`ePiX` Sample Document

Version 1.0

September, 2004

1 Overview

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

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

This sample document demonstrates a few of `ePiX`'s capabilities, with side-by-side comparisons of input files[1] and corresponding output. `ePiX` has several distinguishing features:

- Scenes are described in mathematically natural Cartesian coordinates, making `ePiX` effectively a vector format. Well-designed figures are of camera quality over a range of sizes and aspect ratios. There are almost no default choices; aspect ratio, color, viewpoint (for 3-D figures) and line width are wholly controllable.
- High-quality typography is easily incorporated.
- All the power of `C++` is available in using and extending the program's capabilities.
- The license, the GNU GPL, is similar to the terms on theorems: You may run `ePiX` for any purpose, examine the code, study how the program works, make improvements, and distribute your improvements so long as you do not restrict the rights of others to do the same.

2 Examples

`ePiX` provides traditional plotting capabilities, such as Cartesian and polar plots, data plotting from a file, and parametric curves and surfaces (without automatic hidden object removal). As mentioned, the implementation allows `ePiX` to be regarded as a programming language; any numerical algorithm written in `C` or `C++` can be used in an `ePiX` figure. The use of programming constructs can make a figure flexible (so that the appearance can be precisely but dramatically altered by making a few small changes to the input file) and easier to maintain.

The figures below were chosen to emphasize results that are relatively difficult to achieve with existing Free (and some commercial) plotting software. Simple line figures containing polygons, ellipses, and splines are almost self-explanatory, so no examples are given.

In the files below, several short statements are often put on a single line to make the file fit on a page; this is not good programming practice, and should be avoided in real files. Several sample files could be shortened by hard-wiring constants.

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

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

// the function to be plotted
double f(double x) { return x*x; }

int main()
{
  bounding_box(P(-2,0),P(2,4));
  unitlength("1in");
  picture(3,3);

  begin();

  grid(32,32); // fine grid

  bold();
  grid(4,4); // and coarse

  h_axis_labels(4, P(0,-2), b);
  v_axis_masklabels(P(0, 1), P(0, y_max), 3, P(0,0), c); // omit 0

  red();
  plot(f,x_min,x_max,20);

  end();
}
```

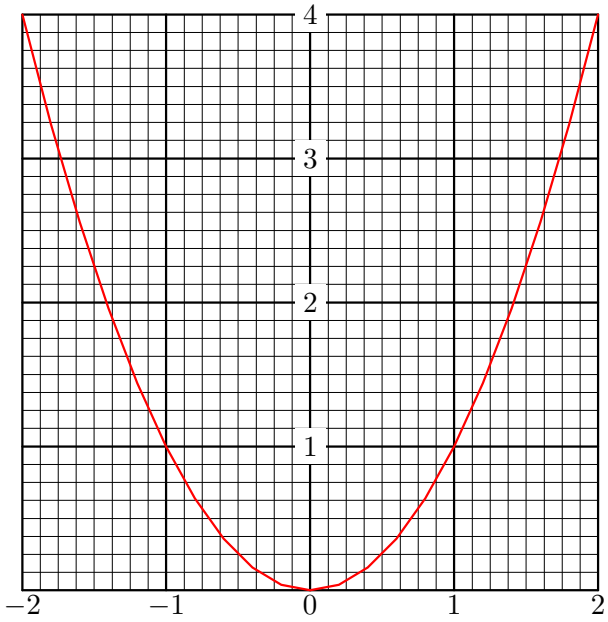


Figure 1: A basic plot.

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

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

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

  begin();

  crop();
  // Vertical asymptotes
  dashed();
  line(P(-1, y_min), P(-1, y_max));
  line(P( 1, y_min), P( 1, y_max));
  solid();

  // Axes
  h_axis(8);
  v_axis(8);

  h_axis_labels(4, P(-1, 2), tl); // align top-left
  v_axis_labels(4, P(-1, 2), tl);

  // Graphs
  plot(f, x_min, x_max, 40);
  bold();
  plot(g, x_min, x_max, 80);

  end();
}
```

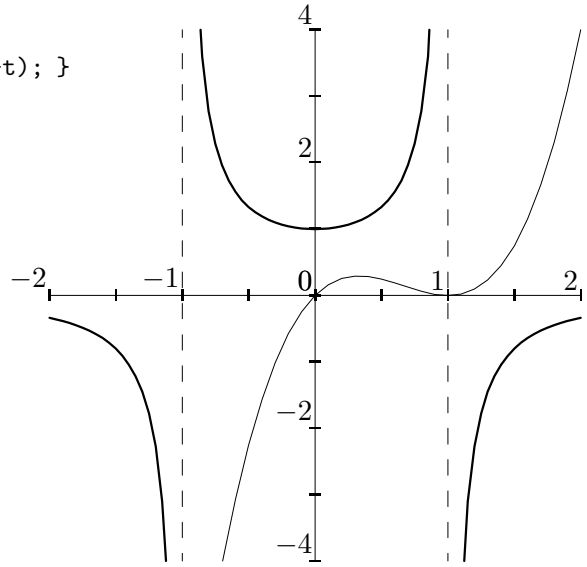


Figure 2: “Cropping” removes elements outside the bounding box.

