by shearing or scaling with an extremely large or extremely small argument, or by supplying three collinear points to the constructor. Calling `invert` on a non-invertible map merely issues a warning and performs no action.

An `affine` may be applied to a `screen`'s current contents. The `screen` class has eight member functions (`shift` through `shear`) with syntax identical to the affine map functions. In addition, an arbitrary `affine` may be applied to a `screen`:

```
scr.shift(arg); // shift scr's contents by arg; etc.
scr.apply(f);   // apply f to scr's contents
```

To apply a composition of several maps to a `screen`, it's best to build an `affine` map by composition, then `apply` the map. Composing `affine`s is cheap; `applying` an `affine` is costly in proportion to the number of elements in the `screen`.

Applying an `affine` to a `screen` has no effect on the bounding box, `border`, or `backing`, and may move elements outside the bounding box even if cropping is active. To ensure a `screen`'s contents lie inside the bounding box, `crop` the `screen` *after* `applying` the `affine`(s).

The sample file `inverse.xp` uses `affine` maps to depict branches of inverse functions in one variable, `symmetry.xp` depicts the permutation group S_3 by its action on a regular hexagon.

3.5 Drawing Attributes

`ePiX` maintains drawing states for filled regions, paths, and text objects.

3.5.1 Filled Regions

Filling is either on or off. When filling is active, closed paths are filled with the current *fill color*.

```
fill();   // turn filling on
fill(color); // turn filling on, specify color
nofill(); // turn filling off, same as fill(false);
```

3.5.2 Paths

Paths and borders of filled regions are drawn with two *pen*s, each described by color and line width. The *line pen* draws all path-like objects. The *base pen* does nothing unless it is wider then the line pen, in which case it draws an "underlayer" or "border" on the line pen.

```
pen(len);          // set line pen width
pen(color);        // set line color, keep width
pen(color, len);   // set color and width
```

As usual, `len` may be either a length string or a `double`, interpreted as a width in `pt`. There are `base` functions with the same signatures and analogous meanings for the base pen.

The line width can be set with named declarations; the optional argument sets the line color:

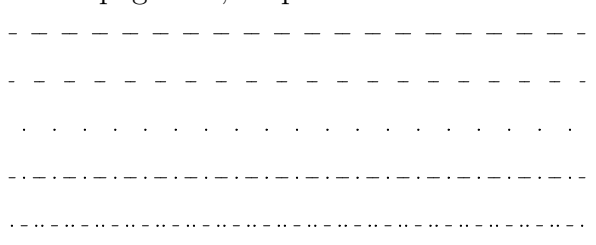
```
plain([color]); // 0.4pt
bold([color]);  // 0.8pt
bbold([color]); // 1.6pt
```

Path Style

By default, path-like objects are drawn with solid lines. Dashed, dotted, and free-form path style patterns are also available. (The `base` underlayer is always solid.) The page length of a pattern defaults to 12 pt, but can be set.

```
line_style(string);
dash_size(len=12);
```

The path style is set with a WYSIWYG string of dashes, spaces, and periods, representing a pattern of dashes, gaps, and dots. In the sample styles below, the repeating units have the same page size, 12 pt.



For brevity and uniformity, named commands are provided.

```
solid(); "-"     dashed(); "-  -"    dotted(); " . "
```

A sequence of n dashes, spaces, and dots corresponds to a dash/dot pattern in an interval divided into n subintervals of equal length. If the ith character is a dash or space, the ith subinterval is drawn solid or empty, respectively. If the ith character is a period, a dot is placed at the midpoint of the ith subinterval.

This pattern is applied to a path-like object as follows. The page length of each edge is divided by the current `dash_size` and the ceiling (next largest integer) taken. This many copies of the current path style are scaled onto the edge. The first and last characters are adjacent in repeated units.

There are three inequivalent ways to adjust the dash length/dot spacing in a non-line path: Change the `dash_length`, create a path with a different number of points, or use a longer, repetitive pattern. For best results, the style string should not be longer than about a dozen characters.

If you need several dashed/dotted line styles in multiple figures, it's best to define a custom header instead of hard-coding line styles. See Section 4.2 for guidance.

3.5.3 Text Objects

Two types of textual element may appear in a file: *labels* (text boxes) and *markers* (LaTeX symbols). A marker occupies a box of zero size, and is placed at a specified Cartesian location. A label has typographical size, and is usually offset from its Cartesian location. For placement, an *alignment point* is attached to each label, and Cartesian coordinates position the alignment point.

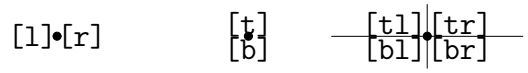
Labels

A label is printed as a LaTeX box. By default, the alignment point 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: $y = f(x)$

The alignment point may be *offset* manually by a specified number of `pt`. Additionally, a label's location with respect to the alignment point can be chosen with an optional LaTeX-style alignment option. This scheme allows labels to be placed easily where they will not overlap other parts of the figure, and ensures labels stay properly positioned when the size or aspect ratio of a figure changes.

```
label(P posn, P offset, string msg, [align]);
label(posn, msg);
```

The first two components of the `offset` argument are numbers of `pt` to shift the alignment point right and up. The optional `align` argument may be one—or an appropriate pair—of `t`, `b`, `r`, or `l` (top, bottom, right, left), or `c` (center). These alignment options specify the position of the label *relative to the Cartesian location* `posn`, namely they work *opposite* to the way they work in LaTeX.



The `msg` argument is usually a snippet of LaTeX code enclosed in double quotes. `C++` treats "\" as an escape character, so a double backslash is needed in the source to get a single backslash in the output. For example,

```
label(P(0,0), P(2,-1), "$\\rho = \\sin\\theta$", br);
```

positions an alignment point `2pt` right and `1pt` below the (Cartesian) origin, and typesets the equation $\rho = \sin\theta$ below and to the right.

Labels can be rotated; the (counterclockwise) angle is set in current angle units with the command `label_angle(theta)`. For example, a rotation angle of 90 degrees prints labels along a vertical axis. Though label rotation has legitimate uses, it can make labels more difficult to read, defeating their purpose. Do not use rotated labels merely because they are available. As a practical matter, if an output file contains rotated labels, the enclosing document must use the `rotating` package. `elaps` automatically loads this package.

An `affine` behaves moderately intelligently when acting on a marker or label. The mapping is applied to the label's location, and the "linear part" is used to adjust the offset and label angle. No attempt is made to treat alignment. For best results, if a `screen` will have an `affine` applied to it, label positions should be fine-tuned only with offsets, not alignment arguments. In all situations, the font itself is unchanged; no attempt is made to print sheared, scaled, or reflected text. The sample file `inverse.xp` illustrates the effect of affine maps on labels.

Fonts and Type Size

By default, the font in an `ePiX` figure is that of the enclosing document. The font size and face are changed with "declaration-style" commands such as

```
font_size("Large");
font_face("sc");
```

The argument of `font_size` is a valid LaTeX size. If no argument is given, `normalsize` is understood. `font_face` accepts a two-letter string, appended to the string "`text`" to give a LaTeX font declaration command ("`textsc`" above). Finer-grained control is accomplished by placing LaTeX commands into the label text.

Label Attributes and Masked Labels

Each label command has a corresponding "mask" version (`masklabel`) that draws an opaque rectangle under the label text. The mask size exceeds the label's size by an amount of *padding*, and the mask is itself surrounded by a rectangular *border*, of specified color and width.

```
label_color(color);         // set label text color
label_mask(color=White());  // set mask color
label_pad(string len);

label_border(color, [len]); // set color (and line width)
label_border(len);          // set line width only
no_label_border();          // turn off label borders
```

CIRC	SPOT	RING	DOT	DDOT
PLUS	OPLUS	TIMES	OTIMES	
DIAMOND	UP	DOWN	BOX	BBOX

Table 3.1: ePiX's marker types.

Markers

ePiX markers are obtained with `marker(P pt, <MARKER TYPE>);` (Table 3.1.) Several "dot-like" marker types are available by name:

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

A `circ` is filled with the current `mask_color`, and a `ring` is "hollow". Each dot-like marker can also be called with label syntax, generating a labeled marker with one command.

```
dot(P posn, offset=P(0,0), msg="", align=none); // etc.
```

By default, `spot` and `ring` are 4 pt in diameter; `dot`, `box`, and `circ` are 3 pt in diameter; `ddot` and `bbox` are 2 pt in diameter. At arbitrary diameter, a `spot` is 4/3 the diameter of a `dot` and a `ddot` is 2/3 the diameter. The command `dot_size(diam=3)` sets the diameter of a `dot`, and hence the size of all dot-like markers.

Reminders

When constructing and placing a label,

- Offsets are specified in `pt` (true length), not Cartesian units: A label's location relative to its 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.

3.5.4 Color Declarations

Old-style color declarations set the fill color, line color, and text color.

```
rgb(r, g, b);      cmyk(c, m, y, k);
rgb(P);            cmyk(P);  // for function-controlled colors
red(d);  // similarly for other primaries
```

3.6 Creating and Drawing Objects

Scene elements include geometric objects, coordinate grids, axis labels, and function plots. A few commands create an object (a point, line segment, circle, sphere, or plane) that can be used in subsequent computations, but most drawing commands automatically create, draw, and discard objects.

3.6.1 Geometric Data Structures

Points

The simplest object in the world, and by far the most common named data structure, is `P`, 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. The standard basis is available: `E_1=P(1,0,0)`, etc.

Depending on context, a `P` may represent either a *location* (point) or a *displacement* (vector). Almost all `ePiX` functions treat a `P` as a point. However, algebraic operators and commands that plot vector fields treat `P` arguments as displacements.

