1 Overview

ePiX is a lightweight runtime utility for creating mathematically accurate 2-dimensional LaTeX plots and figures from simple, mnemonic commands. A detailed user's manual is available in several formats from

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

This sample document demonstrates what ePiX can do, with side-by-side comparisons of input files and corresponding output.

1.1 Line Figures

ePiX was originally designed for function plotting, but contains provisions for creation and manipulation of "standard" plot objects, including arbitrary triangles, coordinate rectangles, splines, arbitrary elliptical arcs, and hyperbolic lines. The latter are, of course, elliptical arcs, but ePiX allows you to specify circular arcs in several mathematically natural ways.

Figure **??** is very basic, but nonetheless illustrates useful features; particularly, sizes and locations are given symbolically, in terms of the golden ratio, and the arc is specified by center and angle, so the ends are placed precisely.[1]

```
#include <epix.h>

main()
{
  double a = (1+sqrt(5))/2;

  bounding_box(P(0,0), P(a,1));
  picture(P(100*(1+a), 100*a));
  unitlength("0.0125in");

  begin();

  line(P(1,0), P(1,1));
  dashline(P(0.5,0), P(0.5,1));
  line(P(0.5,0), P(1,1));

  arc(P(0.5,0), a-0.5, 0, atan(2));
  boldrect(P(0,0), P(a, 1));

  end();
}
```

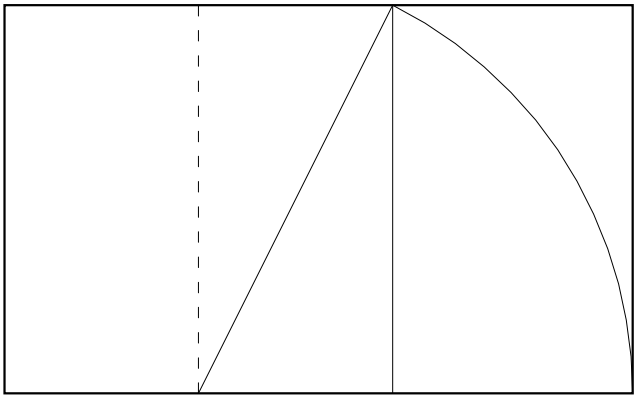
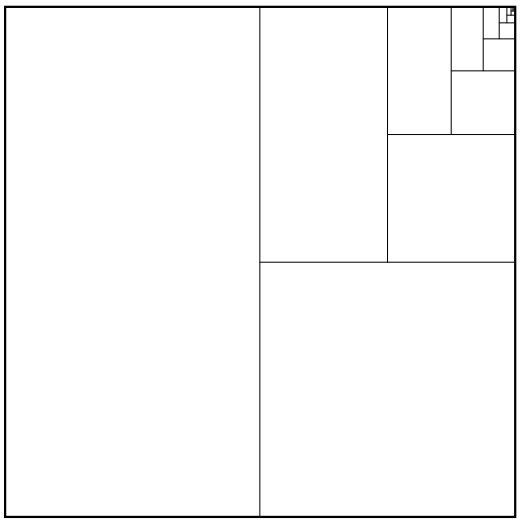


Figure 1: Constructing a golden rectangle.

[1] The sample files distributed with ePiX have `offset` lines to place the figures next to their source code.

With variables and `for` loops, objects can be placed regularly, accurately, and in an easily-modified way. For example, to change the number of rectangles in the figure below, only the definition of `N` needs to be changed. Variables in `ePiX` can be used like macros and user-defined commands in LaTeX, to express a figure in terms of its logical structure.



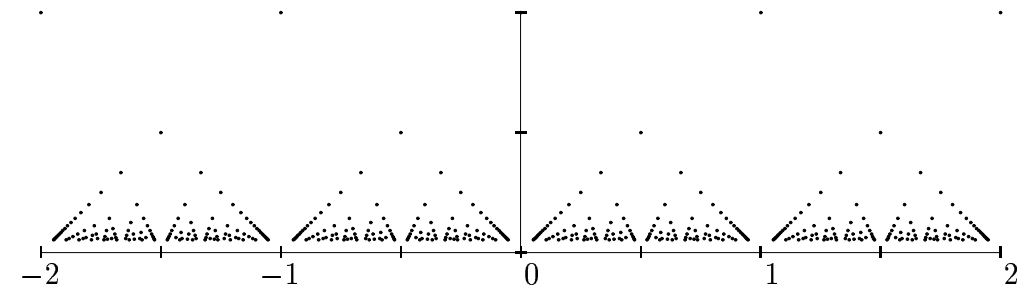
```
#include <epix.h>
#define N 8

main()
{
  double t;

  unitlength("1.5pt");
  bounding_box(P(0,0), P(1,1));
  picture(P(128,128));
  offset(P(96,0));

  begin();

  for(int i=0; i<8; ++i)
    {
      t = pow(0.5, i+1); // 2^-(i+1)
      line(P(1-t, 1-2*t), P(1-t, 1));
      line(P(1-t, 1-t), P(1, 1-t));
    }
  boldrect(P(x_min, y_min), P(x_max, y_max));

  end();
}
```

Figure 2: A geometric series with ratio 1/2.