There are angular "modes" for polar and spherical coordinates. `ePiX` defines its own trig functions, which are sensitive to the current mode.

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

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

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

  begin();
  degrees();

  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), x_size, P(-12,-14));
  v_axis_labels(P(x_min,y_min), P(x_min,y_max), y_size, P(-4,0), 1);

  bold();
  polarplot(f, 0, 180, 120); // plot over [0, 180] using 120 intervals

  end();
}
```

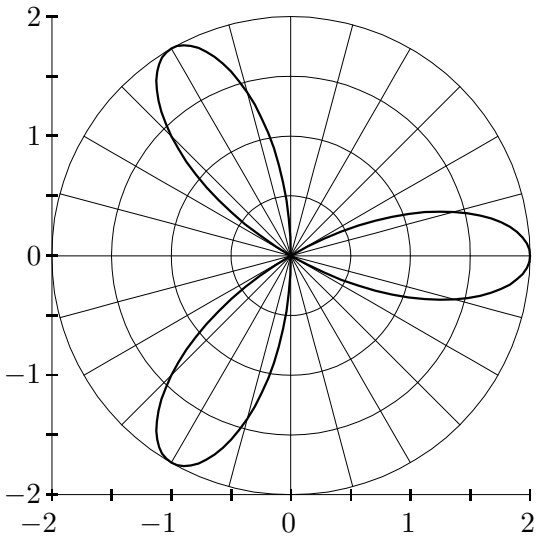


Figure 3: The polar graph $r = 2\cos 3\theta$ for $0 \leq \theta \leq \pi$.

Regions between graphs can be shaded; gray density ranges from 0 (white) to 1 (black).

```
#include "epix.h"
using namespace ePiX;
double k=4;           // change width of hump
double dx = 0.05;     // width of thin shaded region
double x = 1/sqrt(k); // position of thin shaded region
double dy = 0.25;     // for vertical distance between labels; kludgey

double f(double t) { return sqrt(fabs(k)/(2*M_PI))*exp(-k*t*t); }
P pt1 = P(x, f(x)+2*dy), pt2 = P(x+dx,f(x)+dy), pt3 = P(x+2*dx,f(x));

int main()
{
  unitlength("1pt");
  picture(150, 150);
  bounding_box(P(0,0), P(1,1));

  begin();
  fill();
  gray(0.1);     shadeplot(f, x_min, x, 90);
  gray(0.4);     rect(P(x,f(x)), P(x+dx, 0));
  gray(0.6);     shadeplot(f, x, x+dx, 10);
  fill(false);

  label(pt1, P(0,0), "$F(x)=\\displaystyle\\int_a^x f(t)\\,dt$", tr);
  arrow(pt1, P(0.5*(x_min+x), 0.5*f(0.5*(x_min+x))));

  label(pt2, P(0,0), "Area of rectangle = $f(x)\\,dx$", tr);
  arrow(pt2, P(x+0.5*dx, f(x)));

  label(pt3, P(0,0), "Area = $F(x+dx)-F(x)$", tr);
  arrow(pt3, P(x+dx, 0.5*f(x+dx)));

  bold();
  plot(f, x_min, x_max, 120);

  plain(); h_axis(4); v_axis(4);

  label(P(x_min,0), P(0,-5), "$a$", b);
  label(P(x,0), P(0,-5), "$x$", b);
  label(P(x+dx,0), P(0,-2), "$x+dx$", br);

  end();