Polar, cylindrical, and spherical coordinate `P` constructors are sensitive to the current angular mode.

```
P pt=polar(r,t);   // (r*Cos(t), r*Sin(t), 0)
P pt=cis(t);       // (Cos(t), Sin(t), 0) = polar(1, t)
P pt=cyl(r,t,z);   // (r*Cos(t), r*Sin(t), z)
P pt=sph(r,t,phi); // polar(r,t)*Cos(phi) + (0,0,r*Sin(phi))
```

Algebraic operations—addition/subtraction, scalar multiplication; scalar, cross, and componentwise products; orthogonalization—can be performed on `P`s. In compound expressions, the binary operators below should be enclosed in parentheses, and scalars must be collected at left, `P`s at right.

```
double u=pt.x1();   // first coordinate of pt, etc.
P(a,b,c)|P(x,y,z);  // scalar product, ax+by+cz
P(a,b,c)&P(x,y,z);  // componentwise product (ax, by, cz)
P(a,b,c)*P(x,y,z);  // cross product (bz-cy, cx-az, ay-bx)
J(p);               // quarter turn about the x3-axis
p%q;                // orthogonalization, p (mod q)
```

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

`P` operations express mathematical relationships, and therefore imbue a figure with logical structure, making the input file easier to read, modify, and maintain. Commonly, a file preamble will define a few named points with hard-coded coordinates, then define additional points of interest using `P` operators.

Complex Numbers

The `Complex` class represents a complex number, specified by its real and imaginary parts. Arithmetic operators (addition, subtraction, multiplication, division) are provided, as well as exponential, logarithmic, circular, and hyperbolic functions (sensitive to angle units), the latter with `C` appended to their names. A complex number implicitly converts to a point with third coordinate zero. The sample file `cubic_cutaway.xp` illustrates usage, including how to define and plot functions and color-shade Riemann surfaces.

```
Complex u, z(4,3);  // u = 0, z = 4 + 3i
double len(norm(z));// len = 5
double th(Arg(z));  // th = atan2(3, 4);
P pt(z);            // pt = (4, 3, 0)
Complex w(expC(z)); // w = e^{4 + 3i}
u = LogC(z,k=0);    // u = Log(z) + 2*pi*ki
u = sqrtC(z,k=0);   // sqrt(z)*expC(pi*ki)
u = rootC(z,n,k=0); // (z^{1/n})*expC(2*pi*ki/n)
u = powC(z,n);      // z^n
```

Other Geometric Classes

In addition to `P`, objects of type `Circle`, `Plane`, `Segment`, and `Sphere` can be used for Euclidean geometry constructions. Simple affine operations are supplied for each type, as is a `draw()` function, which represents the object as a path in the screen.

```
obj.shift(P arg);    // translate by arg
obj.move_to(P arg);  // move center to arg
obj.scale(double c); // scale about center by c
obj.draw();
```

A `Segment`'s "center" is its midpoint. A `Plane` has no center; `move_to` translates the `Plane` to pass through `arg`, and `scale` has no effect.

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

```
Circle(center=P(0,0,0), double rad=1, normal=E_3);
Circle(P center, P point);
Circle(P p1, P p2, P p3);
```

The second constructor creates the `Circle` parallel to the (x_1, x_2) plane, with given center, and radius equal to the distance between the arguments. (A warning is printed if the second argument does not lie on the circle, namely, if the arguments do not lie in a plane parallel to the (x_1, x_2) plane.) The third returns the `Circle` passing through the given points; the points must not be collinear.

The data defining a `Circle` are recovered with member functions named `center()`, `radius()`, and `perp()`.

A `Plane` is specified by a point and normal vector, or by three non-collinear points. The `draw()` function clips the plane and draws the resulting polygon. Unless the clip box has been set manually, the clipped polygon's vertices will have large coordinates.

A `Segment` is constructed from its endpoints. The member function `midpoint()` returns the center.

A `Sphere` is specified by a point and a radius—by default the origin and unity, or by the center and a point on the sphere. Member functions `center()` and `radius()` return the defining data. Capabilities specific to geography and spherical geometry are described below, pp. 59ff.

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 is generally an ellipse. Antipodal points are not generally mapped to points 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`.

Intersection

To facilitate geometric computation, `ePiX`'s `Circle`, `Plane`, `Segment`, and `Sphere` classes can be intersected with the `*` operator. Table 3.2 lists the return types for each pair of arguments. Intersection is commutative, so only the top half of the table is shown. For purposes of intersection, a `Segment` is extended into a line. The sample file `pascal.xp` gives typical applications of objects and intersection.

*	Segment	Circle	Plane	Sphere
Segment	P	Segment	P	Segment
Circle		Segment	Segment	Segment
Plane			Segment	Circle
Sphere				Circle

Table 3.2: Object intersection types.

A `Circle` has a center, radius, and unit normal; a `Plane` has a distinguished point and unit normal; a `Segment` has two endpoints; a `Sphere` has a center and radius. An object is *malformed* if these conditions are not met. The constructors return well-formed objects with two exceptions: `Circle` and `Plane` create malformed objects if called with three collinear points. The operator `*` returns a malformed object if either argument is malformed, or if the operands are disjoint, tangent, or coincident. Malformedness is benign: Calling `draw()` on a malformed object does nothing.

Orthonormal Frames

A `frame` comprises three mutually perpendicular unit vectors. The constructor takes three vectors. The `frame`'s third vector e_3 is positively proportional to `v3`, the second vector e_2 is positively proportional to `v2%v3`, and the first is the cross product, $e_1 = e_2 \times e_3$. Thus, a `frame` is right-handed, and does not depend on `v1`.

The elements of a `frame` are named `sea`, `sky`, and `eye`, just as for the `camera`. A `frame` can be rotated through an arbitrary angle about any of its elements.

```
frame();        // the standard basis {E_1, E_2, E_3}
frame fr(v1, v2, v3); // orthonormalize {v1, v2, v3}
fr.sea();       // the first element of fr, etc.
fr.rot1(theta); // rotate fr through theta about sea, etc.
```

3.6.2 Path-Like Elements

Basic path-like objects are drawn with named commands. Arguments of polygon commands are endpoints/vertices. Except for `line` and `Line`, the following are subject to filling.

```
line(P p1, P p2, [double expand]);
Line(p1, p2);  // draw line through p1, p2 (crop required)
triangle(P p1, P p2, P p3);
rect(P p1, P p2);
quad(P p1, P p2, P p3, P p4); // quadrilateral
circle(ctr=P(0,0,0), rad=1, normal=E_3);
circle(ctr, pt);
circle(pt1, pt2, pt3);
```

The optional `line` argument is an expansion parameter: `line(p1,p2,t);` draws a segment centered at the midpoint of `p1` and `p2`, with length scaled by $2^{t/100}$. (Setting $t = 100$ doubles the length, while $t = -100$ halves the length.) The arguments of `rect()` must lie in a plane parallel to a coordinate plane. The arguments to `circle` commands are the same as for `Circle` constructors.

Quadratic and cubic splines are described by their control points. A list of `P` is drawn as a "natural" spline (the C^2 piecewise cubic curve with vanishing second derivatives at the endpoints); the number of points per cubic segment must be specified. Circular and elliptical arcs are given by center, a basis, angular range, and an optional number of intervals.

```
spline(P p1, P p2, P p3, [int n]);       // quadratic
spline(P p1, P p2, P p3, P p4, [int n]); // cubic
spline(vector<P>, int n);                // natural spline
```

```
arc(P ctr, rad, t_min, t_max);  // parallel to (x1,x2)-plane
ellipse(P ctr, P v1, P v2);     // in plane spanned by v1, v2
ellipse(P ctr, P v1, P v2, t_min, t_max, [int n]);
```

Mathematically, these commands draw parametric paths

$$\begin{aligned}
\text{Spline:}\quad & (1-t)^2 p_1 + 2(1-t)tp_2 + t^2 p_3, && t \in [0,1] \\
\text{Spline:}\quad & (1-t)^3 p_1 + 3(1-t)^2 tp_2 + 3(1-t)t^2 p_3 + t^3 p_4, && t \in [0,1] \\
\text{Arc:}\quad & \mathrm{ctr} + (\cos t)(\mathrm{rad},0,0) + (\sin t)(0,\mathrm{rad},0), && t \in [t_{\min}, t_{\max}] \\
\text{Ellipse:}\quad & \mathrm{ctr} + (\cos t)v_1 + (\sin t)v_2, && t \in [t_{\min}, t_{\max}].
\end{aligned}$$

If parameter bounds are omitted in an `ellipse` command, the entire ellipse is drawn. When the angular range subtends one or more full turns in an `arc` or `ellipse` the curve is subject to filling.

Commands for planar (half-)ellipses remain from `ePiX`'s early days:

```
ellipse(P ctr, P radius);
ellipse_left(P ctr, P radius);
ellipse_right(P ctr, P radius);
ellipse_top(P ctr, P radius);
ellipse_bottom(P ctr, P radius);
```

If `radius` is `P(a,b)`, these commands draw all or half of the ellipse with given center in the (x_1, x_2) plane, axes parallel to the coordinate axes, and axis lengths $2a$ and $2b$.

Two commands are available to mark off right angles or a subtended angle. Each accepts a spatial location and two non-zero directions, and draws a scene element in the plane spanned by the vectors.

```
right_angle(P loc, P v1, P v2, scale=8);
arc_measure(P loc, P v1, P v2, scale=8);
arc_measure(P loc, P v1, P v2, offset, text, align, scale=8);
```

The `right_angle` command does not check its arguments for perpendicularity. The `arc_measure` commands mark the *small* angle subtended by the directions; the label form places a label at the midpoint of the arc. The (optional) `scale` argument is the true size in `pt` of the marker.

Arrows

Line segments, splines, and arcs can be drawn with arrowheads at one end. In profile, an arrowhead's width is `3pt`, and its height is 5.5 times the width. The actual printed height depends on the head's orientation with respect to the camera. By default, an arrowhead is a filled triangle. The shape and size are adjusted with declarations:

```
arrow_width(w=3);   // width in pt
arrow_ratio(r=5.5); // height-to-width
arrow_inset(c=0);   // base indent as frac of ht
```

→ Inset= 0 → Inset= 0.25 → Inset= 0.5

The `inset` must be between −1 and 1. Each `arrow` command accepts an optional `scale` argument, which scales the arrowhead.

```
arrow(P tail, P tip, [scale]);
arrow(P p1, P p2, P p3, [scale]);   // spline arrows
arrow(P p1, P p2, P p3, P p4, [scale]);
arrow(P ctr, P v1, P v2, t_min, t_max, [scale]); // ellipse
```

A few "special-purpose" commands are supplied:

```
dart (P p1, P p2);  // same as arrow(p1, p2, 0.5);
aarrow(P p1, P p2); // double-headed arrow <--->
arc_arrow(ctr, rad, t_min, t_max, [scale]);
```

If an `arc_arrow` is too short, nothing is drawn.