`ePiX` now uses the `C++` compiler, so variables need not be declared at the start of a file; the loop index is declared in the loop itself. It is a good idea to declare "tweakable" variables in the preamble, but indices can generally be hidden safely.

The "denominator" function f is defined by

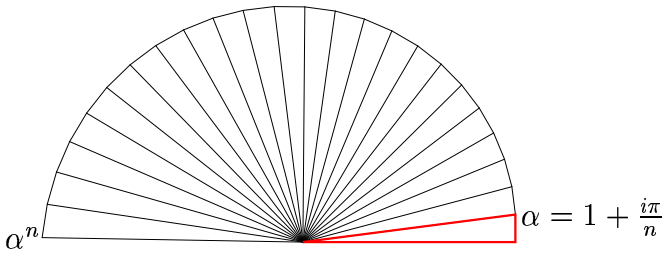
$$f(x) = \begin{cases} \frac{1}{q} & \text{if } x = \frac{p}{q} \text{ in lowest terms} \\ 0 & \text{if } x \text{ is irrational} \end{cases}$$




```
#include <epix.h>

#define N 40

main()
{
  int i, j;

  unitlength("0.0125in");
  bounding_box(P(-2,0), P(2,1));
  picture(P(400,100));

  begin();

  h_axis(P(x_min, 0), P(x_max, 0), 2*x_size);
  v_axis(P(0, 0), P(0, y_max), 2);

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

  for (i=1; i< N; ++i)
    for (j=i*x_min; j <= i*x_max; ++j)
      if (gcd(i, j) == 1)
        ddot(P(( (double) j )/i, 1.0/i));

  end();
}
```

Figure 3: The "denominator" function.

In ePiX, points in the plane are specified by an ordered pair data structure; ePiX provides functions for adding/subtracting, scalar multiplying, and complex multiplying pairs, and for creating pairs in both Cartesian and polar coordinates.




```
#include <epix.h>

main()
{
  int n=24;
  pair pow = e_1; // (1,0)
  pair z = e_1 + (M_PI/n)*e_2;

  bounding_box(P(-1.5,0), P(1,1.25));
  picture(P(200,100));
  unitlength("1pt");

  begin();

  for(int i=0; i<n; ++i)
    {
      line(P(0,0), z*pow);
      line(pow, z*pow);
      pow *= z;
    }
  red();
  boldtriangle(P(0, 0), e_1, z);
  end_red();
  label(z, P(2, -4), "$\\alpha=1+\\frac{i\\pi}{n}$");

  label(pow, P(-14,-4), "$\\alpha^n$");

  end();
}
```

Figure 4: A geometric interpretation of Euler's formula $e^{i\pi} + 1 = 0$.

Figure ?? is a "fake" 3-d figure; aspect ratios of the ellipses were calculated manually, and lines were hidden explicitly. ePiX does plot space curves, but does not currently remove hidden lines.

```
#include <epix.h>

main()
{
  double r=1/sqrt(3);

  bounding_box(P(-1, -1), P(1, 1));
  picture(P(160, 160));
  unitlength("1pt");

  begin();
  ellipse(P(0,0), P(1,1));

  red();
  boldellipse_half(P(0,0), P(r, 1),  30);
  dashellipse_half(P(0,0), P(r, 1), 210);
  end_red();

  boldellipse_half(P(0,0), P(r, 1), -90);
  dashellipse_half(P(0,0), P(r, 1),  90);

  boldellipse_half(P(0,0), P(r, 1), 150);
  dashellipse_half(P(0,0), P(r, 1), -30);

  arrow(P(0,0), P(0,1.2));
  boldarrow(P(0,0), P(-0.6*sqrt(3), -0.6));
  arrow(P(0,0), P(0.6*sqrt(3), -0.6));
  end();
}
```

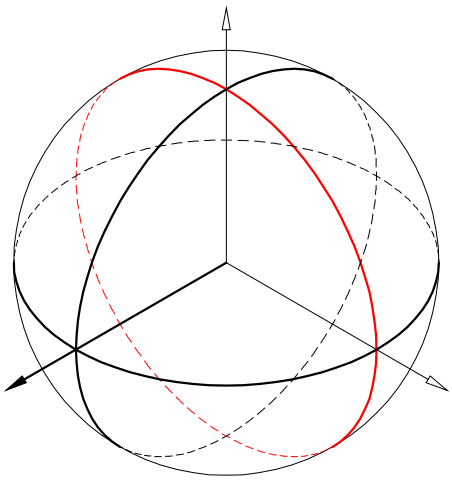


Figure 5: An orthant of a sphere.

