Graphics

ePiX: A utility for creating mathematically accurate figures

Andrew D. Hwang

1 Introduction

Mathematical and scientific writing call for figures that accurately and attractively integrate typography and numerical data. Widely-used commercial and non-commercial drawing programs exist, as do dozens of lesser-known utilities. This article describes an addition to the list: ePiX, a collection of command line utilities for creating mathematically accurate, logically structured, camera-quality 2- and 3-dimensional figures and animations in LaTeX. Despite superficial similarities with existing programs, ePiX fills a distinct niche in the ecosystem of drawing software by providing a bridge between the powerful numerical capabilities of C++ and the high-quality typesetting of LaTeX.

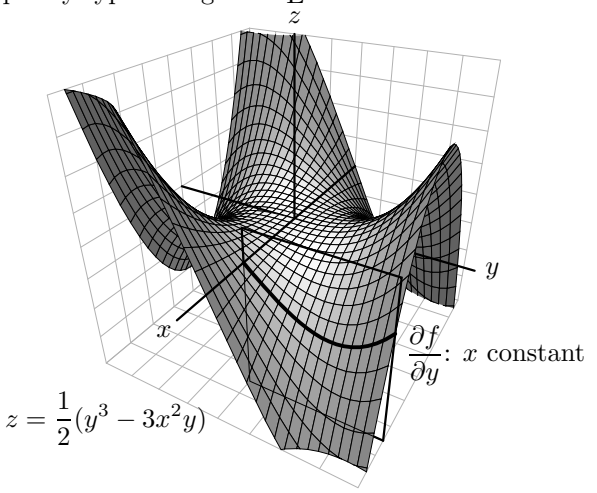

Figure 1: A surface with simulated transparency.

ePiX's relationship to a graphical drawing program is analogous to LaTeX's relationship to a word processor. A logically structured input file is prepared in a text editor, then compiled into a plain text (eepic) file that is included into a LaTeX document. Optionally, the figure may be processed into eps or pdf. This note focuses on the user interface, though certain issues of implementation arise of necessity.

ePiX's strengths include:

- Ease of use: Figure objects are specified by simple, mnemonic commands that refer to a natural coordinate system.

- Quality of output: ePiX creates mathematically accurate line figures whose appearance matches that of LaTeX. Typography is added to an ePiX figure as easily as to a LaTeX picture environment. The mechanism for text placement is robust under changes of scale.
- Wide availability: ePiX runs on platforms with a C++ compiler and the GNU shell bash, particularly on GNU/Linux, Mac OS X, Windows (Cygwin), FreeBSD, and Solaris. An output file may be incorporated into a document on any platform that supports LaTeX.
- Programming: ePiX's input is a widely-spoken, easily-learned programming language. Even simple figures can benefit from logical structuring, while complex figures may employ algorithms and generate their own numerical data.
- Extendability: Users can write custom code and incorporate the functionality with a command line switch or a Makefile. This feature, suggested by Andrew Sterian, endows ePiX with the computational power of C++.
- Economy of storage and transmission: A compressed tar file of the LaTeX sources and compiled eepic files is typically a small fraction of the size of a compressed PostScript file or a tarball containing eps files, making ePiX output particularly attractive for archiving.
- License: ePiX is *Free Software*, published under the GNU General Public License.

This note focuses on general issues of image creation and ePiX's approach to integrating numerical and algorithmic capabilities with high-quality typography. The project home page has source code, documentation, sample images, and animations:

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

The latest stable version is also available from CTAN (in graphics/epix). Please visit the project page for a more thorough showcase of ePiX's capabilities.

I am grateful to Jay Belanger, Robin Blume-Kohout, Andrew Sterian, and Gabe Weaver for detailed and insightful design discussions and advice.

2 Source and Output Files

In LaTeX, a document preamble specifies the default appearance and sets up an environment by including packages and defining macros, while the body contains commands that generate the actual output. Similarly, an ePiX preamble (Figure 2) accesses library code and defines symbolic constants and functions that reflect the internal structure of the figure,

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

// function definition
double f(double x) { return x/(1-x*x); }

int main()
{
  unitlength(".85in");  // LaTeX unitlength
  picture(P(3, 1.5)); // printed size

