Figure Design and Drawing Tips with ePiX

1 Overview

LaTeX has a relatively simple picture environment permitting line figures to be drawn. External packages such as eepic, PSTricks, and tikz substantially expand the picture environment's capabilities. ePiX adds layers of abstraction, providing a 3-dimensional "world", camera, and Cartesian 2-dimensional drawing region with page layout capabilities; high-level commands for common objects such as shapes, coordinate axes, and function graphs; and the programming capabilities of C++, including variables, loops, recursion, control structures, extensible data types, and numerics.

ePiX is not a graphical drawing program, but instead is more akin to script-based languages such as Asymptote, MetaPost, or gnuplot. An ePiX input file contains human-readable instructions for describing a scene, just as a LaTeX file contains markup instructions for typesetting a document. The techniques, tricks, and work flow of figure creation are distinctive, and completely different from those used with xfig, dia, inkscape, or similar programs. Particularly, you compose documents in a text editor according to logical structure, not by visual appearance in a graphical window.

1.1 Why Use ePiX?

Many drawing needs are served perfectly well by a point-and-click interface. However, logical structuring permits flexibility and mathematical accuracy while maintaining image quality: fonts matching the document; lines of accurate color, width, and placement; logical relationships expressed in objects' definitions; robustness under changes of scale and aspect ratio.

The programmability of ePiX allows you to write figures depending on a small number of constants, whose relationships are encoded by the input file. When one or more constants vary, the entire figure will change smoothly and trivially, as if controlled by mechanical linkages. Noticing these logical relationships within a figure and using them to advantage when writing is a basic skill distinctive to script-driven structured figure design.

1.2 Logical Design

As a quick example, fleshed out below, consider illustrating the classical calculus exercise of finding the point on the parabola $y = x^2$ closest to a specified point $(a, 0)$ on the horizontal axis. To draw the minimizing solution in a point-and-click interface, you'd draw a parabola (or spline approximation), the coordinate axes, and the point $(a, 0)$. Next, you'd visually locate the point (x, y) on the parabola at which the tangent line is normal to the segment joining $(a, 0)$ to (x, y), and draw the normal and tangent lines (or something close enough). Finally, you'd label selected elements in the figure. Parts of the drawing may be inconvenient to arrange, but all can be done acceptably. The screen version may differ from the hard copy, but if the qualitative point is made, you may well feel the result was "good enough". If the initial choice of figure size requires tweaking, you may be in for a moderately tedious edit, or may decide the change isn't worth the trouble. The author has had similar experiences, and the regrettable quality of some published figures suggests he is not alone.

In ePiX, the initial design ignores considerations of appearance in favor of logical relationships. First, create a variable to hold the value a. Second, define a function accepting a

numerical x and returning the point (x, x^2) on the parabola. These data determine the figure; everything else will be computed. The cubic equation in x defining the minimizer (x, y) is easily expressed in terms of a. The real root must be found, so define the appropriate cubic function; `ePiX` can evaluate the root numerically to perfect visual precision. Finally, add drawing commands to the input file, one for the parabola, one for each coordinate axis, one each for the tangent and normal lines, and one for each label. There's no guesswork here, though the labels require a bit of practice to position correctly on the first try. Now compile and preview the figure to see if it "works" aesthetically. If not, the value of a, the size, aspect ratio, line widths, font, colors, etc. can be changed easily in the input file and trivially recompiled into a vector graphic. At the end of the process, we have a LaTeX picture drawn with mathematical precision according to visual criteria.

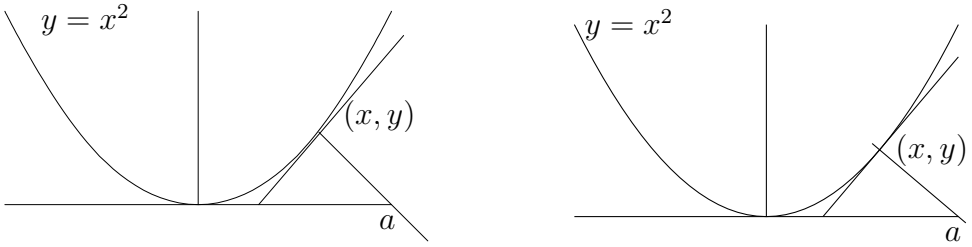
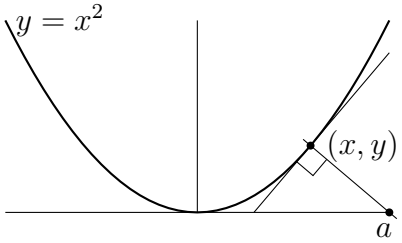