3.6.3 Coordinate Axes and Labels

`ePiX` provides an `axis` class for coordinate axes. Labels are generated automatically in a variety of styles: decimal, scientific notation, fraction, and trigonometric fraction. Logarithmic axes and labels are available. Axis and labeling commands from Version 1.0 have been retained.

The `axis` Class

An `axis` consists of a line segment divided into a specified number of equal-length subintervals by "major" (long) tick marks. Each subinterval may be divided further by "minor" (short) tick marks. Minor ticks may be equally-spaced ("Cartesian") or logarithmically placed. Finally, a label is written at each major tick mark with specified offset and alignment. Labels are generated automatically from the endpoints, so the line should be parallel to a coordinate axis. Label attributes (masking, borders, font size and face, rotation angle) are determined by the current label style, not by the `axis`.

The command

```
axis(P tail, P head, int n, P offset=P(0,0), [align = none]);
```

creates an axis joining `tail` to `head`, divided into `n` segments of equal length, with a major tick mark and label at each division point and endpoint. The `offset` and `align` arguments have the same meaning as for ordinary labels.

The number and length of minor ticks, and the alignment of ticks, are controlled with member functions:

```
axis Ax(P(a,c), P(b,c), n);
Ax.subdivide(n);  // put n-1 minor ticks in each axis segment
Ax.tick_ratio(r); // minor length = r*major length
Ax.align(AL=c);   // align all ticks; AL = t, b, l, r, or c
```

By default, major ticks are `6pt` long and twice the length of minor ticks. The global declaration `tick_size(len)` sets the major length, subject to "reasonable stylistic limits". For visual consistency, tick lengths should not be changed casually.

Labels on an `axis` may be drawn in several styles, selected with member functions:

```
Ax.dec();  // decimals (default)
Ax.frac(); // fractions, e.g. 0.5 -> \frac{1}{2}
Ax.trig(); // fractional multiples of \pi
Ax.sci();  // scientific notation, k\times base^N

Ax.unmark(double); // remove label at selected location
Ax.precision(p);   // set number of digits for decimal labels
Ax.align_labels(AL); // re-align labels
```

Arbitrary textual labels depending on one coordinate are obtained by writing a `string`-valued function of `double` and "registering" it:

```
// f represents x as a string in given precision and base
std::string f(double x, unsigned int prec, unsigned int base);
Ax.label_rep(f);
```

For example, textual tags can be printed instead of numeric labels.

By default, an `axis` is Cartesian. The member function `log(int b=10)` converts an `axis` to "log mode" with specified base; this affects both tick marks and labeling. If `b` is at least 3, minor tick marks appropriate for a logarithmic axis base `b` are drawn. Second, labels are written in decimal or scientific notation appropriately for a logarithmic axis; that is, "$k \times b^N$" (or its decimal value) is written at location $N + \log_b k$.

Minor ticks of a log `axis` may be labeled individually; again, this is controlled with member functions:

```
Ax.tag(d);   // labels at N+log_b(d)
Ax.tag235(); // tag 2, 3, and 5 if b=10
Ax.tags();   // tag 1, ..., b-1
Ax.untag();  // remove all tags, including 1
Ax.untag(double);  // remove one tag, e.g. Ax.untag(9);
```

For convenience, an `axis` object along an edge of the active `screen` can be created with a named command:

```
top_axis(n, offset, align);
bottom_axis(n, offset, align);
left_axis(n, offset, align);
right_axis(n, offset, align);
```

The ticks automatically point into the bounding box.

The functions above create objects or set `axis` attributes, but write no output. Tick marks and labels can be printed separately, or at once:

```
Ax.draw();        // axis, tick marks, and labels
Ax.draw_ticks();  // axis and ticks only
Ax.draw_labels();
```

Other Axis-Drawing Commands

`ePiX` formerly supplied commands for drawing simple axes and their labels. These commands have been kept for compatibility. Horizontal axes are generated with

```
h_axis(p1, p2, n, align=c);    // n subintervals (n+1 ticks)
h_log_axis(p1, p2, n, align=c, base=10);
```

For vertical axes, use `v_axis`. The style of tick mark is appropriate for an axis of the given type. Horizontal axis tick marks may be aligned `t` (above the axis) or `b` (below). Similarly, vertical axis ticks may be aligned `r` or `l`.

The endpoint arguments of a coordinate axis may be omitted, in which case they default to $p_1 = (x_{\min}, 0)$ and $p_2 = (x_{\max}, 0)$ for a horizontal axis, or to $p_1 = (0, y_{\min})$ and $p_2 = (0, y_{\max})$ for a vertical axis. If the bounding box has integer width and/or height, omitting the number of points draws tick marks one unit apart.

Labels for a horizontal Cartesian or logarithmic axis are generated with the commands

```
h_axis_labels(P p1, P p2, int n, P offset, [align]);
h_axis_masklabels(p1, p2, n, offset, [align]);
h_axis_log_labels(p1, p2, [n], offset, [align], base=10);
h_axis_log_masklabels(p1, p2, [n], offset, [align], base=10);
```

Labels for a vertical axis are generated with `v_axis_labels`, etc. The labels are automatically generated to match their horizontal location. The first puts $(n+1)$ evenly-spaced labels on the segment joining `p1` and `p2`. As with ordinary labels, the `offset` is in `pt`, and the optional LaTeX-style alignment option positions the labels using their corners. The second command draws masked labels according to the current label masking attributes. The third writes labels in exponential notation, using the Cartesian coordinate as exponent.

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

Broken Axes

Broken axes are best drawn using page layout, especially if axis labels are to be drawn. To accomplish the task, create a screen for each piece of axis, using appropriate Cartesian coordinates for the corners, then `inset` the screens so the axis pieces are nearly end to end. A zig-zag glyph signifies the break:

```
axis_break(P, P, scale=12);
```

The `P` arguments are the screen coordinates of the ends to be joined; the optional third argument is the true height and width in `pt` of the glyph. The sample file `coord_tricks.xp` uses layout and axis breaks.

Coordinate Grids

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

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

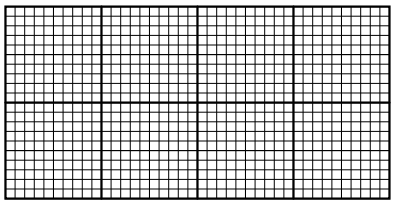
Log and semi-log grids are drawn with analogous syntax:

```
log_grid (p1, p2, n1, n2, [base1], [base2]); // log-log
log1_grid(p1, p2, n1, n2, [base]);           // log-lin
log2_grid(p1, p2, n1, n2, [base]);           // lin-log
```

The n_i arguments dictate the number of squares, namely the number of orders of magnitude spanned in the logarithmic direction(s). The (optional) base arguments default to 10, and control the number of lines drawn per order of magnitude. As with Cartesian grids, the corners are optional; if omitted, the grid fills the bounding box.

Graph paper may be created by superimposing grids:

```
pen(0.25);
grid(10*xsize(), 10*ysize());
pen(0.5);
grid(2*xsize(), 2*ysize());
pen(1);
grid(xsize(), ysize());
```



3.6.4 The Path Class

A `path` data structure is an ordered list of points that can be cropped, clipped, mapped, concatenated, and drawn. The first four constructors mirror polygon-drawing commands above. Function graphs and parametric paths are built from a real- or `P`-valued function `f` of one variable.

```
path(p1, p2, expand=0);     // line (endpoints)
path(p1, p2, p3, [n]);      // quadratic spline
path(p1, p2, p3, p4, [n]);  // cubic spline
path(p1, v1, v2, t_min, t_max, [n]); // ellipse
path(f, t_min, t_max, [n]);
```

The member function `pt()` accepts a `P` or three (or two) doubles and appends the specified point to a `path`. This snippet creates a regular n-gon:

```
path ngon; // declare new path
for (int i=0; i<=n; ++i)
  ngon.pt(cis(i*full_turn()/n)); // works in all angle modes
```

Compound paths may be built by concatenation. If `path1` and `path2` share an endpoint, 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). For expected results, the first (or last) point on `path2` should be the last point of `path1`. The notation suggests 1-dimensional homology chains. The sample file `contour.xp` illustrates path creation and manipulation.

A `path` is a data structure, and must be drawn explicitly to create visible output. By default a `path` is not a closed loop (even if the first and last points are the same), and is not filled when drawn. Member functions perform these tasks. Continuing the n-gon snippet above,

```
ngon.close(); // mark path as closed
ngon.fill();  // draw filled region if filling is active
ngon.draw();  // print to the screen
```

`path::close()` adds a closing edge if necessary. Once a `path` is closed, no more points can be added. A closed path clips and crops differently than an open `path` with the same data. `path::fill()` has no effect on an unclosed `path`.

3.6.5 Function Plotting

The noun "map" refers to a `C++` function that accepts one or more `double` arguments and returns a `double` or a `P`. Mathematically, a map can be depicted in two ways: as a graph (which retains information about the domain), or as a parametrized curve or surface (which discards domain information). `ePiX` assumes that `double`-valued maps are graphed and `P`-valued maps are drawn parametrically. Either sort of depiction is called a "plot". `ePiX` plots are either "wire mesh", produced by a `plot` command, or "shaded", produced by a `surface` command.

Basic Plotting

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.

Domains and Wiremesh Plots

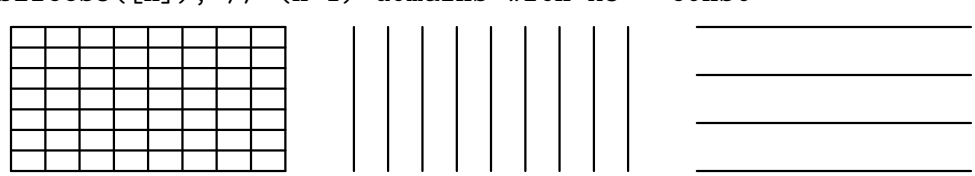
An `ePiX domain` is a coordinate box of dimension one, two, or three, specified by a pair of opposite corners and two *meshes* ("coarse" and "fine", respectively) which specify the amount of data to be plotted. Plotting is explained in detail below.

```
// [a1,a2] x [b1,b2]: n1 x n2 rectangles, m1 x m2 intervals
domain R2(P(a1,b1), P(a2,b2), mesh(n1,n2), [mesh(m1,m2)]);

// [a1,a2] x [b1,b2] x [c1,c2] divided analogously
domain R3(P(a1,b1,c1), P(a2,b2,c2),
          mesh(n1,n2,n3), [mesh(m1,m2,m3)]);
```

If unspecified, the fine mesh is the same as the coarse mesh. For expected behavior, the coarse mesh should "divide" the fine mesh, in that m_i should be a (usually small) integer multiple of n_i for each i.