}
```

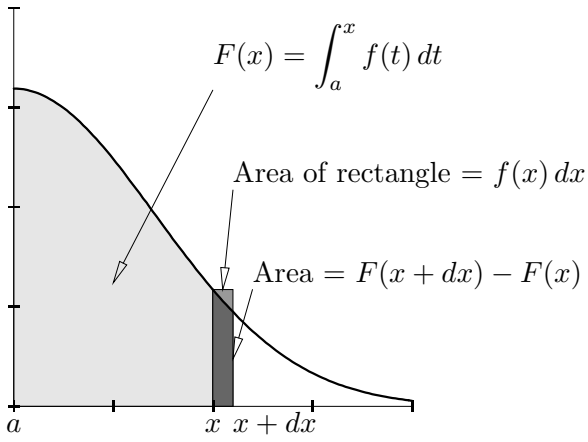
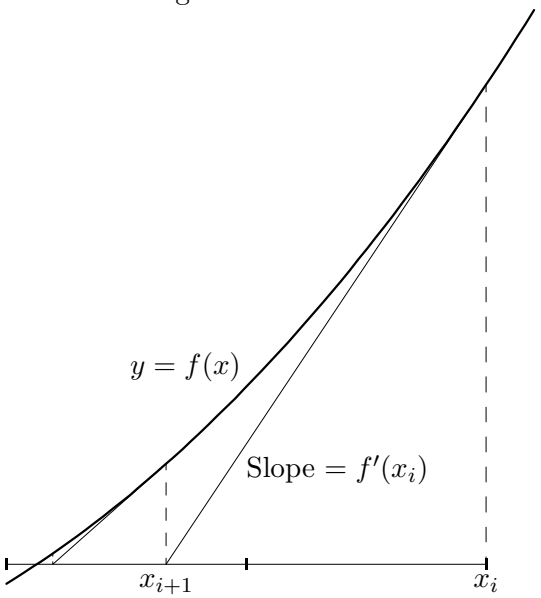


Figure 4: The first fundamental theorem of calculus: $F'(x) = f(x) + o(1)$.

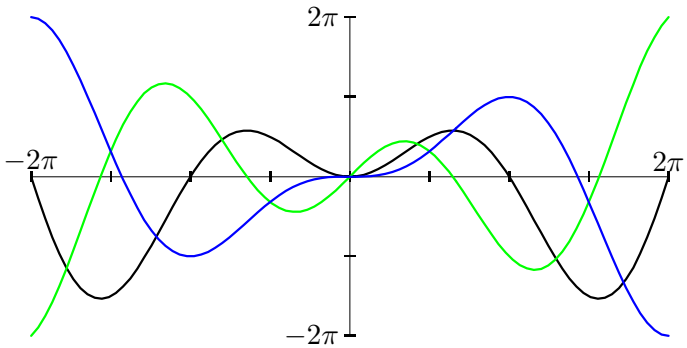
Control structures—loops and decision statements—can be used to create input files whose logical structure matches the mathematical structure of the figure.




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

double f(double t) { return t*t*t-3*t+1; }
double df(double t) { return 3*t*t - 3; }

int main()
{
  bounding_box(P(1.5,0),P(2,3));
  unitlength("1in");
  picture(2.5,2.5);

  begin();
  h_axis(2);

  double x0 = 2;
  label(P(x0,0), P(0,-4), "$x_i$", b);

  for (int i=0; i < 3; ++i)
    {
      dashed();     line(P(x0,0), P(x0,f(x0)));
      solid();      line(P(x0,f(x0)), P(x0-f(x0)/df(x0),0));

      x0 -= f(x0)/df(x0); // Newton's method
    }

  bold();    plot(f, x_min, x_max+0.05, 60);

  double x1 = 2-f(2)/df(2);

  label(P(1.75, f(1.75)), P(-2,2), "$y=f(x)$", tl);
  label(P(x1,0), P(0,-4), "$x_{i+1}$", b);
  label(P(1.75, df(2)*(1.75-x1)), P(0,-4), "Slope $= f'(x_i)$", br);

  end();
}
```

Figure 5: Newton's method for root approximation

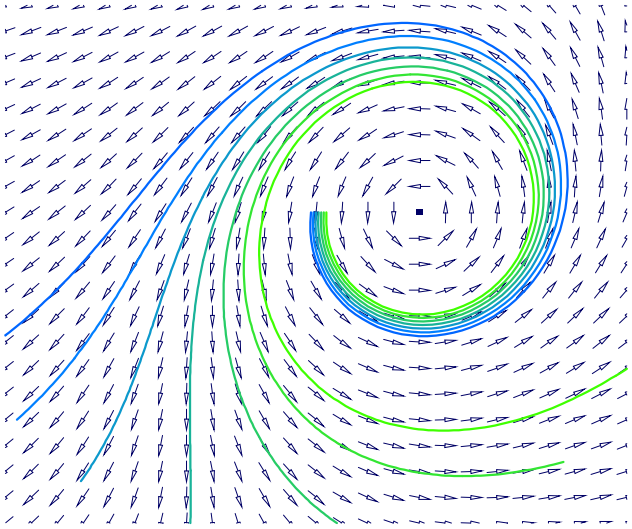
ePiX can graph derivatives and definite integrals:




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

double MAX=2*M_PI;
double f(double t)
{
  return t*Sin(t);
}

int main()
{
  unitlength("1pt");
  picture(240, 120);
  bounding_box(P(-MAX,-MAX), P(MAX,MAX));

  begin();

  // Coordinate axes and labels
  h_axis(8);
  v_axis(4);

  label(P(0,y_max), P(-4,0), "$2\\pi$", l);
  label(P(0,y_min), P(-4,0), "$-2\\pi$", l);

  label(P(x_min,0), P(0,2), "$-2\\pi$", t);
  label(P(x_max,0), P(0,2), "$2\\pi$", t);

  bold();
  plot(f, x_min, x_max, 90);
  green();
  plot(D(f), x_min, x_max, 90);    // derivative class "D"

  blue();
  plot(I(f, 0), x_min, x_max, 90); // definite integral from x=0

  end();
}
```

Figure 6: $y = x \sin x$ (black), its derivative (green) and integral from 0 (blue).

Solutions of ODEs are computed with Euler's method. Vector fields may be plotted at constant (shown) or true length.



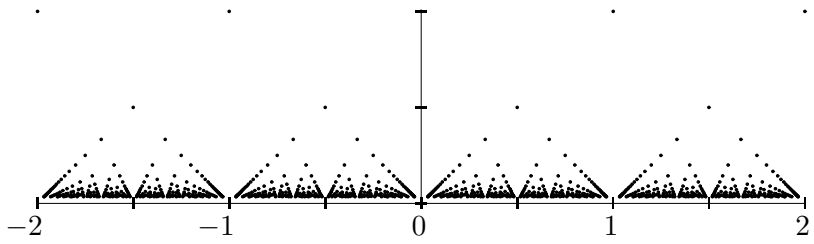
```
#include "epix.h"
using namespace ePiX;