Figure 1: A GUI drawing (left) versus `ePiX`

Superficially, the results are similar, and in this case took roughly comparable amounts of time to produce. LaTeX code had to be added to the labels in the GUI version, and the point of tangency is slightly off due to the author's unsteady hand and inexpensive mouse. More importantly, however, the `ePiX` figure is flexible: It can be resized, redrawn with a different value of a, or colored with trivial effort. A bit of polishing yields a finished figure.



2 Simple Planar Drawing

To draw a figure, you *need* either a rough sketch (hand- or machine-drawn) or a good mental picture of your final image. The author usually employs a mental sketch, and a single `ePiX` file as both rough draft and finished product. A figure containing irregularly-shaped curves should be physically sketched in advance; see Section 4. The overall process is seldom a linear path from blank canvas to complete figure; there are bound to be tentative choices refined as more details get filled in.

2.1 The Preliminary Sketch

True and Virtual Dimensions

The skeleton code inserted by `emacs` into a new `ePiX` file cannot be compiled as is because the size of the figure has not been specified. At the very least, you must give the *true* (printed) size of the figure and the *bounding box* or *virtual* size, the corners of the Cartesian rectangle in which drawing takes place.

The true size is given by a "size string" such as `"5 x 3in"`, a double-quoted string containing a number and an optional LaTeX length unit, an `x` (optionally surrounded by spaces), another number and a mandatory length unit. The bounding box should be given by its lower left (southwest) and upper right (northeast) corners. The command

```
picture(P(a,c), P(b,d), "2in x 6cm");
```

sets the bounding box to $[a,b]\times[c,d]$, and draws the figure 2 in. high and 6 cm wide. (Internally, 2 in. is converted to cm.)

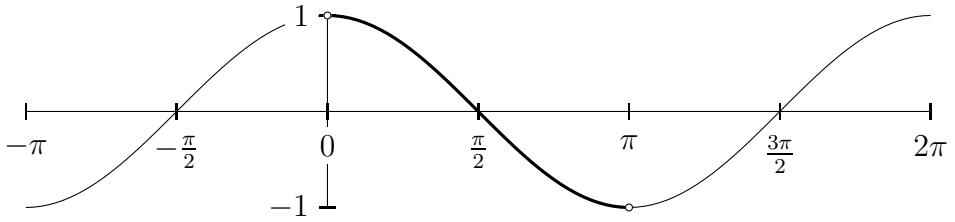
For previewing, you may wish to set the true dimensions larger than the desired size. Dimensions are easy to change so there's no harm setting them approximately and refining the choice later. In 2-dimensional scenes, the choice of bounding box is generally apparent, while bounding a 3-dimensional scene accurately tends to require visual adjustment.

Roughing and Completing the Drawing

An active previewer such as `gv` will be helpful from here on. Add basic elements to the file, compile to encapsulated PostScript (regardless of the final format to be produced), and open the eps file in `gv`. Now you can add commands to the input file, compile from `emacs`, and have the result update in the previewer window. This set-up gives the visual feedback of a GUI environment but retains the precise control furnished by the programming language.

2.2 Layering

Consider a plot of the cosine function; to illustrate branches of the inverse, we'd like to emphasize the "principle branch" part of the graph. The hand drawing might look like:



The bounding box is clearly $[-\pi,2\pi]\times[-1,1]$, but the true size will be dictated by typographical as much as mathematical concerns. If we settle on "true aspect ratio", the width determines the height (or *vice versa*). For the moment, we'll make the figure 1 in. high and therefore 1.5π in. wide. (The author strongly prefers to draw at true aspect ratio. Commercial software sometimes "intelligently" chooses figure dimensions, distorting mathematical truth in the process. `ePiX` allows you to draw visual fibs, but requires you to ask explicitly.)