Figure ?? uses a few variables to control the exact appearance of a figure. Inner and outer radii, arrow and label locations, and points of intersection are all computed in the figure itself, rather than being hard-wired. To rescale the figure, only r, R, and D need be changed.

```
#include <epix.h>

double a=M_PI, b=1;
double R=4.5; // outer radius
double r=0.5; // inner radius
double D=0.25;// half-width of slit

main()
{
  bounding_box(P(-5, -5), P(5,5));
  picture(P(160, 160));
  unitlength("0.35mm");

  begin();

  boldarrow(P(0,0), P(1.5*x_max,0));
  spot(P(0,0));

  // Circular arcs
  arc(P(0,0), R, asin(D/R), 2*M_PI-asin(D/R));
  arc(P(0,0), r, asin(D/r), 2*M_PI-asin(D/r));

  // Slit
  line(P(r*cos(asin(D/r)),  D), P(R*cos(asin(D/R)),  D));
  line(P(r*cos(asin(D/r)), -D), P(R*cos(asin(D/R)), -D));

  // Arrows
  arrow(P(  R/4,  r), P(3*R/4,  r));
  arrow(P(3*R/4, -r), P(  R/4, -r));

  arc_arrow(P(0,0), 0.9*R, a-b, a+b);

  label(P(R,D), P(2,4), "$R\\to\\infty$");
  label(P(0,r), P(-30,2), "$\\delta\\to0$");
  label(polar(R, M_PI/4), P(2,4), "$\\gamma$");

  end();
}
```

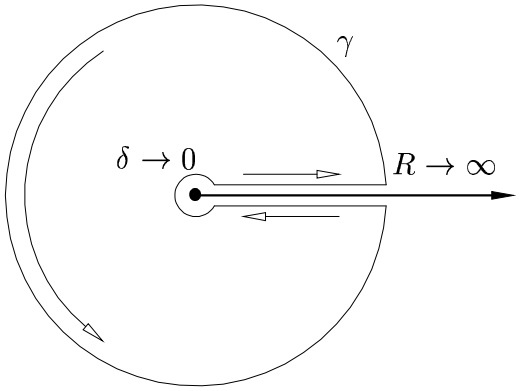


Figure 6: A keyhole contour for $\displaystyle\int_0^\infty f(z)\,dz$.

1.2 Function Plotting

ePiX provides Cartesian and polar plotting, data plotting from a file, and general parametric plotting with a few choices of plot attributes, such as clipping the plot to the bounding box or plotting one or more curves in a family. Figure ?? is a typical use of clipplot.

```
#include <epix.h>

double f(double t)
{
  return 2*t*(1-t)*(1-t);
}

double g(double t)
{
  return 1/(1-t*t);
}

main() {

  bounding_box(P(-2,-4), P(2,4));
  picture(P(200,200));
  unitlength("1pt");

  begin();

  /* Vertical asymptotes */
  dashline(P(-1, y_min), P(-1, y_max));
  dashline(P( 1, y_min), P( 1, y_max));

  /* Axes */
  h_axis(P(x_min, 0), P(x_max, 0), 8);
  v_axis(P(0, y_min), P(0, y_max), 8);

  h_axis_labels(P(x_min, 0), P(x_max, 0), 4, P(-16, 3));
  v_axis_labels(P(0, y_min), P(0, y_max), 4, P(-16, 3));

  /* Graphs */
  boldclipplot(g, x_min, x_max, 80);
  clipplot(f, x_min, x_max, 40);

  end();
}
```

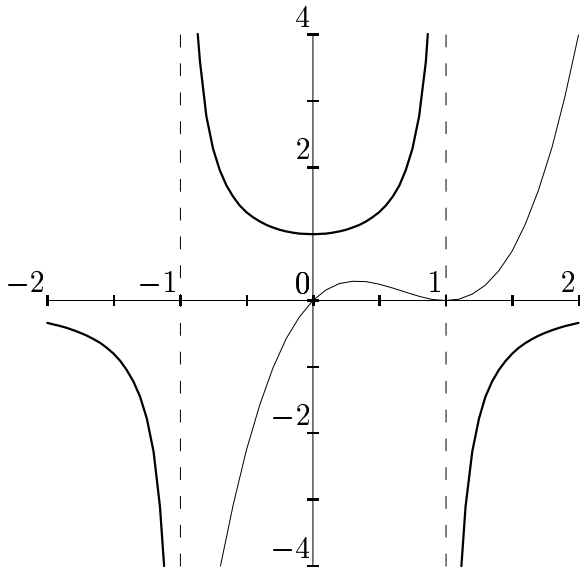


Figure 7: clipplot: Graphs truncated to bounding box.

ePiX contains primitives for Cartesian and polar grids, as well as for polar plotting. Note that angles in polar plots are measured in revolutions rather than in radians.

```
#include <epix.h>

double f(double t)
{
  return 2*cos(3*t);
}

main() {

  unitlength("1pt");
  bounding_box(P(-2,-2), P(2,2));
  picture(P(180, 180));

  begin();

  h_axis(P(x_min, y_min), P(x_max, y_min), 8);
  v_axis(P(x_min, y_min), P(x_min, y_max), 8);

  polar_grid(2, 4, 24); // radius, rings, sectors

  h_axis_labels(P(x_min,y_min), P(x_max,y_min), 4, P(-12,-14));
  v_axis_labels(P(x_min,y_min), P(x_min,y_max), 4, P(-20,-4));

  boldpolarplot(f, 0, 0.5, 60);

  end();
}
```