P F(double s, double t)
{
  return (P(0.1,0.025)&P(s,t)) +
    (1/(0.01+s*s+t*t))*P(-t,s);
}

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

  begin();
  crop();
  blue(0.4);
  dart_field(F, P(x_min, y_min), P(x_max, y_max), 4*x_size, 4*y_size);

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

Figure 7: A slope field and six solutions of an ODE.

The "denominator" function f, defined by

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

can be plotted with a nested `for` loop.




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

int N = 30; // maximum denominator plotted

int main()
{
  unitlength("0.01in");
  bounding_box(P(-2,0), P(2,1));
  picture(400,100);

  begin();

  h_axis(2*x_size);
  v_axis(2);

  h_axis_labels(x_size, P(-12, -12));

  dot_size(1); // draw dots 1pt in diameter
  for (int i=1; i< N; ++i)
    for (int j=i*x_min; j <= i*x_max; ++j)
      if (gcd(i, j) == 1)
        dot(P(j*1.0/i, 1.0/i));

  end();
}
```

Figure 8: A "pathological" function in real analysis.

Finite sums are defined by an algorithm and are therefore easy to plot. 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 not a line.

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

const int N=8; // Number of summands

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

  return y;
}

int main()
{
  bounding_box(P(-2, 0), P(2, 1.5));
  picture(320, 120);
  unitlength("0.01in");

  begin();

  h_axis(2*x_size);
  v_axis(2*y_size);
  h_axis_labels(x_size, P(-12,-14));

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

  bold();
  black();
  plot(weierstrass, x_min, x_max, pow(2,N));

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

  end();
}
```

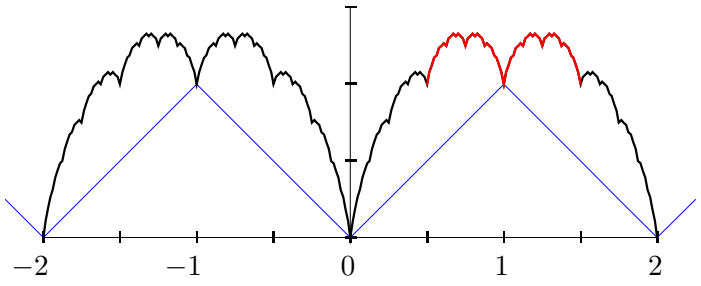


Figure 9: A Weierstrass nowhere-differentiable function.

`ePiX` can approximate the extreme values of a function on an interval, which is useful for drawing inscribed or circumscribed rectangles in a graph. The sine function is symbolized by `f` throughout the body (except the label, of course), so re-drawing the figure with a different integrand only requires changing the definition of `f` and re-sizing the `bounding_box`. The number of rectangles is similarly "parametrized".

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

const int N=12; // Number of rectangles
double f(double t) { return Sin(t); } // gather references to integrand

int main()
{
  bounding_box(P(0,0), P(3,1));
  unitlength("1in");
  picture(3, 1);

  begin();

  double dx = x_size/N;
  gray(0.1); // (un)set filling in loop

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

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

  h_axis(x_size);
  v_axis(2*y_size);

  h_axis_labels(x_size, P(0,-4), b);
  label(P(x_max, f(x_max)), P(4,0), "$y=\\sin x$", r);

  bold();    blue();
  plot(f, x_min, x_max, 40);

  end();
}
```

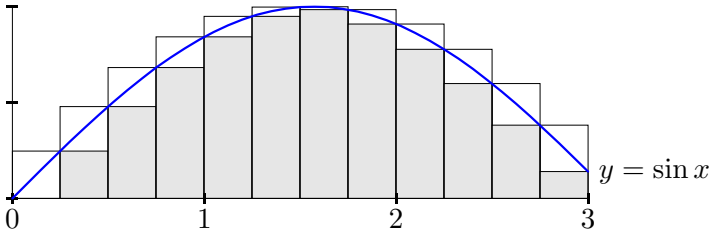


Figure 10: Upper and lower sums for $\int_0^3 \sin x\,dx$.

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. The `ePiX` project page contains links to animations that can be played in any web browser.

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

P F(double t, double r)
{
  return P(t-r*Sin(t), 1-r*Cos(t));
}

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

  picture(8, 1);
  bounding_box(P(-1,0), P(15,2));
  unitlength("0.625in");

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

      line(P(x_min, y_min), P(x_max, y_min)); // the ground

      circle wheel(P(t,1), 1);  wheel.draw();

      domain R = domain(P(0,0), P(t, 1), // [0,t] x [0,1]
                        mesh(10*i,5), mesh((int) ceil(1+4*t), 5));

      // the paths
      bold();
      for (int j=0; j < 6; ++j)
        {
          rgb(1-0.125*j, 0.125*j, 0.5+0.25*j);
          plot(F, R.slice2(0.2*j));
        }

      // the spoke
      green();
      line(P(t,1), F(t,1));

      end();

      // figure separator and vertical space
      if (i < 8)
        printf("\n\\vspace*{3ex}\n%%%%");
    }
}
```