The dimensions may be specified in two equivalent ways:

```
picture(P(-M_PI,-1), P(2*M_PI,1), "4.7124 x 1in");
```

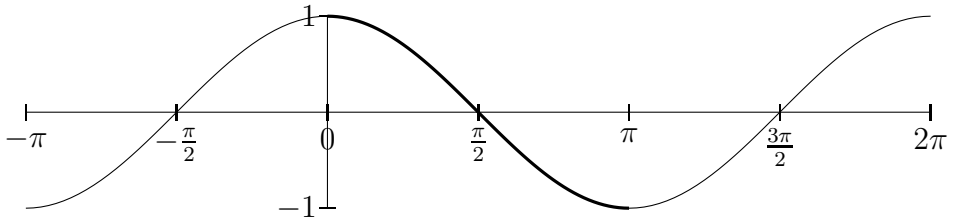
or

```
bounding_box(P(-M_PI,-1), P(2*M_PI,1));
picture(3*M_PI, 2);
unitlength("0.5in");
```

Generally the first form is preferable, but in the present situation the second may be more convenient, as it does not require a numerical value for 1.5π. Adding this content to an `emacs` template creates a compilable, though still empty, input file.

In the present example, we need commands for the axes, a command for the entire graph, and a separate command for the emphasized piece of plot.

```
axis Ax(P(-M_PI,0), P(2*M_PI,0), 6, P(0,-4), b);
Ax.trig().draw();

axis Ay(P(0,-1), P(0,1), 1, P(-4,0), l);
Ay.draw();

plot(Cos, xmin(), xmax(), 120);

bbold();
plot(Cos, 0, M_PI, 40);
```



This is the rough sketch.

Substantial defects are apparent: axis labels colliding with the vertical axis or the graph. Labels can be "masked", printed over a solid white rectangle to hide portions of the figure underneath the text. To repair the overlapping curves and labels, we'll manually order scene elements. Closer inspection reveals a circular annoyance: The vertical axis underlies the horizontal axis labels, which underlie the graph,[1] which underlies the vertical axis labels. Bother. However, `ePiX` allows an axis and its labels to be drawn separately. Thus, we can resolve collisions by first drawing the vertical axis without labels, the horizontal axis with masked labels, the cosine graph, the vertical axis labels, and the open circle markers, in this order.

It's slightly more efficient, though not strictly necessary, to draw the cosine graph in three pieces rather than two. The complete input file now looks like this:

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

[1] As an exercise, draw the horizontal axis with masked labels last.

```
int main()
{
  picture(P(-M_PI,-1), P(2*M_PI,1), "4.7124 x 1in");
  begin();

  label_mask(White()); // set mask color
  axis Ay(P(0,-1), P(0,1), 1, P(-4,0), l);
  Ay.draw_ticks();     // draw axis, ticks only

  axis Ax(P(-M_PI,0), P(2*M_PI,0), 6, P(0,-6), b);
  Ax.trig().draw();    // labeled axis

  plot(Cos, xmin(), 0, 40); // leftmost third
  Ay.draw_labels();    // and the vertical axis labels

  bbold();
  plot(Cos, 0, M_PI, 40);   // middle third, heavy line

  plain();
  plot(Cos, M_PI, xmax(), 40); // right third

  circ(P(0,1));        // open interval markers
  circ(P(M_PI,-1));
  end();
}
```

This file compiles, naturally, to the "hand-drawn" version on page 3.

Collisions between axis labels and other parts of the picture make this example a bit more involved than a typical function plot. When collisions occur and cannot be fixed by moving labels slightly, the rough version requires manual layering. However, since layers are easily rearranged by reordering lines in the input file, final polishing is no more difficult than the layering techniques required to achieve the same effect in a GUI program.