  // specify corners; depict [-2,4] x [-4,4]
  bounding_box(P(-2,-4), P(4,4));

  begin();            // picture starts here
  crop();             // crop to bounding_box

  dashed();           // draw dashed lines
  line(P(-1, y_min), P(-1, y_max));
  line(P( 1, y_min), P( 1, y_max));

  solid();            // use solid lines
  h_axis(P(x_min, 0), P(x_max, 0), x_size);
  v_axis(P(0, y_min), P(0, y_max), y_size);

  h_axis_labels(P(x_min, 0), P(x_max, 0),
                0.5*x_size, P(-2,2), tl);
  bold();   // draw in bold (fonts unaffected)
  plot(f, x_min, x_max, 120);     // function plot

  label(P(2,3), P(0,0),
        "$y=\\displaystyle\\frac{x}{1-x^2}$");
  end();
}
```

Figure 2: An ePiX source file, cf. Figure 3.

while the body contains commands that adjust the appearance of objects and write the output file.

Body commands include objects, labels, and attribute declarations. ePiX supplies standard geometric primitives: points, lines, circles, spheres, planes, quadratic and cubic splines, ellipses and arcs, arrows, polygons and polylines, and coordinate grids. In addition, ePiX provides plotting: graphs, parametric curves and surfaces, data from files, vector fields, derivatives and integrals, and solutions of ordinary differential equations. Basic geometric objects can be constructed and used in mathematically natural ways, such as finding the intersection point of two lines, constructing a circle through three noncollinear points, or drawing the tangent line to a function graph.

Four internally documented shell scripts constitute the user interface: epix (creates eepic files), elaps (converts ePiX and eepic files to eps, pdf, or PostScript), flix (creates png images and mng animations), and laps (converts LaTeX to PostScript).

3 Design

The notion of "ideal" drawing software is too dependent on authors' individual needs and preferences to

be meaningful. Nonetheless, commonly useful features can be identified. ePiX does not satisfy all the criteria below, but its development has proceeded with these goals in mind.

3.1 Capabilities

A general-purpose command-driven drawing utility provides three basic services: an input language, a set of data structures for representing figure objects and their attributes, and output routines. The input language should be easy to learn and use, yet powerful, flexible, and extendable. Frequently-encountered objects and algorithms should be represented natively, allowing users to program (when necessary) in a high-level language. Both 2- and 3-dimensional figures should be supported. A variety of output file types should be available, so that the resulting images can be exchanged easily, used in printed documents, or published on the Web.

Less technical but equally important are issues of convenience and freedom. A program should supply sensible defaults, so that simple figures can be drawn without micro-management. At the same time, figure attributes should be modifiable with short, easily-remembered commands. Users' files should compile quickly, preferably in no more than a couple of seconds on a moderately fast machine. Output files should be small, perhaps tens of KB, yet of high typographical quality. The program should be widely available, and free from proprietary algorithms and file formats.

3.2 Logical Structuring and Input

Mathematical figures represent structured information. Bitmapped images, and to a lesser extent eps files, discard this structure. By contrast, a programming language exploits logical structure through use of symbolic constants, data structures, functional relationships, and algorithms, including control statements, loops, and recursion. A high-level figure description language is potentially both efficient and convenient, for the same reasons that a Taylor polynomial compactly encodes a trig table. Naturally, users do not want to learn a new language in order to create figures, but software can accommodate users by providing intuitively-named functions that implement common figure objects. Ultimately, however, a language that provides plotting and other algorithmic and numerical capabilities must utilize more complex syntax. To ease the learning curve, a scene description language might piggyback itself onto a widely-used programming language, such as C++, Fortran, or Lisp.