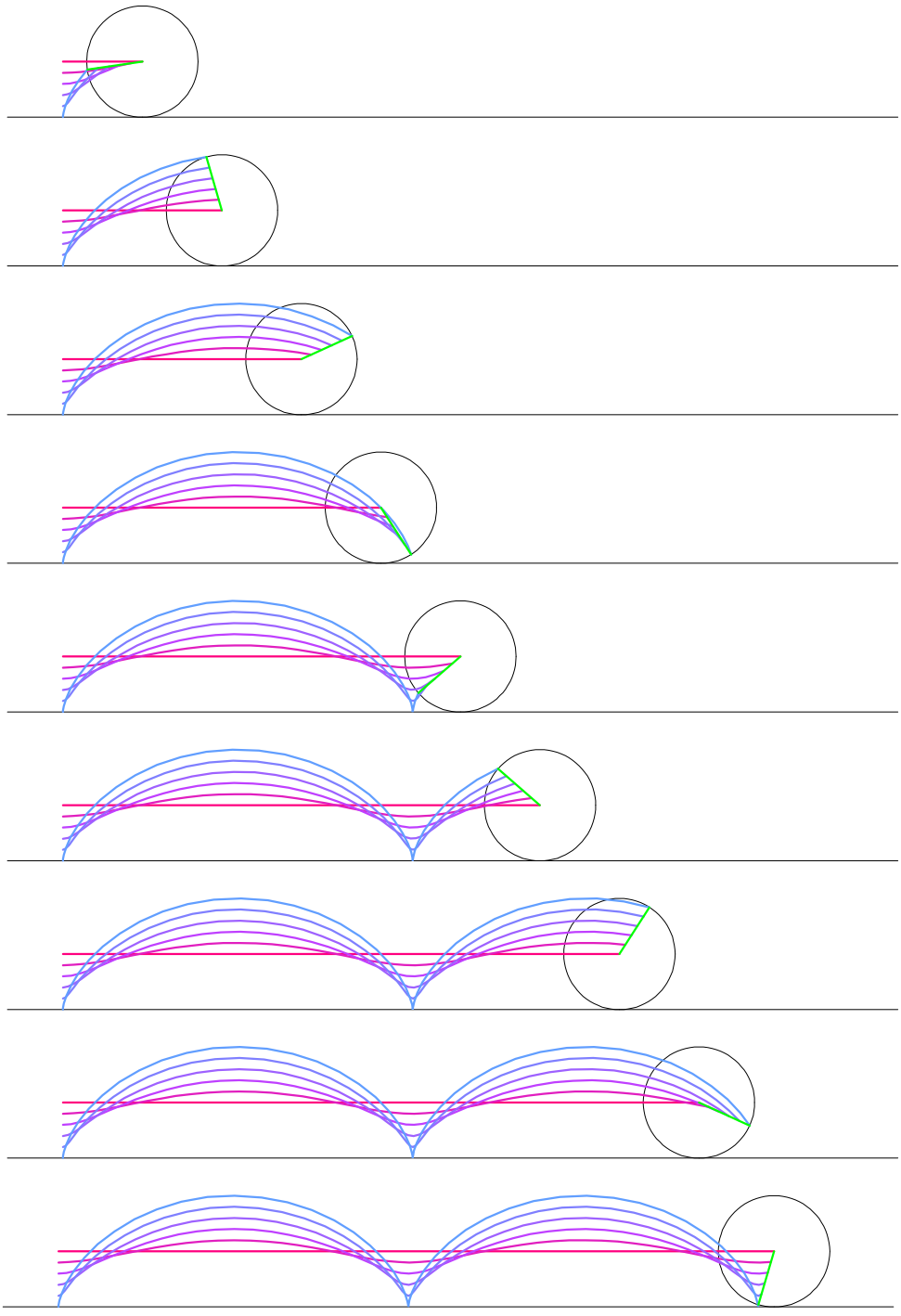


Figure 11: Snapshots of cycloids.

Figure **??** (by Jacques L'helgoual) demonstrates some spherical geometry capabilities: Space curves can be projected to the unit sphere radially, and plane curves can be projected stereographically from the north or south pole, with or without hidden line removal.