3 Tricks for Discontinuous Functions

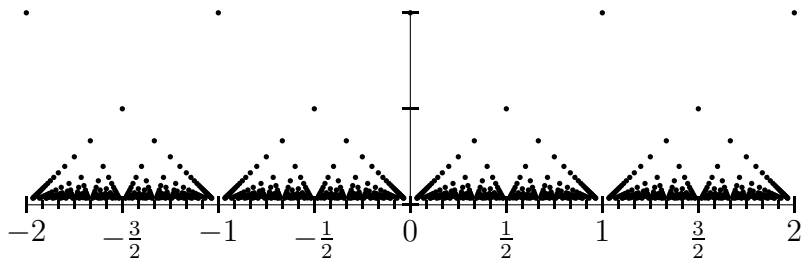
Discontinuous functions defined by formulas (especially in calculus courses) require an assortment of (reasonably intuitive) tricks to be plotted well.

3.1 Removable Discontinuities

There's no difficulty to plotting a function with a removable discontinuity; place a plain `circ` marker on the graph, and a `dot` at the value. To handle more complicated examples, such as Thomae's "denominator" function, use a loop to position points on the graph.

```
/* -*-ePiX-*- */
#include "epix.h"
```

```
using namespace ePiX;

int N(30); // maximum denominator plotted

int main()
{
  picture(P(-2,0), P(2,1), "4x1in");

  begin();

  axis Ax(P(-2,0), P(2,0), 8, P(0,-6), b);
  Ax.subdivide(6).frac().draw();

  v_axis(2);

  for (int i=1; i< N; ++i)
    for (int j=i*xmin(); j <= i*xmax(); ++j)
      if (gcd(i, j) == 1)
        dot(P(j*1.0/i, 1.0/i));

  end();
}
```

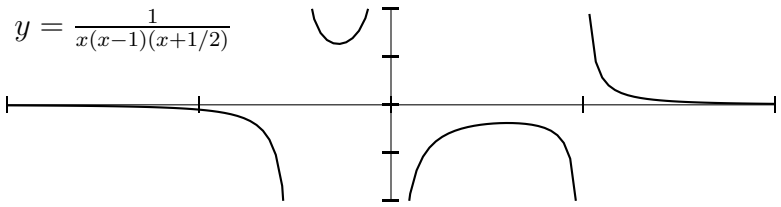


3.2 Poles

Normally ePiX plots by drawing a simple connect-the-dots path. If the plotted function is undefined at a sample point, the adjacent intervals are not drawn. Thus, if a function has a pole (infinite discontinuity), the plot interval and number of samples should be chosen so the pole is *sampled*, not missed. When plotting a function with poles, cropping should be activated. The number of points may require visual adjustment; use as few points as possible to get a visually pleasing plot. (An up-to-date LaTeX distribution should be able to handle many thousands of points.)

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

double f(double x)
{
  return 1.0/((x-1)*x*(x+0.5));
```

```
}

int main()
{
  // strong vertical compression
  picture(P(-2,-20), P(2,20), "4 x 1in");

  begin();

  h_axis(4);
  v_axis(4);

  label(P(xmin(), ymax()), P(2,-2),
        "$y=\\frac{1}{x(x-1)(x+1/2)}$", br);

  set_crop();

  bold();
  plot(f, xmin(), xmax(), 120); // multiple of 8

  end();
}
```

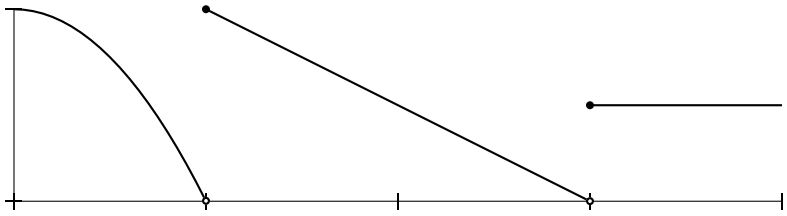


3.3 Jump Discontinuities

"Piecewise" functions may be defined with `if-then` decision statements. To avoid (nearly) vertical segments bridging the discontinuities, use one plot command for each continuous piece, and place appropriate markers at the "breaks".

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

double f(double x)
{
  if (x<1) return (1-x)*(1+x);
  else if (x<3) return 1.5 - 0.5*x;
  else return 0.5;
}

int main()
{
```

```
  picture(P(0,0), P(4,1), "4 x 1in");

  begin();
  h_axis(4);
  v_axis(1);
  bold();
  plot(f, 0, 0.999, 20); // plot on [0,1)
  dot(P(1, f(1)));

  plot(f, 1, 2.999, 1); // linear!
  dot(P(3, f(3)));

  plot(f, 3, 4, 1); // plot on [0,1)

  plain();
  circ(P(1, f(0.999)));
  circ(P(3, f(2.999)));
  end();
}
```