ePiX attempts to meet these goals by furnishing a user-friendly interface to C++, harnessing its speed, flexibility, and computational power to the creation of mathematical figures. An ePiX source file is a compact, high-level scene description written in C++. Even moderately complicated figures require no prior knowledge of C++, and the source code comes with dozens of samples files suitable for study and experimentation.

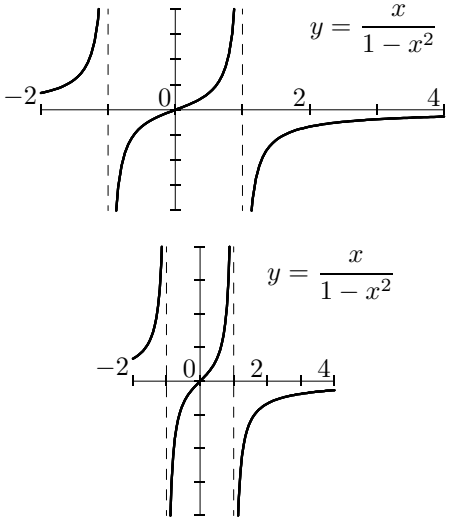

Figure 3: Rescaling: Two figures generated from the input file in Figure 2.

3.3 Page Coordinates and Resizing

Logical markup is fundamental to LaTeX: a document does not directly specify its visual appearance, but relies on packages loaded at compile time. Mathematical figures benefit similarly from logical structuring. Designing and writing a figure in page coordinates, as in the LaTeX picture environment, is conceptually WYSIWYG.

Except as required to size and place the finished product, and to align text (below), an ePiX figure refers exclusively to Cartesian coordinates. The use of logical coordinates makes the input file easier for a human to read, and enhances flexibility: software can render a figure according to user-specified criteria at compile time, changing the size, aspect ratio, or viewpoint, for example.

Incorporation of typography imposes an additional requirement on a figure's coordinate system. A text box is attached to a specific logical location in a figure. However, fonts do not (and usually should not) scale when the size of a figure changes. Consequently, a LaTeX box cannot always be placed using only its basepoint if the result is to compile attractively at various aspect ratios: the Cartesian location of the basepoint does not generally undergo the expected affine scaling when a figure is resized (Figure 3). ePiX handles this difficulty by positioning a

label "coarsely" using Cartesian coordinates, then offsetting it "finely" in true coordinates, namely, aligning the text box on a point other than its LaTeX basepoint. In other words, a scale-invariant alignment point is manually attached to each label, and Cartesian coordinates are used to position this alignment point. There seems to be no simple, high-quality alternative to aligning labels visually and individually.

3.4 Scene Representation

An `ePiX` input file describes a 3-dimensional *world*, which is represented on an abstract 2-dimensional *screen*. World and screen coordinates are Cartesian, and not directly related to the printed figure's size. The screen contains a *bounding box*, a user-specified Cartesian rectangle that is affinely mapped to a LaTeX `picture`. The overall size of the figure is given directly in the input file, while the aspect ratio is determined by the relative aspect ratios of the bounding box and the picture box.

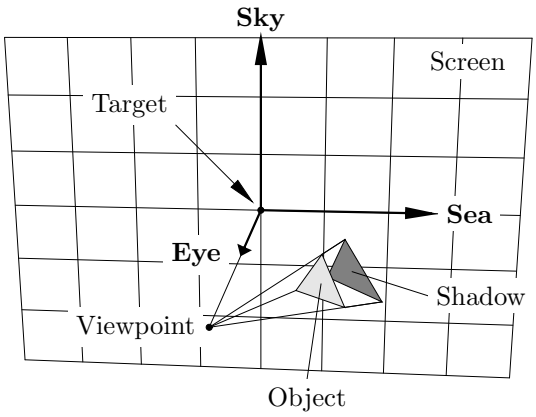

Figure 4: `ePiX`'s point-projection camera model.

The *camera*, consisting of a *body* and a *lens*, maps the world to the screen; indeed, the screen should be regarded as the camera's film plane. The camera body contains information about the location and spatial orientation of an abstract observer, while the lens is the actual mapping, point projection by default (Figure 4). The camera is designed to behave like a real camera: The viewpoint and target may be set arbitrarily, the camera rotated about its axes (sea, sky, and eye), and the lens changed.

To control the abstract and/or printed size of a figure, `ePiX` can remove figure elements that lie outside a user-specified "clip box" (Figure 1), and can "crop" a figure by masking elements that lie outside the bounding box (Figures 3 and 5). Clipping and cropping are disabled by default, in accordance with the design philosophy of imposing minimal default behavior.

3.5 Layering and Hiding

The `eepic` file produced by `ePiX` is at some stage converted to PostScript or PDF. In either case, the output is layered: objects occlude earlier parts of the file. For 2-dimensional black and white line drawings, layering is a minor concern, but for shaded, color, or 3-dimensional pictures, layering is usually important.

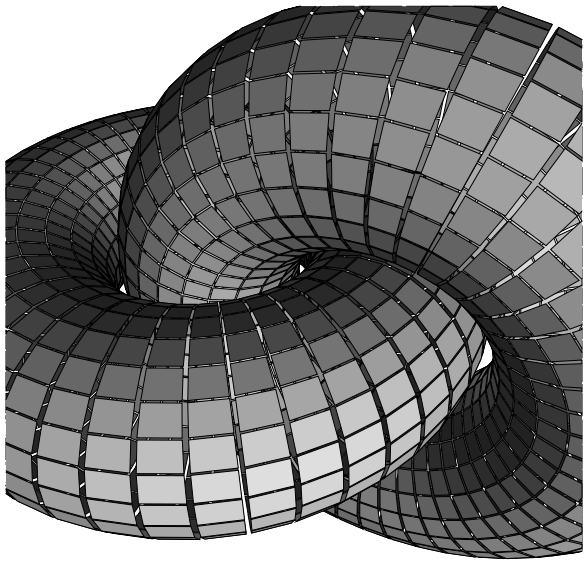

Figure 5: Layering, shading, and cropping.