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

const double k = 2*M_PI/(360*sqrt(3)); // assume "degrees" mode
double exp_cos(double t) { return exp(k*t)*Cos(t); }
double exp_sin(double t) { return exp(k*t)*Sin(t); }
double minus_exp_cos(double t) { return -exp_cos(t); }
double minus_exp_sin(double t) { return -exp_sin(t); }

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

  begin();
  degrees(); // set angle units
  viewpoint(1, 2.5, 3);

  sphere S1=sphere(); // unit sphere
  S1.draw();

  pen(0.15); // hidden portions of loxodromes
  blue();
  backplot_N(exp_cos, exp_sin, -540, 540, 90);
  backplot_N(minus_exp_cos, minus_exp_sin, -540, 540, 90);
  red();
  backplot_N(exp_sin, minus_exp_cos, -540, 540, 90);
  backplot_N(minus_exp_sin, exp_cos, -540, 540, 90);

  black(); // coordinate grid
  for (int i=0; i<=12; ++i) { latitude(90-15*i,0,360); longitude(30*i,0,360); }

  bold(); // visible portions of loxodromes
  blue();
  frontplot_N(exp_cos, exp_sin, -540, 540, 360);
  frontplot_N(minus_exp_cos, minus_exp_sin, -540, 540, 360);
  red();
  frontplot_N(exp_sin, minus_exp_cos, -540, 540, 360);
  frontplot_N(minus_exp_sin, exp_cos, -540, 540, 360);

  end();
}
```

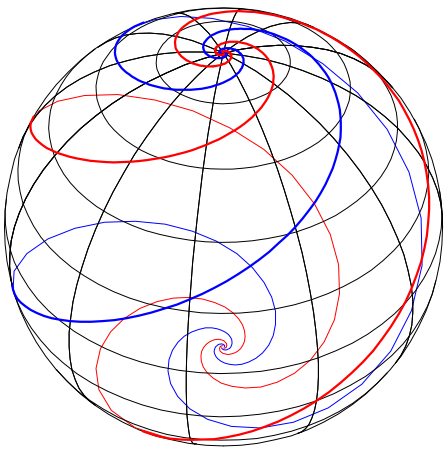
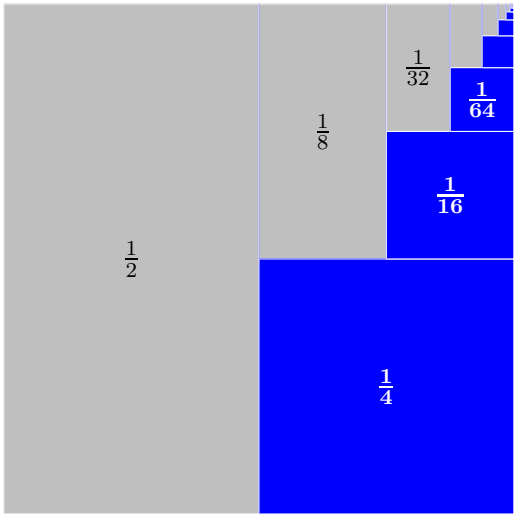


Figure 12: Loxodromes on the unit sphere.

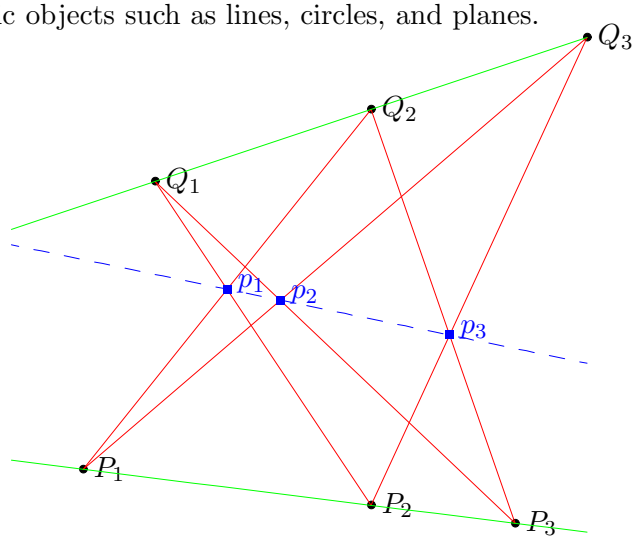
PSTricks can be incorporated in ePiX files to achieve colored filling and other effects. The psset command puts its argument into the output file.




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

const int N=8;

int main()
{
  unitlength("1.5pt");
  bounding_box(P(0,0), P(1,1));
  picture(128,128);

  use_pstricks();
  begin();

  psset("fillcolor=lightgray, linecolor=white");
  rect(P(0,0), P(1,1));
  label(P(1.0/4,   1.0/2), "$\\frac{1}{2}$");
  label(P(5.0/8,   3.0/4), "$\\frac{1}{8}$");
  label(P(13.0/16, 7.0/8), "$\\frac{1}{32}$");

  psset("fillcolor=blue");
  double t=0.5;

  for(int i=0; i<N; ++i, t *= 0.5)
    {
      rect(P(1-t, 1-2*t), P(1, 1-t));
      line(P(1-t, 1-2*t), P(1-t, 1));
    }
  white();
  label(P(3.0/4,     1.0/4), "$\\mathbf{\\frac{1}{4}}$");
  label(P(7.0/8,     5.0/8), "$\\mathbf{\\frac{1}{16}}$");
  label(P(15.0/16, 13.0/16), "$\\mathbf{\\frac{1}{64}}$");

  end();
}
```

Figure 13: A geometric series with ratio 1/2.

ePiX computes intersections of geometric objects such as lines, circles, and planes.