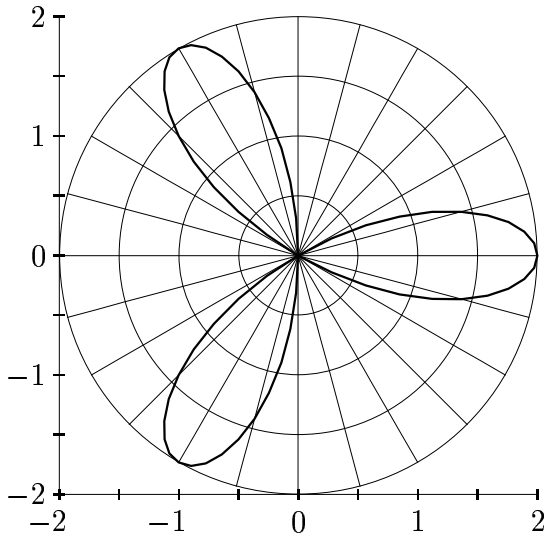


Figure 8: The polar graph $r = 2\cos 3\theta$ for $0 \leq \theta \leq \pi$.

A loop index may be used to control an entire figure, generating a sequence of snapshots of a time-varying picture, such as a rolling wheel, or the flow of an ODE.

```
#include <epix.h>

double f1(double t, double r)
{
  return t - r*sin(t);
}
double f2(double t, double r)
{
  return -r*cos(t);
}

main()
{
  double dt = 5*M_PI/11;
  double t = 0;

  picture(P(420, 120));
  bounding_box(P(0,-1), P(7,1));
  unitlength("0.005in");

  for(int i=0; i < 9; ++i)
    {
      t += dt;
      begin(); // Entire picture inside loop body

      line(P(x_min - 2, y_min), P(x_max + 6, y_min)); // the ground

      ellipse(P(t,0), P(1,1)); // the wheel

      /* the paths */
      bold();
      for (int j=0; j < 6; ++j)
        {
          rgb(1-0.125*j, 0.125*j, 0.5+0.25*j);
          sliceplot2(f1, f2, 0, t, 0.2*j, (int) ceil(1+10*t));
        }

      /* the spoke */
      green();
      line(P(t,0), P(f1(t,1), f2(t,1)));
      end_green();
      end_bold();

      end();

      /* figure separator and vertical space */
      printf("\n\\vspace*{3ex}\n%%%%");
    }
}
```