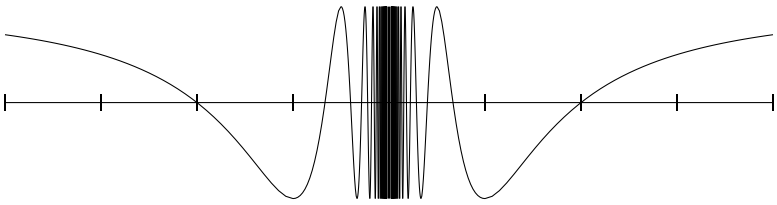


3.4 Wild Discontinuities

Infinitely oscillatory graphs, such as $y = \cos\frac{1}{\pi x}$, test the mettle of any plotting software. Approaching such a plot naively may yield acceptable output, but usually requires far more than a minimal number of sample points. Generally, break the graph into pieces of small curvature, and/or plot over intervals on which the function is monotone.

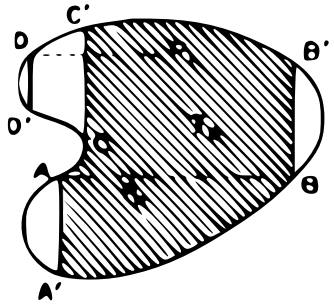
```
/* -*-ePiX-*- */
#include "epix.h"
using namespace ePiX;

double f(double x)
{
  return Cos(1.0/(M_PI*x));
}

int main()
{
  // fibbed aspect ratio
  picture(P(-2,-1), P(2,1), "4 x 1in");
  begin();
```

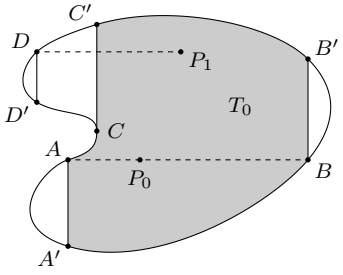
```
  h_axis(8);
  for (int i=1; i<40; ++i)
    {
      plot(f, 1.0/(i+1), 1.0/i, 20);
      plot(f, -1.0/i, -1.0/(i+1), 20);
    }

  plot(f, 1, xmax(), 40);
  plot(f, xmin(), -1.0, 40);

  bbold();
  line(P(0,-1), P(0,1));

  end();
}
```



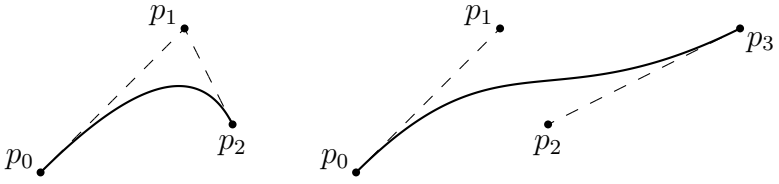
4 Matching a Hand Drawing

In topology or complex analysis, one frequently wishes to draw plane curves lacking a simple parametric description. Such curves can be drawn using splines. The technique below was used to re-create the scanned image (left) from a book on complex analysis as a vector drawing (right).





`ePiX` provides quadratic and cubic splines, given by three or four *control points* p_0, ..., p_3. Each type of spline passes through the initial and final points. At each end, the curve is tangent to the segment from the end point to its neighboring control point:



Approximating a hand-drawn curve therefore amounts to selecting appropriate control points on the curve and adjusting the remaining control point(s) for each spline segment. For the "horseshoe" contour above, each arc between a pair of labeled points is a spline segment.

The first step is to overlay a square grid on the picture being rendered, either using `ePiX` (if an electronic image is available) or a sheet of graph paper. Open a new `ePiX` file, and create points using coordinates read off from the drawing. It's a good idea to name these points systematically, say `p0`,..., `pk` reading along the curve.