A `domain` may be *resized* in any coordinate for which the thickness is positive, and can be *sliced* by setting one variable to a constant. The result of slicing is a `domain` whose dimension is one smaller than the original. Finally, "`slices`" operators return the list of `domain`s obtained by setting one variable to evenly-spaced constants. By default, the number of slices is specified by the coarse mesh. An optional argument specifies the number of slices. This argument need not be related to the coarse mesh.

```
R2.resize2(a,b); // [a1,b1] x [a,b]
R2.slice1(t);    // set x1 = t
R3.slices3([n]); // (n+1) domains with x3 = const
```



R R.slices1() R.slices2(3)

When possible, resizing preserves grid square sizes. Generally, though, integer truncation occurs: If `R=domain(P(0,0), P(1,1), mesh(10,6))`, then `R.resize1(0,0.25)` is the rectangle $[0, 0.25] \times [0, 1]$ subdivided into 2×6 subrectangles, since $10 \div 4 = 2$ in integer arithmetic. For expected behavior, choose mesh sizes to avoid integer truncation.

The arguments of a `plot` command are a map, followed by either a domain or its logical equivalent.

```
double f(double t) { return t*t; }
P F(double u, double v) { return P(u, v, exp(u)*Sin(v)); }
P G(double u, double v, double w) { return P(v*w, u*w, u*v); }

plot(f, a, b, n);       // f:[a,b] -> R, using n intervals
plot(F, R2);            // graph of exp(u)*Sin(v)
plot(G, R3.slice2(0.5); // G: R^3 -> R^3 restricted to y=0.5
```

By (compiler-enforced) convention, `plot` commands involving a `P`-valued map accept a `domain` argument, as in the second and third commands above. To plot a `double`-valued function, by contrast, supply the logical equivalent of a `domain`, usually the endpoints and the number of intervals, as in the first `plot` command above.

Resizing and slicing allow a map `F` to be plotted selectively over parts of its domain. This can be used to emphasize parts of the image, layer scene elements, patch surfaces together, and so forth. Resize and slice(s) commands may be used directly in a `plot` command:

```
plot(F, R2.resize1(0,0.5));
plot(F, R2.slices1());
```

Meshes and Plotting

The `P` arguments of a `domain` are a pair of opposite corners. The first `mesh` argument, the *coarse* mesh, specifies the number of subdivisions in each coordinate direction. The second `mesh`, the *fine* mesh, determines the number of points used in each direction when plotting.

Separating the roles of coarse and fine meshes allows a plot to conform closely to a surface without using a fine grid of curves. Both parts of Figure 3.3 are drawn with a 6×20 coarse mesh. In the first picture, the fine mesh is also 6×20, while in the second, the fine mesh is 12×60.

The coarse mesh is significant only for domains of dimension at least 2. The coarse mesh's size determines the number of curves or surfaces plotted *perpendicularly* to a coordinate direction, while the fine mesh's size determines the number of segments used *along* that direction. For predictable results, the fine mesh should be a small multiple of the coarse mesh.

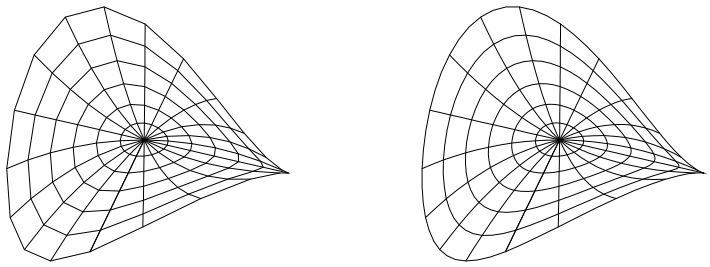


Figure 3.3: Coarse and fine meshes.

Plotting works analogously for 3-dimensional `domain`s and maps depending on three variables: The "one-dimensional skeleton" of the `domain`'s image is drawn. A `P`-valued map of three variables can be plotted over a 1- or 2-dimensional `domain`. (The effect may be unexpected unless the domain arises by slicing, however.) A map depending on one or two variables cannot be plotted over a 3-dimensional `domain`.

Shaded Surfaces

`ePiX`'s shaded surface plotting implements a degree of hidden surface removal. The algorithm breaks a surface into mesh fragments, sorts them in decreasing (approximate) distance to the camera, and prints them. If filling is active, a mesh fragment is shaded according to the angle between the normal vector and the direction to the camera, simulating constant ambient lighting; otherwise, the current fill color is used. This technique works fairly well for surfaces without intersection, and even acceptably handles intersecting surfaces for which mesh elements intersect only along boundaries.

The syntax of a `surface` command is identical to a `plot` command when only a single surface is drawn. For example,

```
surface(F, R, cull=0);
```

plots the `P`-valued function `F` over the domain `R`, the shaded equivalent of the corresponding `plot` command. The optional `cull` argument removes elements that point toward (`cull=-1`) or away from (`cull=1`) the camera. Naturally, orientation of mesh elements depends on the parametrization `F`, not merely on the surface. Culling reduces the output file size, but is useful mostly for closed, convex surfaces.

There are special commands for surfaces of rotation; each accepts a final `cull` argument.

```
surface_rev(f, t_min, t_max, n_lats, n_longs);
surface_rev(f, g, t_min, t_max, n_lats, n_longs=24);
surface_rev(f, g, R, frame coords);
```

The first revolves the graph of f about the x-axis, the second uses the parametric curve $t \mapsto \big(f(t), g(t)\big)$ as profile. In each case, the parameter interval `[t_min,t_max]` is divided into `n_lats` equal-length subintervals, `n_longs` copies of the profile curve are drawn, and the complete surface (one full turn) is drawn.

The third form uses a `domain` to control the range of longitudes, and draws a surface of rotation in the Cartesian coordinate system defined by the orthonormal basis `coords`, by default the standard basis. The arguments `f` and `g` define a parametric curve in the plane spanned by the first two elements of `coords`, and the first element is the axis of rotation.

As in wire mesh plotting, the fine mesh is used to draw the boundaries of surface patches; this tends to make surfaces look smoother for modest-sized coarse meshes. If the coarse mesh is too coarse, however, two visually undesirable effects can occur. First, adjacent regions of the surface may be shaded very differently, since shading is constant over patches defined by the coarse mesh. Second, a patch nearly tangent to a line of sight may be drawn badly if the patch bends back on itself, since the boundary of the *patch* is drawn, not the visible edge of the mathematical surface. See `samples/artifacts.xp`.

Multiple Domains and/or Maps

A scene containing two or more shaded surfaces cannot generally be built up one surface at a time. Instead, multiple surfaces must be assembled into a single data structure before they can be drawn. Multiple surfaces are built from one or more maps and one or more 2-dimensional `domain`s. In the code snippets below, `F` and `G` are `P`-valued functions of 3 variables, and `R` is a 3-dimensional `domain`.

To plot the images of several `domain`s under a *single map*, assemble the `domain`s into a list if necessary, then issue a `surface` command:

```
surface(F, R.slices3(), cull=0);

domain_list DL(R.slice1(0));  // build domain list
DL.add(R.slice2(0.5));        // add a domain, etc.
surface(G, DL, cull=0);       // draw
```

For multiple maps, `ePiX` provides the `scenery` class. Conceptually, `scenery` is an agglomeration of shaded surfaces, built one surface at a time from maps and 2-dimensional `domain`s. The `add` function accepts two arguments—a map, and either a `domain` or a list of `domain`s—and contributes its data to the `scenery` rather than plotting immediately. Completed `scenery` is drawn manually.

```
scenery S(F, R.slice3(0.25)); // S contains one surface
S.add(F, R.slice2(0));        // S contains two surfaces
S.add(G, R.slices1(3));       // S contains six surfaces
S.draw(cull=0);
```

Complete examples are included in the `samples` directory: `spherical.xp` and `minkowski.xp`.

In principle, a scene may contain arbitrarily many surfaces. However, figures that contain many objects tend to tax LaTeX's internal stacks. Frequent color changes exacerbate the problem. Even if you use `hugelatex` (or increase LaTeX's memory), a figure containing more than a few thousand mesh elements is unlikely to compile. At moderate resolution, a surface can easily contain 1000 patches. Each shell script has a command-line option to invoke `hugelatex`; your mileage may vary.

User-Specified Color Shading

By default, a `surface` or `scenery` is colored according to the current fill color. For finer control, each `surface`, `surface_rev`, and `scenery` command accepts an optional position-dependent `color` specification.

```
surface(F, R, color, cull=0);
surface_rev(f, [g], t_min, t_max, n_lats, n_longs, color);
surface_rev(f, g, R, color, [coords]);

scenery S(F, R, color);
S.add(F, R, color);
```

The `color` argument is a `P`-valued function of two or three `double`s whose output is interpreted as a set of `RGB` densities. If `color` takes two arguments, they are `domain` coordinates, and the surface is colored according to parameter values. If `color` takes three arguments, they are Cartesian coordinates, and the surface is colored according to spatial location. Please see the sample files `surface_shade.xp` and `S2_harmonics.xp`.

3.6.6 Calculus Plotting

`ePiX` provides high-level commands for plotting derivatives and definite integrals, Riemann sums, tangent lines, slope- and vector fields, and solutions of planar and spatial systems of differential equations.

In this section, `f` and `g` are `double`-valued functions of one variable.

Utility Functions

```
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, which should be reasonably close to the expected solution. If a critical

point is hit or 5 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.

Derivatives and Integrals

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

```
Deriv df(f); // function object: df(x) = f'(x)
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

Integral prim(f,a); // function object: prim(x) = int_a^x f
prim.eval(b);  // numerical integral of f over [a,b]
double val(Integral(f).eval(1));   // val = \int_0^1 f
```

The lower limit on an integral is 0 by default. `Deriv`s and `Integral`s can be used directly in a `plot` command:

```
plot(Deriv(f), a, b, n);  // plot f' over [a,b]
plot(Integral(f, x0), a, b, n);
riemann_sum(f, a, b, n, TYPE);
```

The second 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 draws rectangles or trapezoids whose area approximates the definite integral of f over $[a, b]$. The `TYPE` may be `UPPER`, `LOWER`, `LEFT`, `RIGHT`, `MIDPT`, or `TRAP`.

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(f, g, t_min, t_max, n); // field of tangents
```

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

Systems of Differential Equations

Let `F` be a `P`-valued function of two or three variables.