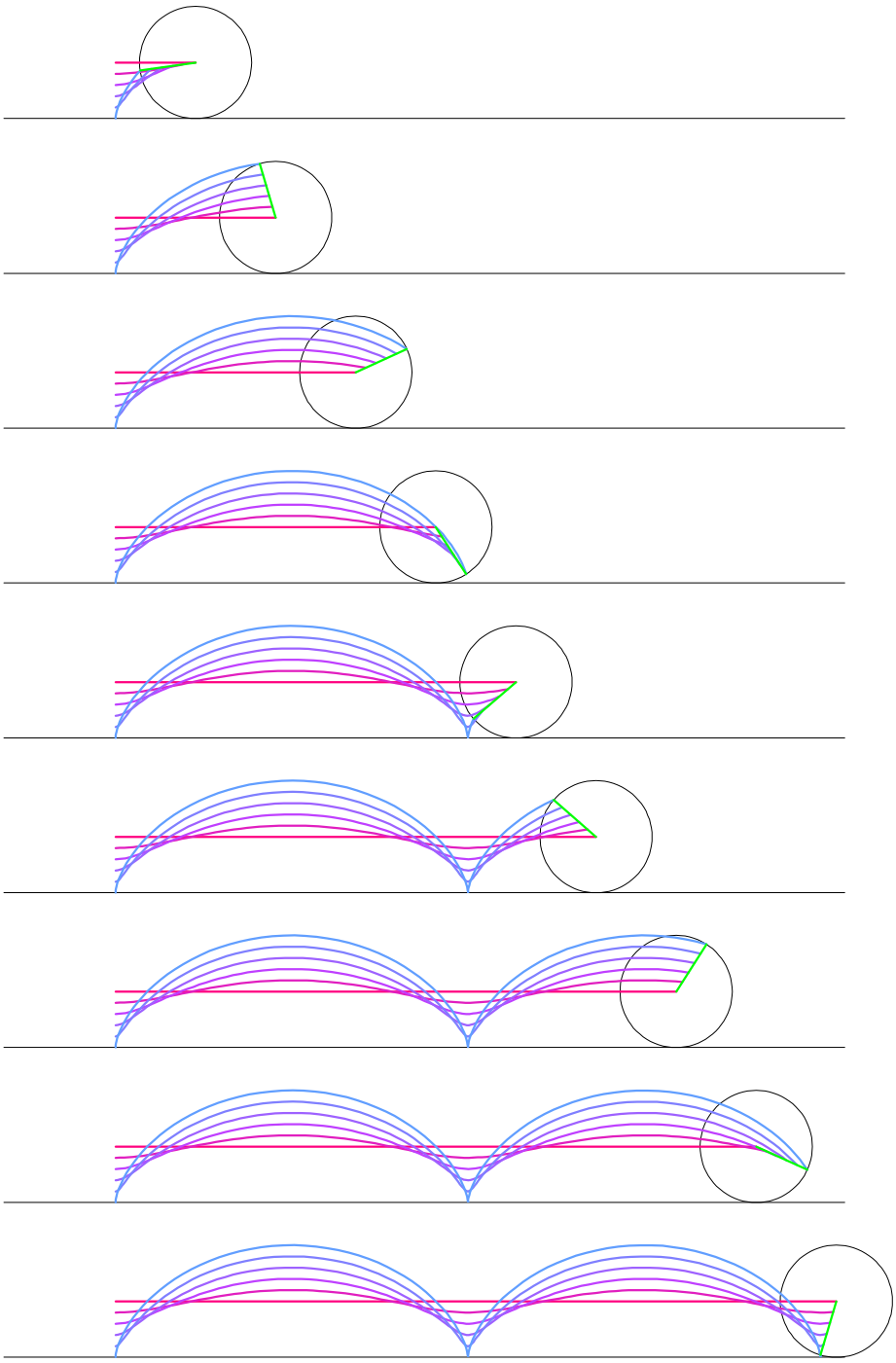


Figure 9: Snapshots of cycloids.

Because an ePiX file is a C program, it is easy to plot finite sums of series, as in Figure ??. The function being scaled and summed is cb, the "Charlie Brown" function (blue). Nowhere differentiability is essentially obvious from the picture, because the graph is self-similar (red), and is not a line.

```
#include <epix.h>
#define N 8 // Number of summands

/* "cb" is the Charlie Brown function, see functions.c */

double weierstrass(double t)
{
  int i;
  double y=0;
  for(i=0; i < N; ++i)
    y += pow(2,-i)*cb(pow(2,i)*t);

  return y;
}

main() {

  bounding_box(P(-2, -0.5), P(2, 1.5));
  picture(P(300, 150));
  unitlength("0.01in");

  begin();

  h_axis(P(x_min,0), P(x_max,0), 8);
  v_axis(P(0,y_min), P(0,y_max), 4);
  h_axis_labels(P(x_min,0), P(x_max,0), x_size, P(-8,-14));

  boldplot(weierstrass, x_min, x_max, pow(2,N));

  blue();
  plot(cb, x_min - 0.25, x_max+0.25, 4*x_size + 2);
  end_blue();

  red();
  boldplot(weierstrass, 0.5, 1.5, pow(2,N-2));
  end_red();

  end();
}
```

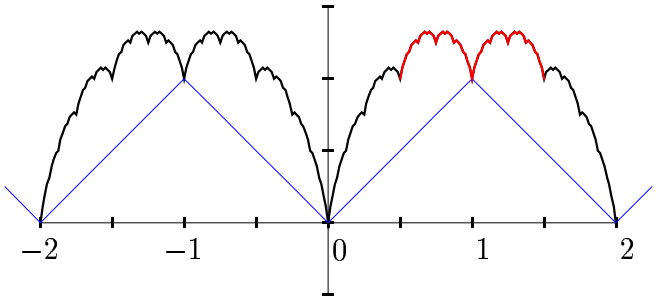


Figure 10: A Weierstrass nowhere-differentiable function.

ePiX can approximate the extreme values of a function on an interval, which is useful for drawing inscribed or circumscribed rectangles in a graph.

```
#include <epix.h>
#define N 12.0 // Number of rectangles
#define f sin  // Gather references to integrand

main() {
  int i;
  double dx;

  picture(P(300, 100));
  bounding_box(P(0,0), P(3,1));
  unitlength("0.01in");

  begin();

  dx = x_size/N;

  for(i=0; i < N; ++i)
    {
      double ai=x_min + i*dx;
      double bi=x_min + (i+1)*dx;

      rect(P(ai, 0), P(bi, sup(f, ai, bi)));
    }

  h_axis(P(x_min, y_min), P(x_max, y_min), x_size);
  v_axis(P(x_min, y_min), P(x_min, y_max), 2*y_size);

  h_axis_labels(P(x_min, 0), P(x_max, 0), x_size, P(-4, -12));
  v_axis_labels(P(x_min, 0), P(x_min, y_max), 1, P(-10, -2));

  label(P(x_max, f(x_max)), P(4,-4), "$y=\\sin x$");

  blue();
  boldplot(f, x_min, x_max, 40);
  end_blue();

  end();
}
```