To ensure the eventual curve is smooth (has continuous tangents), the control points of adjacent arcs must be chosen compatibly. To this end, define velocity vectors `v0`,..., `vk`, one for each point on the curve, and let `ePiX` compute the control points. At each control point `pi`, place a ruler tangent to the curve, estimate the slope, and let `vi` be a direction vector for the tangent line. Be sure to choose these direction vectors compatibly along the path. (It's often helpful, though not necessary, to choose vectors of length close to unity.)

Now you're ready to lay out a first approximation. If the curve is to be filled or otherwise manipulated as a whole, create a `path` object; otherwise simply invoke the `spline` function once for each arc.

```
path bd(p0, p0 + v0, p1 - v1, p1);
bd += path(p1, p1 + v1, p2 - v2, p2);
bd += path(p2, p2 + v2, p3 - v3, p3); // etc
```

Close and fill the path as needed, and compile the resulting figure.

Chances are the first approximation doesn't adequately resemble the hand drawing. To remedy this, multiply the velocities by suitable constants and recompile. It's enough to adjust the curve arc by arc. After adjustment, the curve might look like this:

```
path bd(p0, p0 + 0.75*v0, p1 - 1.25*v1, p1);
bd += path(p1, p1 + 0.5*v1, p2 - v2, p2); // etc
```

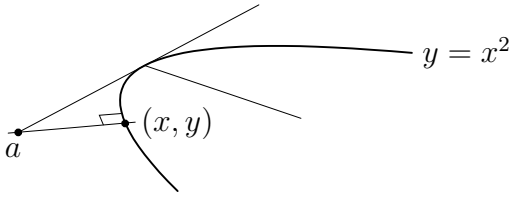
If you're not intuitively familiar with the way splines depend on their control points, it's worthwhile to experiment.

5 Three-Dimensional Figures

All `ePiX` figures are implicitly 3-dimensional. The camera projects orthogonally onto the (x, y)-plane, providing ordinary 2-dimensional Cartesian drawing. Add a line such as

```
camera.at(P(10,8,6));
```

to an input file and recompile; the result will be the original figure viewed from a point in the first orthant.



A naively converted 2-dimensional figure will not be of high quality. Certain elements, primarily axis ticks and labels, are likely not to appear as they would if you viewed a 2-dimensional printout from the camera's spatial location. Further, the bounding box is probably incorrect, perhaps badly so. Naturally, a 3-dimensional figure should be composed from the start with the camera close to its eventual location.

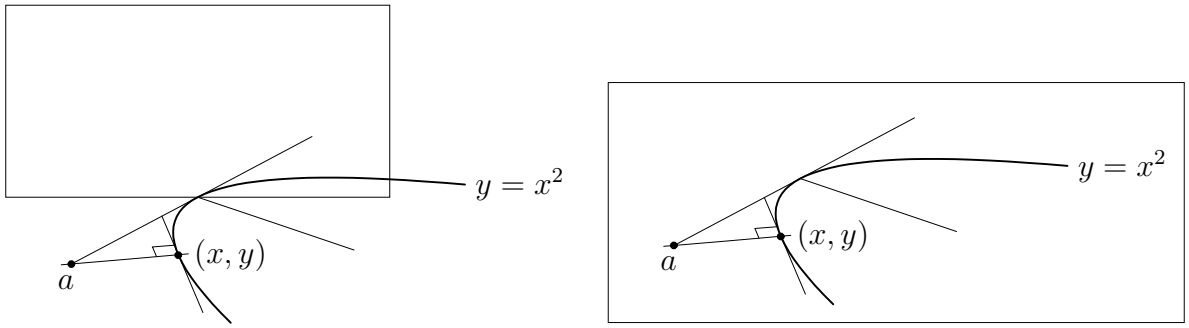
For 2-dimensional scenes, you must distinguish page coordinates (the true size) from Cartesian coordinates (the bounding box). Spatial scenes introduce yet another coordinate system—the 3-dimensional Cartesian coordinates of space, as well as a camera which performs point projection from spatial to planar Cartesian coordinates.