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

int main()
{
  bounding_box(P(-2,-2),P(2,2));
  unitlength("1in");
  picture(3,3);

  begin(); crop();

  P V = P(2,-0.25),   W = P(3,1);
  P P1=P(-1.5,-1),    P2 = P1 + V,       P3 = P1 + 1.5*V;    // points
  P Q1 = P(-1,1),     Q2 = Q1 + 0.5*W,   Q3 = Q1 + W;

  segment L12(P1, Q2),   L13(P1, Q3);
  segment L21(P2, Q1),   L23(P2, Q3);
  segment L31(P3, Q1),   L32(P3, Q2);

  P R1 = L12*L21,    R2 = L13*L31,    R3 = L32*L23;   // points of intersection

  dot(P1);   dot(P2);   dot(P3);    dot(Q1);   dot(Q2);   dot(Q3);

  label(P1, P(4,0), "$P_1$", r);  label(Q1, P(4,0), "$Q_1$", r);
  label(P2, P(4,0), "$P_2$", r);  label(Q2, P(4,0), "$Q_2$", r);
  label(P3, P(4,0), "$P_3$", r);  label(Q3, P(4,0), "$Q_3$", r);

  red();
  L12.draw();    L21.draw();   L13.draw();    L31.draw();    L32.draw();    L23.draw();

  green();    Line(P1,P3);    Line(Q1,Q3);

  blue();
  box(R1);    label(R1, P(4,2), "$p_1$", r);
  box(R2);    label(R2, P(4,2), "$p_2$", r);
  box(R3);    label(R3, P(4,2), "$p_3$", r);

  dashed();    Line(R1,R3);
  end();
}
```

Figure 14: Pascal's theorem on sides of a hexagon.

Paths may be built in pieces and manipulated as a single object.

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

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

  use_pstricks();
  begin();  degrees();
  double theta=60;
  double rad=0.25, Rad=4.5; // arc radii
  double ht=1.0/16;         // half-width of slit

  // "start" angles of arcs
  double theta1 = Asin(ht/rad), theta2 = Asin(ht/Rad);

  // build keyhole in pieces; += concatenates, -= reverses orientation
  path contour(P(0,0), Rad*E_1, Rad*E_2, theta2, 360-theta2);    // outer arc
  contour -= path(polar(rad,-theta1), polar(Rad,-theta2));       // lower slit
  contour -= path(P(0,0), rad*E_1, rad*E_2, theta1, 360-theta1); // inner arc
  contour += path(polar(rad, theta1), polar(Rad, theta2));       // upper slit

  fill();
  std::cout << "\n\\newrgbcolor{orange}{1 0.7 0.2}";

  psset("fillcolor=orange,linecolor=red,linewidth=1.5pt");
  contour.set_fill();
  contour.draw();
  use_pstricks(false);

  dot(P(0,0));   // the origin
  gray(1); // blacken arrowheads
  arrow(P(Rad/4, 0.1*Rad), P(3*Rad/4, 0.1*Rad)); // arrow ends in terms of Rad
  arrow(P(3*Rad/4, -0.1*Rad), P(Rad/4, -0.1*Rad));
  arc_arrow(P(0,0), 0.9*Rad, 180-theta, 180+theta);

  label(P(Rad,ht), P(2,4), "$R\\to\\infty$", tr);
  label(P(0,rad), P(0,4), "$\\delta\\to0$", tl);
  label(polar(Rad, 45), P(0,0), "$\\gamma$", tr);

  end();
}
```

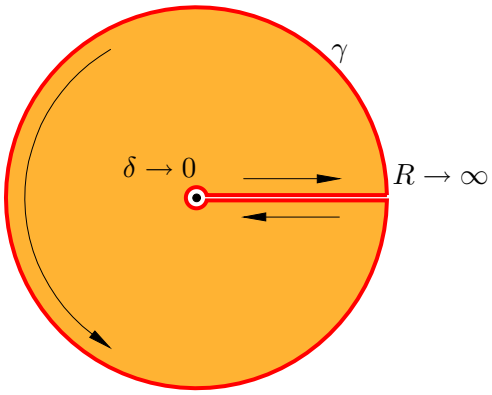


Figure 15: A keyhole contour for a branch cut integral.

ePiX provides “clipping”, a 3-dimensional analogus of cropping; figure elements that lie outside a coordinate box are removed.