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

The first 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 of two variables; see the sample file `dipole.xp`. The `flow` function returns the result of starting at p_0 and flowing by F for time $t_{\max}$, using Euler's method with n time steps. This is useful for placing markers or arrowheads precisely along a flow line.

A planar or spatial vector field itself may be plotted over a domain `R` in three styles:

```
vector_field(F, R, [scale]); //  true length
dart_field  (F, R, [scale]); // const length
slope_field (F, R, [scale]); // const length
```

The field is sampled at the grid points of the coarse mesh. If the domain is 2-dimensional, the plot is a planar slice of the field, even if the field depends on three variables. If the domain is 3-dimensional, the field is drawn in successive slices $x_3 =$ const, starting at the height of the first corner of `R` and ending at the height of the second corner.

The optional final argument, which defaults to 1, scales the arrowheads in a vector field, and scales the (constant) length of field elements for slope and dart fields. The sample files `layout2.xp`, `lorenz.xp`, `slopefield.xp`, and `vfield.xp` illustrate usage.

In each field-plotting command, the domain argument may be replaced by two points, representing corners of a coordinate rectangle, and two integers, the number of grid intervals in the selected coordinate directions. Only planar slices of a vector field can be plotted using the alternative syntax.

3.6.7 Non-Euclidean Geometry

`ePiX` provides limited features for spherical and hyperbolic geometry: the ability to draw lines in the half-plane and Poincaré disk models of the hyperbolic plane, and to draw latitudes, longitudes, great circle arcs, spherical triangles, regular polyhedra, and parametrized curves on a sphere.

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.

Spherical arcs and triangles are described by their endpoints. Only the direction vector from the center of the sphere to an endpoint is significant; if a sphere is scaled or moved, the same function call will draw the corresponding object on the new sphere.

The following draw the visible (front) portions of great circle arcs:

```
front_arc(p1, p2, S);  // short arc on S from p1 to p2
front_arc2(p1, p2, S); // arc from p1 to -p1 through p2
front_line(p1, p2, S); // great circle through p1 and p2
```

Triangles and regular (Platonic) polyhedra are provided. The sample file `sample/polyhedra.xp` illustrates usage.

```
front_triangle(p1, p2, p3, S); // spherical triangle
front_tetra(S, coords);  // regular tetrahedron
front_cube(S, coords);   //    hexahedron
front_octa(S, coords);   //    octahedron
front_dodeca(S, coords); //    dodecahedron
front_icosa(S, coords);  //    icosahedron
```

Each function has a `back` version, which draws the hidden portion. The tetrahedron, cube, and octahedron are (up to scale) inscribed in the cube of side length 2 centered at the origin whose sides are parallel to `frame`. The point $(1, 1, 1)$ is a vertex of the tetrahedron.

Up to scale, the icosahedron's vertices lie on the golden rectangle with vertices $(\pm\gamma, 0, \pm 1)$ and its images under cyclic permutation of coordinates. The dodecahedron is dual to the icosahedron.

Spherical Geometry

The `Sline` class represents geodesic arcs on the unit sphere, and provides intersection and reflection operators.

```
Sline L1(tail, head); // non-proportional vectors
L1.pole();            // pole of equator
L1.cosine();          // subtended arc
L1.reflect(arg);      // reflect arg across L1
L1.reflect(L2);       // reflect L2 across L1
L1.collinear(L2);     // test for collinearity
```

```
L1.cuts(L2);          //   or crossing
L1.draw();            // arc
L1.draw_front();      // front arc (same for back)
L1.draw_line();       // draw entire line
L1.line_front();      // front line (same for back)
```

Spherical Plotting

Parametrized paths on a `Sphere S` (by default the unit sphere) can be specified either by radial projection of a `P`-valued curve, or by stereographic projection of a plane curve given as a pair of `double`-valued functions:

```
frontplot_R(phi, t_min, t_max, n, [S]);    // radial
frontplot_N(f1, f2, t_min, t_max, n, [S]); // from north pole
frontplot_S(f1, f2, t_min, t_max, n, [S]); // 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 has a `back` version that prints only the portion of the path invisible 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.

3.6.8 Data Plotting

Thanks in large part to code and ideas from Marcus Hanwell, files of numerical data can be created, manipulated, analyzed, plotted (paths, scatter plots, and histograms), read, and written. The format for a data file is one or more floating-point numbers per line, with the same number of entries per line. Anything that appears on a line after the LaTeX comment character `%` is a comment.

`ePiX` provides two `plot` commands for file data. The first facilitates plotting selected columns; the second simplifies plotting the first two columns with one or both scales logarithmic. Either form can be used to plot selected columns logarithmically.

The general commands read numbers from two or three columns of a specified file, pass them as arguments to a `P`-valued function `F`, and plot the resulting points:

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

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 3.1. The integers i_k specify columns from which to extract data; these default, respectively, to

the first column, second column, and null (a column of zeroes). If the "coordinate system" F is omitted in the first command, it defaults to the Cartesian point constructor. The function F is mandatory in the second form; useful choices include log_log, log_lin, and lin_log, which plot the corresponding coordinate logarithmically.

Data Files

For more elaborate analysis, the data_file class presents an interface to a file as an ordered list of columns. There are two general ways to create a data_file: read in an external file, or generate data (up to three columns) using specified double-valued functions. In the constructors below, each function fi is a double-valued function of one variable.

```
data_file DF("my_data"); // read data from disk file
data_file DF(f1, t_min, t_max, num_pts); // values of f1
data_file DF(f1, f2, t_min, t_max, num_pts);
data_file DF(f1, f2, f3, t_min, t_max, num_pts);
data_file DF(3);  // create empty data_file with 3 columns
DF.read("file1"); // read a disk file
```

Columns of a data_file can be transformed by a user-specified function, averaged, correlated, extracted (for use by other code), scatter plotted, and written to a disk file at specified precision. Below, the function f is a double-valued function of one variable and F is a P-valued function of two or three variables, whose components are written back to the selected columns.

```
DF.transform(f, i);    // apply f to selected column(s)
DF.transform(F, i=1, j=2);
DF.transform(F, i, j, k);
```

Basic statistical operations on columns are provided.

```
DF.dot(i,j);        // dot product of columns i, j
DF.avg(i);          // mean of column i
DF.var(i);          // population variance
DF.covar(i,j);      // covariance
DF.regression(i,j); // plot regression line
```

A data_file is scatter plotted using syntax as described above.

```
DF.plot(STYLE, [i1], [i2], [i3], [F]);
DF.plot(STYLE, F, [i1], [i2], [i3]);
```

Histograms and bar charts are described below.

A data_file can be written to a disk file as raw data, or in specified format. Below, fmt denotes a string-valued formatting function of two variables and myfile is the name of the disk file to be written.

```
DF.precision(4);    // set to 4 significant figures
DF.write("myfile"); // write as tab-separated columns
DF.write("myfile", fmt, [i1], [i2]); // apply fmt to cols
```

A column can be extracted as a `C++ vector` for use by another function.

```
DF.column(i);    // i-th column
DF.column(f, i); // i-th column, transformed by f
```

Data Containers

`ePiX` provides a `data_mask` class for culling data from a file according to the values in a specified column, and a `data_bins` class for sorting data by value.

A `data_mask` consists of an interval of numbers and a "filter" function. The (closed, open, or half-open) interval is given as a string in standard mathematical notation, or by its endpoints (for a closed interval). The filter is a `double`-valued function of `double`, by default the identity, $f(x) = x$.

```
data_mask dm("[a, b]", [f]);
data_mask dm(a, b, [f]);
```

A `data_mask` "passes" inputs x if $f(x)$ lies in the interval. The member function `reverse()` inverts this logical test.

The `data_file` class has `prune` functions to cull rows for which a specified column's entry satisfies a `data_mask`'s criterion.

```
DF.prune(dm, i);   // remove row if i-th column entry fails
DF.prune(a, b, i); // remove row if i-th column outside [a,b]
```

A `data_bins` object models an interval divided at specified locations into "bins", not necessarily of equal length. Numerical data is read in and the number of points in each bin counted. The lifetime of a `data_bins` object has two stages. First, "cuts" (endpoints of subintervals) are added. Then, once data is read, the cuts are "locked" and cannot subsequently be changed.

```
// [xmin, xmax] divided into n equal intervals, 1 by default
data_bins db(xmin, xmax, [n]);
db.cut(x);  // add a cut at x (if x is in bounds)
db.read(vector<double>); // read data, lock bins
```

A `data_bins` object can be plotted as a histogram (rectangles' *area* is proportional to the bin population), bar chart (rectangles' *height* is proportional to the bin population), or spline interpolation of a bar chart.

```
db.histogram(c=1);  // c = vertical scale factor
db.bar_chart(c=1);
db.plot(c=1);
```

By default (`c=1`), the height of a bar chart rectangle is the fraction of the total population contained in the bin; thus, the height is always between 0 and 1. For a histogram, the height of a rectangle is the fraction of the population per horizontal unit in the bin; thus, the total area over an interval $[a, b]$ does not depend on how $[a, b]$ has been subdivided. The sample files `dataplot.xp` and `histogram.xp` illustrate use.

Statistical convention dictates cuts be chosen distinctly from data values; that is, values should all fall strictly within a bin, not at a boundary point. With large, unknown data sets, this convention may be difficult to uphold. `ePiX` attempts to handle anomalous data intelligently, by keeping counts of values "out-of-range" or "on-cut".

In detail, if $x < a$ or $x > b$ is a data value, it is counted as out-of-range and does not contribute to the histogram population. If $x = a$ or $x = b$, the value counts as both out-of-range and on-cut but is added to the population of the lowest or highest bin, respectively. Any other cut appearing as a data input is flagged as on-cut, and increments the population of each adjacent bin by one-half. When a histogram or bar chart is written, `ePiX` prints a warning message summarizing the number of anomalous data seen.

Error Bars

Simple horizontal and vertical error bars are provided. The final (optional) argument is the true height or width (respectively) in pt.