5.1 The Bounding Box

The simplest, most reliable technique to set the bounding box of a 3-dimensional figure is visual adjustment. With experience, you'll be able to estimate where a scene will appear when "photographed" from a given location. For fine-tuning, the command

```
border();
```

draws a thin black rectangle around the bounding box. The corners can then be adjusted to achieve the desired fit, and the border removed.

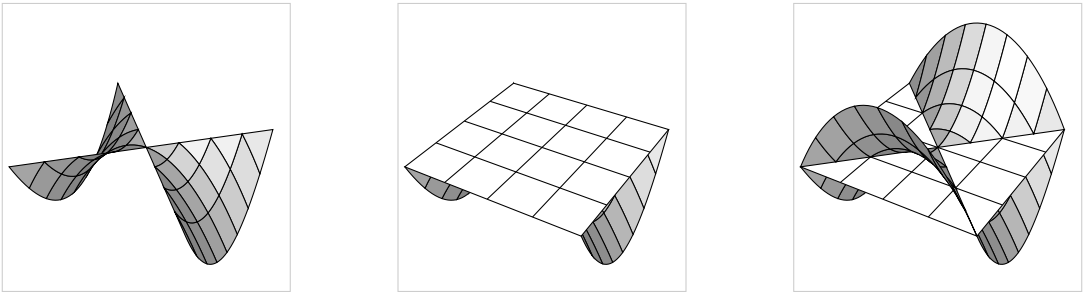


5.2 Object Hiding

In `ePiX`, objects in a scene are drawn in the same order they appear in the input file. This ordering is particularly significant for 3-dimensional scenes: Farther objects must be drawn before nearer objects.

More sophisticated techniques are often necessary. Consider, for example, the problem of graphing a function of two variables along with coordinate axes. Unless the axes, graph, or both are broken into pieces, no ordering of scene elements is likely to be correct. Instead, assuming the camera lies in the region where $z > 0$, one should plot the portion of the graph "below" the (x, y)-plane (where $z < 0$), then the scene elements in the (x, y)-plane, and finally the portion of the graph "above" the (x, y)-plane. `ePiX`'s `clip_box` can be used to chop scenery into pieces for ordering.

```
clip_box(P(-4,-4,-4), P(4,4,0)); // clip to z < 0
<drawing commands>
clip_box(); // restore default
<(x,y)-plane>
clip_box(P(-4,-4, 0), P(4,4,4)); // clip to z > 0
<drawing commands>
```



Similar techniques can be used any time a scene is naturally cut by planes into pieces whose correct ordering depends only on the location of an object with respect to the cutting planes.

If the repeated drawing commands are fairly extensive, the sensible strategy is to collect them into a `void`-valued function, and to call this function each time the clip box is reset.

5.3 Label Positioning

Most objects in a scene are described by their Cartesian position. A simple text label might be given as

```
label(P(3,-2), "Right here");
```

This command puts the payload ("Right here") into a LaTeX box of zero size and places this box at Cartesian location $(3, -2)$. (More accurately, the point $(3, -2)$ is "promoted" to $(3, -2, 0)$ and photographed. This *screen location* is used to position the text box.)

In practice this scheme doesn't give the desired effect; a label goes *near* the object it indicates, not *on top of* it. One's first inclination might be to position the label visually in Cartesian coordinates. A bit of experience shows the inadequacies of the approach: If the viewpoint (camera location) or figure size change, the Cartesian location will not keep the text aligned properly.

For these reasons, `ePiX` labels are positioned by Cartesian location, an offset, and an alignment flag. Think of a label as a Cartesian location with a chunk of LaTeX code attached. The LaTeX code has a *base point*, shown below as a dot. The offset shifts the base point relative to the screen location by a specified amount in true pt. Finally, the alignment flag (`c` for centered, `l` (ell) for left, `tr` for top right, etc.) moves the base point to a corner or midpoint of the LaTeX box. Label alignments in `ePiX` work *opposite* to the way they work in LaTeX; for example, `tr` places the label above and to the right of the location, namely sets the base point to the lower left corner. Each figure below shows the square $[0, 4] \times [0, 4]$; note how label offsets are unaffected by aspect ratio, and that a combination of alignment and offset is required to place labels properly and conveniently.