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

P F(double x, double y) { return P(x, y, x/(x*x+y*y)); }
P pt0=P(), pt1 = P(1.5,1.5), pt2 = P(-1.5,1.5);
domain R(-pt1, pt1, mesh(24,24), mesh(72,72));

int main()
{
  bounding_box(-pt1, pt1);
  unitlength("1in");
  picture(2.5,2.5);

  begin();
  viewpoint(-2,-4,3);  camera.range(10);
  clip_box(P(-1.5,0,-2), P(1.5,1.5,2)); clip();

  magenta(); grid(pt0, pt1, 6, 6); // right half, w/layering
  bold(); black(); plot(F, R);     // surface mesh

  label(P(0, 0, -1.5), P(2,0),
        "$\\displaystyle z=\\mathrm{Re}\\,\\frac{1}{x+iy}$", r);

  plain(); gray(0); fill(); rect(pt0, pt2); // left half
  fill(false); magenta(); grid(pt0, pt2, 6, 6);
  label(P(0,0), P(2,-2), "0", br);
  label(P(x_max,0), P(2,0), "$x$", r);
  label(P(0,y_max), P(-1,2), "$y$", t);

  pen(1.5); red(); plot(F, R.slice2(0)); // graph over real axis
  // re-draw mesh to simulate transparency
  pen(0.15); rgb(0.5,0.5,0.5); plot(F, R.resize1(-1.5,0));
  end();
}
```

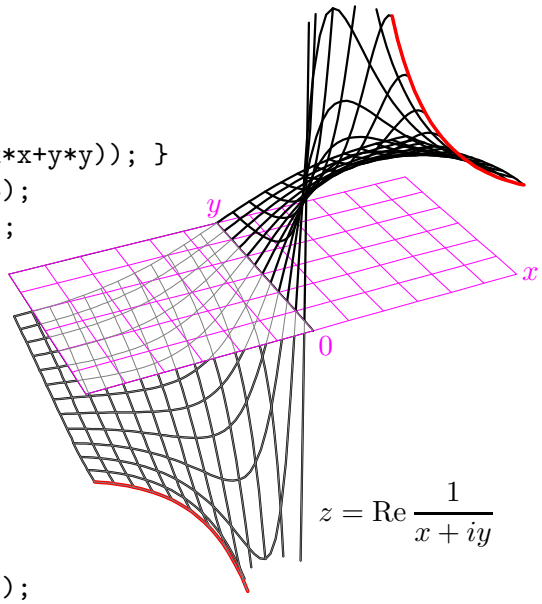
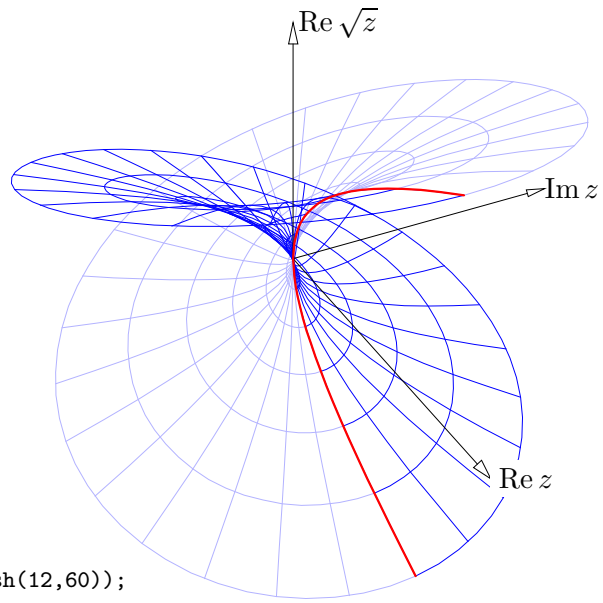


Figure 16: The real part of the complex reciprocal.




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

P f(double x, double y)
{
  pair z = pair(x,y);
  pair temp = z*z;
  return P(temp.x1(), temp.x2(), x);
}

inline P F(double r, double t)
{ return f(r*Cos(t), r*Sin(t)); }

P g(double t) { return t*P(t,0,1); }
const double MAX=1.5;

domain R(P(0,0), P(1.25,0.5), mesh(6,24), mesh(12,60));

int main()
{
  bounding_box(P(-MAX,-MAX),P(MAX,MAX));
  unitlength("1in");
  picture(2.5,2.5);

  begin();
  revolutions();    viewpoint(4,-2,3);

  pen(0.15);  rgb(0.7,0.7,1);
  plot(F, R);

  plain();  blue();
  plot(F, R.resize2(0.5,1));

  black();  pen(0.5);
  arrow(P(0,0,0), 2*E_1);
  arrow(P(0,0,0), 2*E_2);
  arrow(P(0,0,0), 1.5*E_3);

  masklabel(2*E_1, P(0,0), "$\\mathrm{Re}\\,z$", r);
  label(2*E_2, P(0,0), "$\\mathrm{Im}\\,z$", r);
  label(1.5*E_3, P(2,0), "$\\mathrm{Re}\\,\\sqrt{z}$", r);

  bold(); red();
  plot(g, -1.25, 1.25, 40);
  end();
}
```

Figure 17: Two sheets of the Riemann surface of $\sqrt{z}$.

Gallery

The input files for the figures below are in the `samples` directory of the source tree.