`ePiX` does not currently automate hidden object removal, but manual techniques provide satisfactory results. In Figures 1 and 5, paths and surfaces are broken into mesh elements, sorted by distance to the viewpoint, and printed to the file in decreasing order of distance. The shading in these figures exemplifies the use of programming constructs in `ePiX`. For each mesh element, a normal vector and illumination vector are calculated, and the shade of gray is a simple function of the angle between these vectors. Similar techniques can be used to simulate multiple light sources, even light sources of varying colors.

3.6 Implementation

Befitting its role as a bridge between the computational power of C++ and the high-quality typography of LaTeX, `ePiX` is not a stand-alone program, but is instead assembled from standard components: the C++ compiler, libraries and binutils; GNU `bash`;

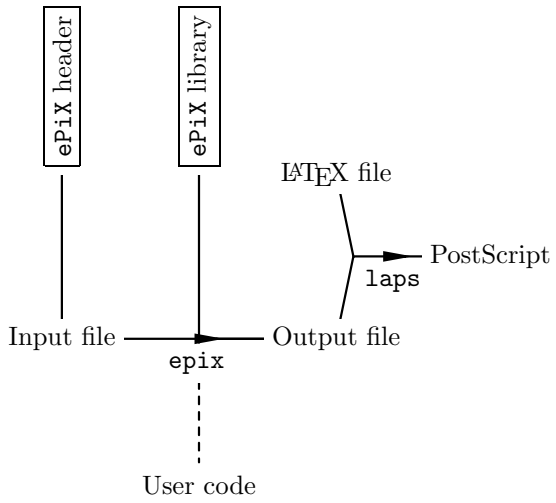

Figure 6: Processing an input file.

and optionally LaTeX, Ghostscript, and ImageMagick. The bulk of ePiX proper consists of a compiled C++ library and header file.

An input file is a short program that incorporates functions from the ePiX library. The shell script epix invokes the compiler on the input file. The resulting binary executable writes the LaTeX code of the figure, which the script directs to a file (Figure 6). Each of ePiX's scripts accepts numerous command-line options, which are listed by running "<script> --help".

From its inception, ePiX has used an external compiler to read and parse input files. This requirement, which may at first seem limiting, is not essentially different from reliance on an interpreter, be it Java, METAPOST, Perl, PostScript, Python, or TeX itself. Further, there are at least three practical reasons for utilizing the C++ compiler.

First, any program processing user-supplied input must recognize and cope with both well-formed and malformed data. The use of an existing compiler avoids both the substantial complication and needless duplication of effort that would result from coding a compiler or interpreter from scratch.

Second, separately compiled code can be incorporated in an ePiX figure with a command-line switch. Use of a widely-spoken language allows users to extend ePiX easily.

Third, when a typical plot is generated, a few functions are called repeatedly, possibly thousands of times. Compiled code runs quickly enough (compared to interpreted code) to justify the time overhead of compiling code to process a figure. When the plot depicts the outcome of a complicated algorithm (such as solving a differential equation), the extra efficiency of compiled code can be substantial.

4 Future Development

Until now, ePiX has existed as a single-developer project, and has grown primarily along lines dictated by a need for features. The current source tree is nearing an evolutionary *cul-de-sac*, and future work will focus on a redesigned and re-implemented version, known informally as The Next Generation. The author welcomes user feedback, design suggestions, and additional coders. The source tree is on the CVS server at savannah.gnu.org.

The Next Generation will separate input, representation, and output, serving as a general-purpose scene description and rendering utility rather than merely a LaTeX-specific image creation tool. However, incorporation of high-quality typography will remain a primary goal. Additional aims of TNG include providing flexible page markup, allowing multiple scenes to be placed in a single figure; more modularized output, so that a single input file can generate a sequence of output files — in various formats — from a single run; and better support for object hiding in 3-dimensional figures.

A framework for high quality scientific drawing and data visualization is of wide interest to the mathematical, scientific, and typesetting communities. It is hoped that ePiX will contribute toward the realization of a GPL-ed utility that is efficient, intuitive, computationally powerful, and sufficiently flexible to grow with its user base for the long-term future.

◇ Andrew D. Hwang
Department of Math and CS
College of the Holy Cross
Worcester, MA 01610-2395, USA
ahwang@mathcs.holycross.edu
http://mathcs.holycross.edu/~ahwang/