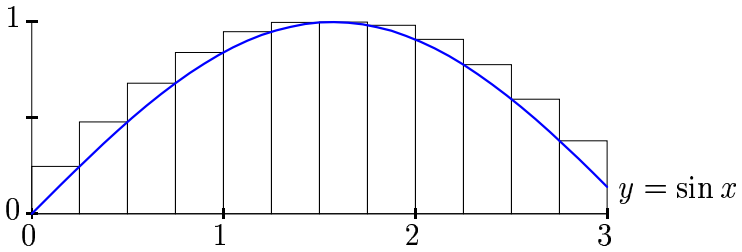
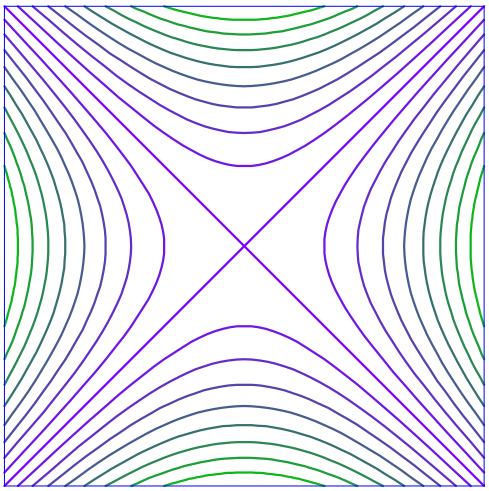


Figure 11: Upper sums for $\int_0^3 \sin x \, dx$.

Level sets may be plotted as parametrized paths using `sliceplot`, the restriction of a mapping $T : \mathbf{R}^2 \to \mathbf{R}^2$ to horizontal or vertical lines; `ePiX` does not do implicit plotting, and probably never will because of inherent instability near critical points and the resulting quality of output. Note the use of `define` to adjust the figure: The rectangle $[-3,3]\times[-3,3]$ is shown, but the size of the bounding box is controlled by the definition of `MAX`, and may be replaced by any valid `C` expression, such as `M_PI` or `0.5*(1+sqrt(5))`.

```
#include <epix.h>
#define MAX 3

double f1(double t, double r)
{
  return r*sinh(t);
}
double g1(double t, double r)
{
  return r*cosh(t);
}

main()
{
  int i;

  bounding_box(P(-(MAX),-(MAX)), P((MAX), (MAX)));
  unitlength("0.625in");
  picture(P(4,4));

  begin();

  rgb(0.5,0,1);
  boldline(P(x_min, y_min), P(x_max, y_max));
  boldline(P(x_min, y_max), P(x_max, y_min));

  for (i=1; i <= (MAX)*(MAX); ++i)
    {
      double t0 = acosh((MAX)/sqrt(i));
      double D = i*1.0/((MAX)*(MAX));

      rgb(0.5*(1-D), 0.8*D, 1-D);
      boldsliceplot2(f1, g1, -t0, t0,  sqrt(i), 40);
      boldsliceplot2(f1, g1, -t0, t0, -sqrt(i), 40);

      boldsliceplot2(g1, f1, -t0, t0,  sqrt(i), 40);
      boldsliceplot2(g1, f1, -t0, t0, -sqrt(i), 40);
  }
  blue();

  rect(P(x_min, y_min), P(x_max, y_max));
  end();
}
```



In addition to handling line objects in space, `ePiX` plots space curves; even text labels may be placed in their correct spatial location. Pairing two figures with slightly different `viewpoints` gives a cross-your-eyes stereogram. The source code for the left frame is shown; the only essential difference between the frames is their respective `viewpoints`. Because PostScript builds figures in layers, and because `ePiX` does not do hidden line removal, the source file must be constructed with some thought.