```
h_error_bar(P location, double error, <mark type>, ht=6);
v_error_bar(P location, double error, <mark type>, wd=6);
```

To create more complex elements, such as asymmetric bars, whisker plots, labeled error bars, and the like, write a custom function using true-size drawing, see page 36. For example, a fillable, labeled, asymmetric, vertical rectangular error bar can be implemented (entirely in `ePiX`) like this:

```
void error_bar(P loc, double lo, double hi, P offset,
               std::string text, align, double wd=6)
{
  const double width(pt_to_screen(0.5*wd)); // converts to 3pt
  rect(loc - P(width, lo), loc + P(width, hi));
  line(loc - P(width,  0), loc + P(width,  0));
  label(loc + P(width, 0), offset, text, align);
}
```

For stylistic uniformity, functions such as this should be put into a library and used systematically. Section 4.2 outlines the process of writing, compiling, and using a custom library.

3.6.9 Legends

A `legend` systematically labels different parts of a plot by associating visual "keys" with explanatory text. This tends to be most useful for plots containing several distinct but conceptually related graphs requiring contrast.

Visually, a `legend` is represented as an aligned list of rows, each containing a box (the key), a gap (the label skip), and some text. These rows are printed in a (usually large) masked label. By default, keys are `12pt` squares bordered in black, the label skip is `6pt`, the background is white, and the border is plain black. These attributes are controlled (simultaneously for all items) with member functions. Parameters of type `double` represent lengths in `pt`.

```
legend L;
L.backing(color);          // set background
L.border(color, [double]); // set border color [and width]
L.border(double);          // border width

L.item_border(color, [double]);
L.item_border(double);     // same, for item borders

L.label_skip(double);
L.key_size(double);
```

Legend Keys

There are three types of legend key, representing filled regions, paths, and markers. Fill and path keys get their visual attributes from the current drawing state. A mark key must be told the marker type. Each is created by specifying the item text.

```
L.fill_item(text);
L.path_item(text);
L.mark_item(<mark type>, text);
```

Keys in the printed legend appear in the same order they are created in the input file.

Creating a Legend

Normally, a `legend` is defined near the start of a file, and an item is added at the point in the file where the corresponding object or plot is drawn, so that the item receives the correct attributes. A `legend` is placed into the figure with the `draw` function. The arguments have the same meaning as for ordinary `label`s.

```
L.draw(P location, P offset, align);
```

All three arguments are mandatory. "Global" `legend` settings (border, backing, etc.) may be changed anywhere between the `legend`'s creation and `draw` function. The sample file `legend.xp` revisits the example on page 18, including a trig-labeled axis and a legend. The file `shadeplot.xp` contains filled keys, and `dataplot.xp` contains a `legend` with items of mixed type.

3.7 More About C++

A textbook or similarly detailed reference is essential for serious study of `C` or `C++`. *The C Programming Language*, second edition, by Kernighan and Ritchie [3], is an excellent, manageable resource for the basics of procedural programming. *C++ Primer Plus*, by Stephen Prata [5], clearly lays out the extensive details of `C++`. Marshall Cline's *C++ FAQ Lite* [1] engagingly discusses common points of confusion and furnishes tips on good design and programming.

`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 a LaTeX picture 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.

3.7.1 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* variable holds the memory address of another variable. Pointers are useful for manipulating (possibly large) data structures through "handles" of fixed small size. C++ also provides *references*, which bind an additional name to an existing object and allow the object to be manipulated through this alias. The statements

```
double  x=1;   // ordinary variable definition
double& rx=x;  // bind a reference, signified by &
```

define a variable x having the value 1, and bind a reference variable rx to it. As long as rx exists, it refers to x. If the value of x changes, the value of rx does as well. Conversely, the value of x can be altered by assigning to rx. However, rx is the size of a pointer, regardless of the size of x, so rx can be passed efficiently in a function call. Some applications are discussed on page 69.

3.7.2 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.

3.7.3 Mathematical Functions

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

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

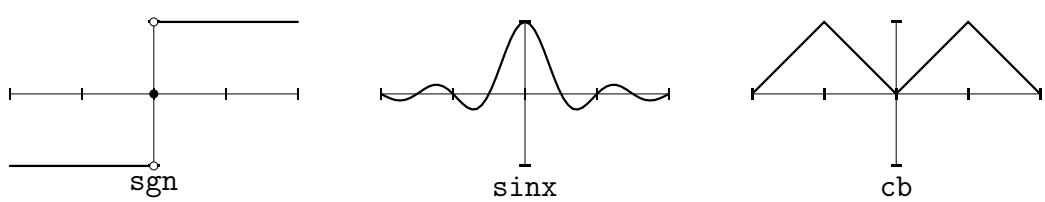
(`fabs` is the absolute value for a floating-point argument.) `ePiX` provides trig and inverse trig functions sensitive to angular mode:

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

The inverse 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:

```
sgn   zero   sinx   cb
```



`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 $[-1, 1]$.

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 [4] 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); }
```

3.7.4 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. Classes allow a programmer to separate an object's *interface* (the set of meaningful operations) from its *implementation* (the data structures and algorithms that realize the interface).

A class implementation consists of *members* (named data elements) and *member functions* (functions that belong to the class and have free access to members). C++ classes enforce access permissions on their members, protecting data from being manipulated except as promised by the interface.

An ideal interface looks like a black box: It hides the implementation completely. In order to cooperate, two classes need only know each other's interfaces. This separation of form and function modularizes a program, and facilitates debugging, code reuse, and overall maintainability, particularly in large programs.

In simple programming, classes may be treated like built-in types. Each class object has its own member functions, whose 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. Generally, a member function call consists of a class object's name, a period, and the name of the member function. Arguments, if any, go in the parentheses after the member function name, just as in a regular function call.

A few short paragraphs cannot do more than scratch the surface of classes and object-oriented programming. For more details, please consult a book, such as Prata [5] or Stroustrup [8], or Cline's on-line FAQ [1].

3.7.5 References and Function Arguments

C and C++ are "call by value" languages. 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 an object from being altered unexpectedly by a function call in a different part of the program. Calling by value helps localize the logic of a program, and circumvents easy-to-write but extremely hard-to-find bugs.

In C++, a function may accept reference arguments. Passing an object by reference grants the calling function access to the object itself, not to a copy. There are two common applications: The object is a large data structure for which copying is "expensive", or the function *needs* to modify its arguments (e.g., a function `swap(x,y)` that exchanges the values of `x` and `y`).

For the first situation, C++ provides the `const` keyword, which ensures the function does not modify its arguments, but accepts a reference merely for efficiency. Any attempt to modify a `const` argument will be caught by the compiler. Most ePiX commands accept `const` reference arguments.

The ability to pass function arguments by reference is sometimes touted as a feature in C++ texts. However, the technique circumvents the data encapsulation of calling by value, and should be avoided unless absolutely necessary. If a function merely "updates" the value of a variable, probably the variable should be of class type, and the update should be performed by a member function.

A function declaration must indicate that its arguments are references. The declarations below have the indicated idiomatic meanings.

```
class matrix;
double det(matrix); // call by value, perhaps inefficient
matrix& transpose(matrix&); // probably changes its argument
double trace(const matrix&); // does not change its argument
```

Unlike pointer arguments, reference arguments impose no syntactic burden on the user. If `A` is a `matrix`, then `transpose(A);` and `trace(A);` will compile. You need not declare explicit reference variables and pass those to the function.

3.7.6 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.

3.7.7 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 which variable names may be re-used without ambiguity. Function bodies are code blocks, as are the alternatives associated to control statements. 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. Variables should be declared in the smallest scope possible.

The compiler is not picky about spaces, tabs, and newlines, so an input file should be formatted to make local scopes visually apparent. `emacs` automatically indents code to reflect scope, though the default behavior does not please all users. As with variable naming, clarity and consistency are paramount.

3.7.8 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 <iostream>
#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. For example, a file might produce either color or monochrome output as follows:

```
#ifdef COLOR
  ...  // code for generating color figure
#endif /* COLOR */
#ifndef COLOR
  ...  // monochrome code
#endif /* undef COLOR */
```

The "compiler symbol" `COLOR` is an ordinary C++ name. To control compilation, either put a `#define COLOR` line in the file, or (better) supply the flag on the command line:

```
epix -DCOLOR <file.xp>
```

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.

3.7.9 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. Preprocessor 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.

3.8 Attribute Quick Reference

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.

A len argument is a double-quoted string containing a number and a two-letter LaTeX length unit, such as "1.5pt" or "6cm". A color argument is a named primary (Red(), Cyan(), White(), etc.), a Color specified by densities (RGB(...), CMYK(...), etc.), or a Color object. Using Neutral() as a Color argument generally turns off the corresponding attribute.

- Angular mode: `radians()`, `degrees()`, or `revolutions()`.

  The angular mode affects 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`.

- Fill style: `fill(color)`, `fill(bool)`, `nofill()`.

- Path style:

  - Width: `plain()`, `bold()`, `bbold()`, `pen(len)`.
  - Line style: `line_style(string)`. The argument is a WYSIWYG sequence of dashes, spaces, and periods. `dash_size(double)` and `dot_sep(double)` set the (approximate) length in `pt` of the pattern. The commands `solid()`, `dashed()`, `dotted()` define "standard" defaults for brevity.
  - Path color: Paths can be drawn using *two* pens, one atop the other. When the "base" pen is white (or the background color) and wider than the "line" pen, a path masks parts of the figure it crosses. A 3-D effect may be obtained by making the base pen a darker shade than the "line" pen.

    `pen(color,[len])`, `base(color,[len])`

    Standard widths: `plain(color)`, `bold(color)`, `bbold(color)`

- Text attributes:

  - Color: `label_color(color)`
  - Mask: `label_mask(color)`, `label_pad(len)`
  - Border: `label_border(color, [len])`, `label_border(len)`. The command `no_label_border()` turns off label borders.
  - Font size: `font_size(LaTeX size)`, no argument means `normalsize`.
  - Font face: `font_face(LaTeX font)`, two-letter font selection string, default is `rm`.
  - Rotation: `label_angle(double)`

Do not confuse `Color` constructors with the similarly-named (deprecated) lowercase attribute-setting commands, `rgb(r,g,b)`, `cmyk(c,m,y,k)`, `red(d)`, etc. These commands affect text, paths, and filled regions. For example, the single command `red()` has the same effect as the three commands `pen(Red())`, `fill(Red())`, and `label_color(Red())`.