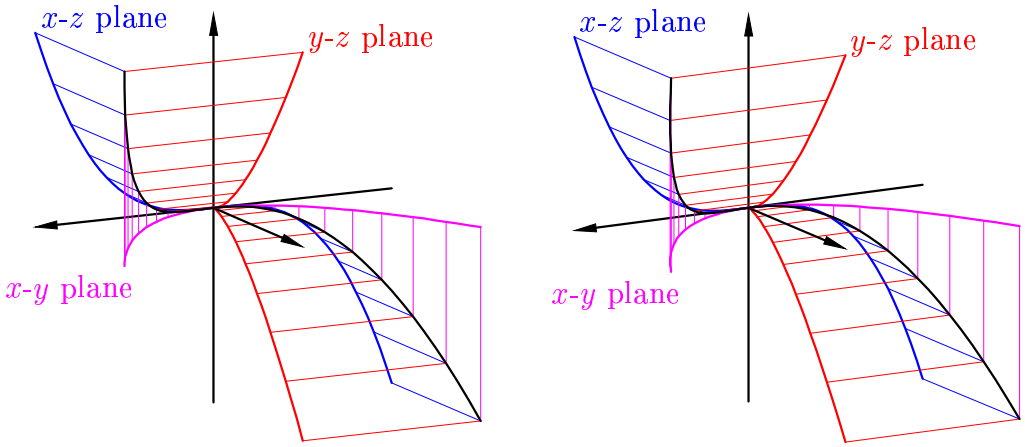


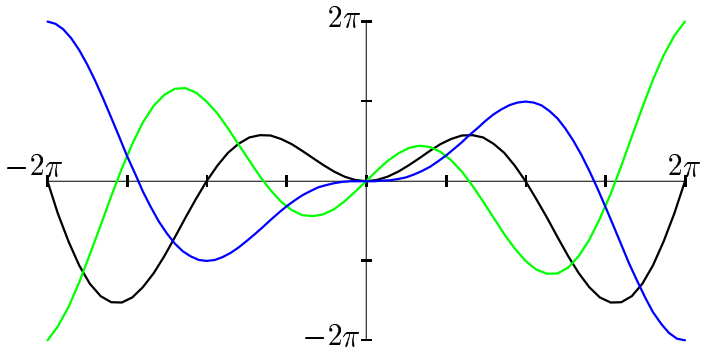
Figure 12: The twisted cubic and its coordinate plane projections.

```
#include <epix.h>

double u1(double t) {
  return t;
}
double u2(double t) {
  return t*t;
}
double u3(double t) {
  return t*t*t;
}
double zero(double t) {
  return 0;
}

int N=60;

main()
{
  int i;
  double t;

  unitlength("1pt");
  bounding_box(P(-1,-1), P(1,1));
  picture(P(150,150));
```

```
  /* Left-hand frame */
  viewpoint(1, 1.8, 0.5);

  begin();

  /* coordinate axes */
  boldarrow(V(-1,0,0), V(1,0,0));
  boldarrow(V(0,0,-1), V(0,0,1));

  blue();
  boldplot3d(u1, zero, u3, -1, 1, N);
  for (i=0; i<=20; ++i) {
      t = -1 + i/10.0;
      line(V(u1(t), 0, u3(t)), V(u1(t), u2(t), u3(t)));
    }
  label(V(1,0,1), P(2,2), "$x$-$z$ plane");
  end_blue();

  red();
  boldplot3d(zero, u2, u3, -1, 1, N);
  for (i=0; i<=20; ++i) {
      t = -1 + i/10.0;
      line(V(0, u2(t), u3(t)), V(u1(t), u2(t), u3(t)));
    }
  label(V(0,1,1), P(2,2), "$y$-$z$ plane");
  end_red();

  boldarrow(V(0,0,0), V(0,1,0));

  magenta();
  boldplot3d(u1, u2, zero, -1, 1, N);
  for (i=0; i<=20; ++i) {
      t = -1 + i/10.0;
      line(V(u1(t), u2(t), 0), V(u1(t), u2(t), u3(t)));
    }
  label(V(1,1,0), P(-45,-12), "$x$-$y$ plane");
  end_magenta();

  /* The cubic */
  boldplot3d(u1, u2, u3, -1, 1, N);

  end();
}
```

1.3 Calculus Plots

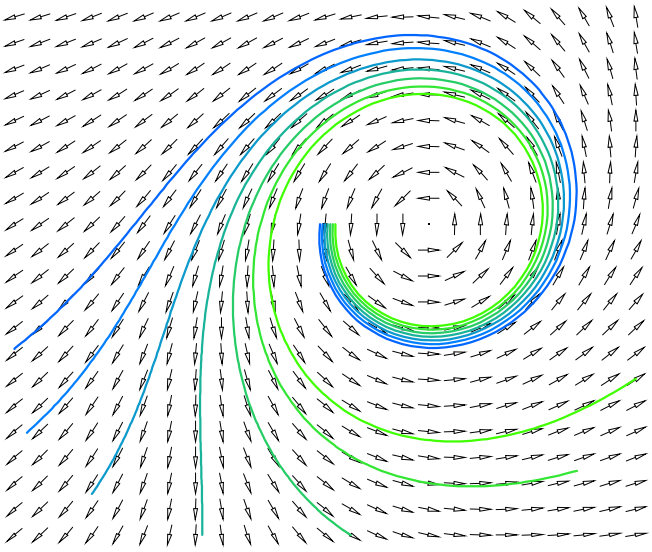
ePiX can graph derivatives, integrals, slope and vector fields, and can solve ODEs.




```
#include <epix.h>
double a=2*M_PI;
double f(double t)
{
  return t*sin(t);
}

main() {

  unitlength("1pt");
  picture(P(240, 120));
  bounding_box(P(-a,-a), P(a,a));

  begin();

  /* Coordinate axes and labels */
  h_axis(P(x_min,0), P(x_max,0), 8);
  v_axis(P(0,y_min), P(0,y_max), 4);

  label(P(0,y_max), P(-24,-2), "$\\phantom{-}2\\pi$");
  label(P(0,y_min), P(-24,-2), "$-2\\pi$");

  label(P(x_min,0), P(-16,2), "$-2\\pi$");
  label(P(x_max,0), P(-16,2), "$\\phantom{-}2\\pi$");

  boldplot(f, x_min, x_max, 60);
  green();
  boldplot_deriv(f, x_min, x_max, 60);
  end_green();
  /* Definite integral from x=0 must be graphed in two pieces */
  blue();
  boldplot_int(f, 0, x_max, 40);
  boldplot_int(f, 0, x_min, 40);
  end_blue();

  end();
}
```

Figure 13: $y = x \sin x$ (black), its derivative (green) and integral from 0 (blue).

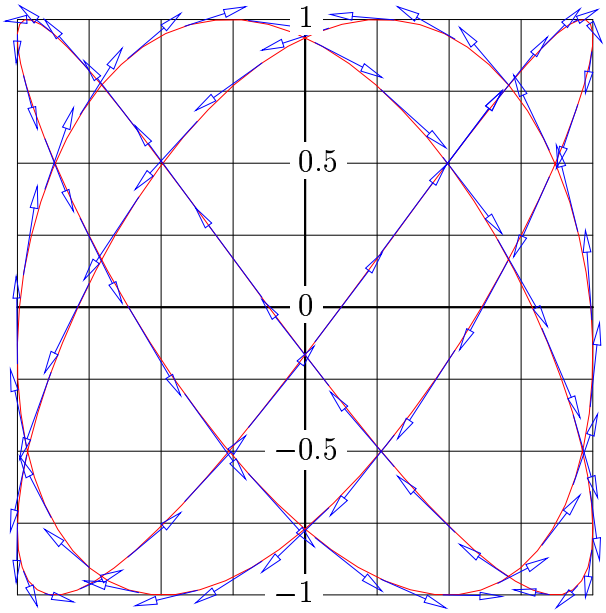
ODE plotting is done via Euler's method with small step size. More accurate numerical methods will not improve output quality in simple situations or over short time intervals, because of limitations in LaTeX itself.



```
#include <epix.h>
double f1(double s, double t)
{
  return 0.1*s - t/(0.01+s*s+t*t);
}
double f2(double s, double t)
{
  return 0.025*t + s/(0.01+s*s+t*t);
}


main() {
  int i=0;

  unitlength("1pt");
  bounding_box(P(-4, -3), P(2,2));
  picture(P(234, 195));

  begin();

  dart_field(f1, f2, P(x_min, y_min), P(x_max, y_max), 4*x_size, 4*y_size);

  for (int i=0; i<7; ++i)
    {
      rgb(0.25 - 0.05*i, 1 - 0.1*i, 0.2*i);
      boldode_plot(f1, f2, P(-0.9-0.025*i,0), 20, 200);
    }
  end();
}
```

Figure 14: A slope field and six solutions of an ODE.

`tan_field` graphs the field of tangent vectors along a parametrized path; the path itself must be drawn separately. The choice of contrasting colors emphasizes places where the curvature is large. This figure again demonstrates how PostScript builds up a figure in layers; objects in the file are overlaid by subsequent objects. In particular, the axis labels are "masked".




```
#include <epix.h>

double f1(double t) {
  return sin(3*M_PI*t+0.5);
}
double f2(double t) {
  return sin(4*M_PI*t+0.5);
}

main() {

  bounding_box(P(-1,-1), P(1,1));
  picture(P(300, 300));
  unitlength("0.01in");

  begin();

  grid(P(x_min,y_min), P(x_max,y_max), 4*x_size, 4*y_size);
  boldline(P(x_min, 0), P(x_max, 0));
  boldline(P(0, y_min), P(0, y_max));

  red();
  paramplot(f1, f2, 0, 2, 200);
  end_red();
  blue();
  tan_field(f1, f2, 0, 2, 81);
  end_blue();

  v_axis_masklabels(P(0, y_min), P(0, y_max), 2*y_size, P(-15, -3));

  end();
}
```

Figure 15: The tangent field of a Lissajous curve.

`ePiX` plots fractal curves generated by "seeds". A seed is a (usually short) piecewise-linear curve whose equal-length segments point in one of n equally-spaced directions ("spokes"). Recursively up to a pre-specified depth D, each segment is replaced by a scaled, rotated, translated copy of the seed (shown in red). The sequence `{6,4,0,1,-1,0}` generates the Koch snowflake curve: There are 6 spokes, the seed has length 4, and the remaining entries are the seed proper.

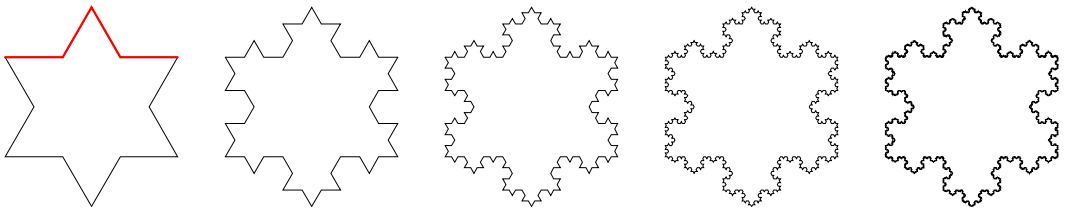


Figure 16: Approximations of the Koch snowflake curve.

```
#include <epix.h>

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

main()
{
  bounding_box(P(-1,-1), P(1, 1));
  picture(P(75, 75));
  unitlength("1pt");

  for (int i=1; i <= 5; ++i)
    {
      begin();

      fractal(cis(5*M_PI/6), cis(  M_PI/6), i, koch_seed);
      fractal(cis(  M_PI/6), cis( -M_PI_2), i, koch_seed);
      fractal(cis( -M_PI_2), cis(5*M_PI/6), i, koch_seed);

      if (i==1)
        {
          bold();
          red();
          fractal(cis(5*M_PI/6), cis(M_PI/6), i, koch_seed);
          black();
          end_bold();
        }
        end();
    }
}
```