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. This section describes the data structures defined in the source files `surface.*`.

The basic idea is to create a shaded polygon class that knows its approximate distance to the camera. For computational simplicity, a mesh "facet" is treated as a quadrilateral, located at the arithmetic mean of its vertices. A facet's boundary is created from a map and a domain by tracing a fine mesh rectangle counterclockwise.

To draw a parametrized surface, facets are stored in a `C++` vector, sorted in decreasing order of distance to the camera, and printed to the output file. If filling is active, the gray density of a facet depends on the cosine of the angle between the normal vector and the vector from the camera to the element.

This simple algorithm works surprisingly well when mesh elements intersect at most along complete edges. To incorporate line-like elements (e.g., coordinate axes, wire-mesh plots) with shaded surfaces, the best technique is often to order high-level scene elements manually, breaking up shaded surfaces (for example, with domain resizing or clipping) as necessary. The sample file `saddle.xp` illustrates possible techniques.

Shaded surfaces can be decorated with a bit of hackery. For example, the `facet::draw` function in `facet.cc` can be modified easily to draw line elements, tangents, or normal vectors along with the facet itself. The sample file `decorate.xp` contains a couple of ideas. (The decorations are activated by compiler flags; please

consult the file itself for information on compiling.)

4.2 Extensions

Thanks to a suggestion of Andrew Sterian, `ePiX` is extensible. User extensions span a spectrum, from header files that require only basic knowledge of `C++` to separately compiled libraries that add substantial new features. The structure of the source code is outlined in Section 4.3.

4.2.1 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() { pen(1.6); }
inline void purple() { 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 pow(x,3); // x^3 }
```

The keyword `void` signifies a function that does not return a value, or (when used as an implicit parameter) a function that does not accept arguments. Inline function definitions are syntactically identical to 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));
```

4.2.2 Compiling

The next few sections outline the creation of a "static library" on GNU/Linux, and explain how to incorporate custom features at runtime. The extensively-commented sample files `std_F.cc` and `std_F.h` illustrate the techniques described below, and may be used for guidance and experimentation.

A small library is usually written as a *header* file, which contains class and 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(const P&);
  bool   spacelike(const 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(const P& arg)
  {
    double x(arg.x1()), y(arg.x2()), z(arg.x3()); // extract coords
    return (y-x)*(y+x) + z*z; // -x^2 + y^2 + z^2
  }
  bool spacelike(const 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.

4.2.3 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>
```

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(s).

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.

4.2.4 Using Multiple Versions

The script `epix` links by default against the `C` math library `libm.a` and the `ePiX` library `libepix.a`. The command option `--no-defaults` clears the header and include paths and removes `libepix.a` from the link list. The script may therefore be used with multiple versions of `ePiX`, a potentially useful feature if you regularly need to compile old source files, or simply prefer the syntax of an older version.

To install and use (say) Version 1.0.0, build the package according to its `INSTALL` instructions, but *do not use the makefile to install*. Instead, manually install the header and library only, using their version number:

```
# install -m 644 epix.h /usr/local/include/epix-1.0.h
# install -m 644 libepix.a /usr/local/lib/libepix-1.0.a
```

A non-system directory may be used instead of `/usr/local`. To use the old version, a source file must `include` the appropriate header file (which is identified by its version number). To compile, issue a command such as

```
epix --no-defaults -I/usr/local -L/usr/local -lepix-1.0 file.xp
```

4.3 Programmer's Guide

This section briefly surveys `ePiX`'s implementation, and is intended for (potential) programmers. The source code is divided into modules with small, well-defined responsibilities, but the user interface is mostly compatible with the syntax of Version 1.0. These constraints demanded a degree of implementation hiding. For example, the user-visible classes defined in `Color.h`, `path.h`, and `screen.h` contain only a pointer to the implementation class, and style data is hidden behind global commands.

Functionally, the code consists of the user interface; implementation classes comprising drawing attributes, spatial objects, screens and representations of their elements, and output; and miscellaneous utility functions. The headers in each group, and their contents, are described in Section 4.3.2.

The user interface headers are assembled into a single file, `epix.h`, and installed in `/usr/local/include` by default. In normal use, the shell scripts read only the user interface header. The individual headers, including the components of `epix.h`, are installed in `/usr/local/include/epix`. These are provided for authors of external libraries, who may need access to implementation details.

4.3.1 External Packages

`ePiX` harnesses the computational power of `C++` to the typographical capabilities of LaTeX. Consequently, `ePiX` should be viewed in part as a framework for expressing numerical data visually.

In the course of your work, you may develop specialized code filling a gap in `ePiX`'s functionality. If your code seems likely to be of interest to other users, please consider bundling it as an external package and notifying the `ePiX` community so your work can be linked from the project pages and distributed to interested users.

There are no formal requirements for external packages, but in the interest of uniformity contributed code should follow the GNU Coding Standards [7]. At a minimum, an external package should build with the standard `./configure; make; make install` commands, and the `configure` script should accept an option `--with-epix` for the user to specify a non-default `ePiX` install directory.

If an external package builds a static library, it should provide a single header containing all the package's entry points, and enclose its interface in a namespace. Entry points should not collide with `ePiX` functions. Naming the package "`epix-<...>`" is a good idea, but not essential. For example, a package providing textual nodes and diagram layout might be named `epix-nodes`. Input files would use the package with the lines

```
#include "epix-nodes.h"
using namespace ePiX-nodes;
```

and be compiled with

```
epix -lepix-nodes <file>
```

The user and internal interfaces of `ePiX`-1.2 are not likely to change. Still, it's prudent to rely only on the user interface in contributed code whenever possible. Doing so also simplifies your work as an author; your library can simply `include` the user header file, and deal only with high-level objects and drawing attributes.

4.3.2 User Interface

These files (in order) comprise the global header `epix.h`.

`enums.h` Marker, alignment, Riemann integral, and vector field types.

`length.h` Physical lengths, conceptually a number and two-letter LaTeX length unit.

`interval.h` Closed, open, and half-open interval ranges for data culling.

`triples.h` The `P` class.

`Complex.h` The `Complex` class.

`functions.h` Angle-sensitive trig functions, miscellaneous utility functions, the `Deriv` and `Integral` classes.

`pairs.h` Screen locations and displacements, with complex arithmetic operations.

`Color.h` The Color class interface, named primaries and constructors.

`state.h` Angle mode; clipping and cropping; label styles; filling; arrow head style; dot and tick sizes, dash length; line style; line and base pen attributes; color-setting commands.

`frame.h` Orthonormal bases.

`domain.h` Coordinate boxes for function plotting.

`camera.h` The camera.

`screen.h` The screen class.

`picture.h` Dimension-setting, offset, layout, decoration, verbatim text, and output format commands.

`markers.h` Point markers, axis labels, and coordinate axes.

`axis.h` Coordinate axes and labels in various styles.

`legend.h` Plot legends.

`path.h` The path class.

`curves.h` Polygons, arrows, ellipses, arcs, splines, coordinate grids, and recursive fractal curves.

`circle.h` The `Circle` shape object class.

`plane.h` The `Plane` shape object class.

`segment.h` The `Segment` shape object class.

`sphere.h` The `Sphere` shape object class.

`intersections.h` Shape object intersection operators.

`plots.h` Plotting commands.

`surface.h` Shaded surface plots.

`data_mask.h` Helper class for data pruning.

`data_file.h` Class for storing and representing data.

`data_bins.h` Class for sorting and counting data.

`geometry.h` Latitudes and longitudes; spherical plotting, arcs, polygons, and polyhedra; hyperbolic arcs.

`Sline.h` The `Sline` (spherical line) class.

4.3.3 Implementation Classes

`Color_Base.h` The Color implementation interface.

`Color_CMY.h` The CMY color model.

`Color_CMYK.h` The CMYK color model.

`Color_Gray.h` The Gray color model.

`Color_Neutral.h` Each color class has a "Neutral" member that converts colors to that model by filtering. This file defines the unique "model-less" Neutral color for which filtering performs no action.

`Color_RGB.h` The RGB color model.

`Color_Sep.h` Classes for CMYK separation.

`active_screen.h` Simple manipulator for the active screen.

`picture_data.h` Picture implementation: two `screen`s (representing the `canvas` and the output page), true dimensions and offsets, pointer to output format, list of colors, and lists of verbatim text to write before and after printing the `picture` environment in the output file. For simplicity, the `screen`s and dimensions are public; encapsulation from the user results from "hiding" this header.

Style Attributes

Declaration-style attributes are maintained with functions returning static references: `the_angle_style()`, `the_arrowhead_style()`, `the_label_style()`, `the_mark_size()`, `the_paint_style()`, and `the_path_style()`. Each function is declared in the analogously-named header.

`angle_units.h` Angular modes: `radians`, `revolutions`, and `degrees`.

`arrow_style.h` Arrow head style data: width, ratio, and inset.

`label_style.h` Text object style: Label and mask colors, padding, border color and width, alignment, font size, font face, and angle.

`marker_style.h` Dot and tick sizes.

`paint_style.h` Line and base pens, fill color.

`path_style.h` Solid, dashed, dotted lines.

`pen_data.h` The pen class.

Objects

`arrow_data.h` Arrow representation.

`facet.h` Shaded surface elements.

`label_data.h` Text (label and marker) objects.

`legend_item.h` Items for `legend`s.

`path_data.h` Path implementation.

`spline.h` The natural spline class.

`spline_data.h` Templates for quadratic and cubic splines.

Screen Representation

Elements in a `screen` are represented polymorphically as "`tile`s", of which there are six types: `glyph` (textual elements), `pen_arrow` (arrows), `pen_fill` (filled regions), `pen_line` (path-like elements), `verbatim` (raw text), and `legend_tile` (legends).

Border and background shape are dictated by the `screen_mask` class. The cropping algorithm assumes the contour of a screen mask is convex.

`affine.h` Affine maps.

`cropping.h` Screen mask cropping.

`glyph.h` Markers and labels.

`legend_tile.h` Screen representation of a `legend`.

`mask_diamond.h` Screen diamond mask.

`mask_ellipse.h` Screen elliptical mask.

`mask_rectangle.h` Screen rectangular mask.

`pen_arrow.h` Arrows.

`pen_fill.h` Filled regions.

`pen_line.h` Paths.

`screen_data.h` Screen implementation class.

`screen_mask.h` Screen mask interface.

`tile.h` Screen element interface.

`verbatim.h` Text in output stream.

Output

Output is divided into a couple of high-level operations and several "atomic" low-level operations. To create a new output format, one need only implement the `format` interface for the desired file type. Paths and filled regions may be implemented however the output type dictates.

`eepic.h` eepic macros.

`fmt_template.h` "skeleton" header for new output formats.

`format.h` The output interface.

`pst.h` PSTricks macros.

`tikz.h` tikz macros.

Utilities

`Color_Utils.h` Functions for setting color channel densities.

`clipping.h` The clip box.

`constants.h` Global constants: Line widths; dot, tick, and dash sizes; arrowhead parameters; miscellaneous numerical constants and internal parameters.

`crop_algorithms.h` Path and loop clipping templates.

`deriv.h` Finite difference template.

`edge_data.h` Path element representation template.

`errors.h` Warning and error messages.

`frac.h` Rational numbers and operators.

`halfspace.h` Halfspace cutting.

`hatching.h` Filling regions in eepic.

`lens.h` Camera lenses.

`map.h` Wrappers for templated plotting.

`plot_algorithms.h` Plotting templates.

`screen_crop.h` Crop paths and loops.

`utils.h` Truncation, date and time, line breaking.

Appendix A

Software Freedom

Academics in general, and mathematicians in particular, depend on free exchange of information. We prove theorems or establish experimental results, write up formal accounts, place preprints on public file servers, and submit papers to peer-reviewed journals. If accepted, the results—data, techniques, methods of reasoning, citations, and conclusions—are published in print and become part of the public record, governed by copyright law. Libraries purchase journal subscriptions, but researchers and scholars may use ideas from the literature merely by giving appropriate citations in their own work. "Theft" arises from false claims of authorship.

Carried over to software, the academic process would guarantee rights similar to those provided by the GNU General Public License (GPL):

- (GPL 0) To run a program for any purpose.
- (GPL 1) To study how the program works, and adapt it to your needs.
- (GPL 2) To redistribute copies of the program.
- (GPL 3) To improve the program, and release improvements to the public, so that the whole community benefits.

In reality, attitudes toward software differ markedly. Most academics work on a proprietary platform, use proprietary software for research and teaching, and share information with colleagues and students in proprietary, even obfuscated, data formats. Contrary to the academic ethic, proprietary software licenses restrict access to information: preventing users from learning how a program works internally ("reverse engineering"), limiting the number of users who may run a piece of software, and forbidding users from running (or sometimes even installing) a purchased copy on multiple machines.

Restrictions on use aside, if one cannot examine a program's source code, one cannot fully trust the output, any more than one can trust (for purposes of scientific publication) results of a commercial testing lab.

Consider a hypothetical future world in which scholarly results are disseminated like software. Instead of subscriptions, journals sell licenses granting readership to a specified number of individuals. Photocopying an article for a class or research seminar constitutes "piracy", though if the institution has purchased a sufficiently large site license the teacher or speaker may bring the physical volume to class and project the pages onto a screen.

Of course, reading an article is scarcely enlightening. Mathematics papers contain only the statements of theorems. Merely opening the journal binds the reader to a lengthy legal agreement, stating that theorems be used only for specific purposes and threatening serious legal consequences for attempting to discover the author's proofs.

The actions of a single student, employee, or faculty member can expose an institution to a costly "journal audit" from the Mathematical Society of America (MSA), with the institution responsible for legal costs if the audit reveals license violations anywhere in the organization.

Mathematicians who long for the Old Days when papers contained proofs and were shared freely are dismissed as idealistic cranks or labeled anti-business communists. Common knowledge asserts the obvious superiority of proprietary journals, and the necessity of licenses for keeping mathematicians gainfully employed.

Back in our world, some vendors have attempted to placate opponents of closed source with "shared source" licenses, under which one may sign a non-disclosure agreement and subsequently examine source code. In the future world analogy, a shared source agreement would allow journal licensees to sign an NDA, then see the proofs of theorems. Readers could thereby correct errors in proofs (benefitting the publisher by improving the reliability of the journal), but would be legally forbidden from using the ideas elsewhere (denying benefit to other mathematicians).

Legally and conceptually codifying software as a commodity ignores a fundamental reality: Like an idea or recipe, software can be copied without loss of the original. The perception of "theft" by copying arises from an artificial belief that software has an owner who must be monetarily compensated each time a person acquires a copy. The nature of software does not enforce the "sale" model in the way services and physical commodities do. It therefore seems philosophically inappropriate to treat software as a commodity, and perilous to conform the legal system to the enforcement of such a model.

At its best, software enhances our productivity and creativity. Sharing software, like sharing ideas, benefits a larger number of people without detriment to existing users. I hope 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 `www.fsf.org`, to learn more about

free software and how you can contribute to its development and adoption.

Appendix B

Acknowledgments

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

Infrastructure Donald Knuth, Conrad Kwok, Leslie Lamport, Tim Morgan, Piet van Oostrum, Sunil Podar, Richard Stallman, Till Tantau, Herbert Voß, Timothy van Zandt

Enhancements Jay Belanger, Robin Blume-Kohout, Julian Gilbey, Marcus Hanwell, Yvon Henel, Svend Daugård Pedersen, Andrew Sterian

Porting and packaging Younès Hafri (Crux), Julian Gilbey (Debian); Tsuguru Kato (FreeBSD); Markus Dittrich, Danny van Dyk, Christian Faulhammer, Olivier Fisette, Chris Gianelloni, Michael Hanselmann, Marcus Hanwell, David Holm, Peter Johanson, Patrick Kursawe, Tobias Scherbaum, Markus Ullmann (Gentoo); Guido Gonzato (RPM); Rene Rebe (T2)

Debugging, advice, and other assistance Maik Beckmann, Jay Belanger, Felipe Paulo Guazzi Bergo, Karl Berry, Robin Blume-Kohout, Aran Clauson, Patrick Cousot, Stephen Gibson, Julian Gilbey, Dov Grobgeld, Bob Grover, Jim Hefferon, Jacques L'helgoualc'h, Yvon Henel, Hartmut Henkel, Herng-Jeng Jou, Walter Kehowski, Paul Kornman, Kevin McCormick, Ross Moore, Mike Protts, Thorsten Riess, Jean-Michel Sarlat, Alan Sill, Neel Smith, Michael Somos, Andrew Sterian, Ryszard Tanas, Ben Tillman, Kai Trukenmueller, Torbjorn Vik, Wenguang Wang, Gabe Weaver, Mariusz Wodzicki

Bibliography

[1] Marshall Cline, *C++ FAQ Lite*, `http://www.parashift.com/c++-faq-lite/`

[2] Uwe Kern, *Extending LaTeX's color facilities: the* `xcolor` *package*, white paper, Jan. 21, 2007

[3] Brian Kernighan and Dennis Ritchie, *The C Programming Language*, Second Ed., Prentice-Hall Software Series, 1988

[4] Sandra Loosemore, Richard M. Stallman, et. al., *The GNU C Library Reference Manual*, GNU Press, 2004

[5] Stephen Prata, *C++ Primer Plus*, Sams, 2002

[6] Keith Reckdahl, *Using Imported Graphics in LaTeX2e*, Version 2.0, white paper, Dec. 15, 1997

[7] Richard M. Stallman, et. al., *The GNU Coding Standards*, `http://www.gnu.org/prep/standards/`

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

[9] Timothy van Zandt, *PSTricks: PostScript Macros for Generic TeX*, Version 0.93a, white paper, Mar. 12, 1993

Index

Affine map, 37
    action on labels, 41
Angle
    mode, 43, 68
    units, 31, 41
Animation, 23–24
Arrow, 47
Aspect ratio, 29
Axes, 48–51
    broken, 51
    labeling, 48, 49
    logarithmic, 50

Backslash character, 22
Bar Charts, 63
Bounding box, 35

Camera, *32–34*, 75
    color separation, 33
    lens, 33
    manipulation, 33–34
    viewpoint, 32
canvas, 13
Circle, 44
Classes, 69
Clipping, 34
Color, 27, 30–42
    blencing, 31
    declaration, 42
    density, 30
    primitive, 30
    separation, 33
    surface plotting, 57
    transparency, 30
Complex number, 44
Conditional statement, 66
`C++`, 21–23, *66–72*
Cropping, 36

Data plotting, 61–64
Domain, 53–55
Drawing style, 38–40

`emacs`, 9, 12, 21, 24
Error bars, 64

Filling, 38
Fonts, 41
Free software, 7, 85–87
Function, 66–70
    call syntax, 69
    class member, 69
    complex, 44
    Euclidean algorithm, 23
    mathematical, 68
    overloaded, 70
    returning `void`, 67

Graph paper, 51
Graphical interface, 14

Header file listing, 80–83
Hidden object removal, 75–76
Histograms, 63

Input file
    comment in, 72
    conditional compilation, 71
Installation, 8–10
Intersection, 45

Label, *40–42*, 72

alignment, 40
axis, 48–50
backslash in, 42
double quotes, 42
fonts in, 41
masked, 41
offset, 42
rotated, 41
Layout, 25, 34
Legends, 65
Length, 27, 29
Line style, 38–40

Mac OS X, 8
Marker, 40
types of, 42

Output file
writing directly to, 23
Output format, 28

Path, 38–40
class, 51
filled, 38
style, 39
Picture
aspect ratio, 29
bounding box, 29
offset, 29
size, 29
Plane, 45
Plotting, 52–59
calculus, 57–59
data, 61–64
spherical, 59–61
surface, 55–57
Point, 43
Pointer, 67
Preamble, 27

Screen, 32–38
active, 34
Segment, 45
Self-contained figure, 23
Sline, 60
Sphere, 45
Stereograms, 25
string, 22

Variable
local, 70
names, 66
pointer, 67
reference, 69
type of, 66
Viewpoint, 32

Windows operating system, 9
Writing directly to output file, 23