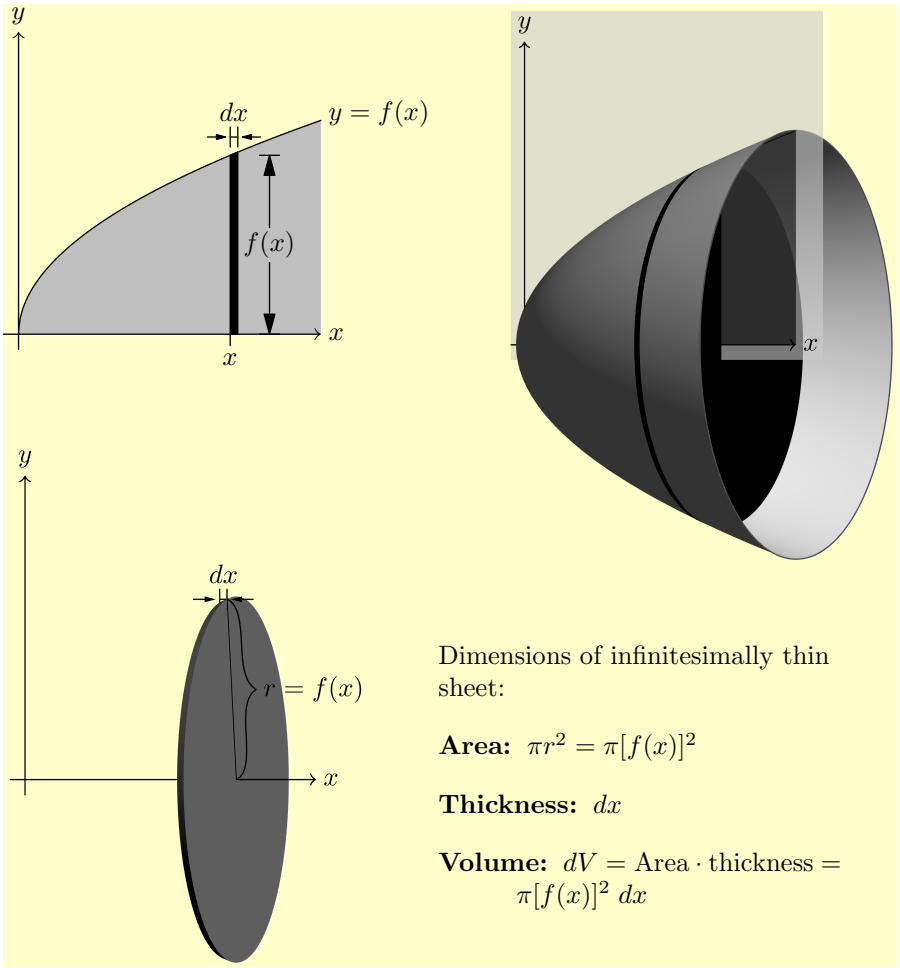
An Asymptote tutorial

Charles Staats III

February 15, 2021

Contents



1 First Steps

1.1 Introduction

Janet is a calculus teacher. She is currently teaching her students how to find volumes of solids of revolution via the "disk method." She would like to produce a diagram to illustrate the method—something like the diagram shown on page 3.

As an experienced user of TikZ, Janet does not think she would have trouble producing the diagram in the top left of the figure. She also believes she could

get the diagram in the bottom left with some fiddling. However, she simply does not believe the three-dimensional capabilities of TikZ are up to drawing a diagram like that in the top right—at least, not without far more time and fiddling than Janet is willing to put in for a single diagram.

Janet's husband Vincent is a programmer. He has some familiarity with the programming language Asymptote, which is especially designed to produce vector graphics and has some fairly substantial three-dimensional capabilities. Working with her husband, Janet decides to try to draw the figure using Asymptote.

1.2 Hello World

Janet already has an up-to-date installation of TeXLive. On the off-chance that this includes an Asymptote installation, she attempts a simple Hello World program. She uses her favorite text editor[1] to produce a document consisting of the single line

```
label("Hello world!");
```

and saves it as `hello_world.asy`. She then goes to the command line and types `asy hello_world`. The result is an eps file named `hello_world.eps`. Opening it, she sees a single page with the following printed on it:[2]

Hello world!

Thus, to the surprise of both Janet and her husband, it appears that Asymptote is already installed on her computer.[3]

Since Janet uses `pdflatex`, she finds it annoying to import eps files, and would prefer that the "graphic" be output in another format. Adding one line to her Asymptote file causes it to output a pdf file instead:

```
settings.outformat = "pdf";
label("Hello world!");
```

Vincent notes that every line should end with a semicolon. Since TikZ behaves the same way, Janet does not find this too difficult to remember.

Now that Janet has a pdf file containing her "graphic," she decides to import it into a latex file. Having done so, she is pleased to notice that Hello World is printed in the same font as the rest of her document (Computer Modern). However, she considers it unfortunate that the font size in the "graphic" is larger than in the rest of her document. Vincent asks her what size she would like

[1] She is inclined to use TeXShop, since she already has it, but Aquamacs would really be more suitable. Vincent, who is a die-hard fan of the command line, recommends using the command-line editor `nano`. On a Windows machine, the simplest thing to use would be Notepad.

[2] Not including the yellow background, of course.

[3] This was more or less my experience, working on Mac OS X. If you are not fortunate enough to have had this miracle happen to you, see Appendix B and the installation instructions in the Asymptote manual.

the font in her Asymptote graphics. Being told that the desired font size is 10 points, he proposes the following:

```
settings.outformat = "pdf";
defaultpen(fontsize(10pt));
label("Hello world!");
```

The result is

Hello world!

which looks nicer with the rest of the document.

1.3 Interpreting the documentation

The `label()` command used in the Asymptote code above is described in Section 4.4 of the Asymptote manual, with the following declaration:

```
void label(picture pic=currentpicture, Label L, pair position,
           align align=NoAlign, pen p=currentpen,
           filltype filltype=NoFill)
```

Janet finds this a bit overwhelming. Vincent suggests that she ignore everything with an `=` sign in it, since those are all optional arguments, and think of such a declaration simply as

```
label(Label L, pair position);
```

Essentially, if Janet wants to use this command to add a label to her picture, she should tell what Label to add and where to add it. For instance, the line

```
label("Hello world", (0,0));
```

would add the text "Hello world" to the picture at position $(0, 0)$. Given this description, Janet wonders why the program she already wrote worked; shouldn't she have been required to specify a position? Vincent confirms that according to the documentation, the line `label("Hello world");` without any position should not have worked. He admits that the documentation is sometimes not completely accurate, which Janet does not find encouraging.

It seems rather peculiar to Janet that the equals sign should be what indicates that an argument can be ignored. Vincent explains that the equals sign provides a default value; when there is a default value, the program knows what to do if the user does not specify a value.

Here are the optional arguments for the `label` command:

type	name	default value
`picture`	`pic`	`currentpicture`
`align`	`align`	`NoAlign`
`pen`	`p`	`currentpen`
`filltype`	`filltype`	`NoFill`

These optional arguments behave something like key-value assignments. For instance, if Janet had wanted to change the text size of just the single line, rather than the entire picture, she could have set the key `p` to value `fontsize(10pt)` as follows:

```
settings.outformat = "pdf";
label("Hello world", p = fontsize(10pt));
```

Note that `fontsize(10pt)` is an object of type `pen`, just as `"Hello world"` (including the quotation marks) is an object of type `Label`.[4] The output is the same as before, since the picture has only one piece of text:

Hello world

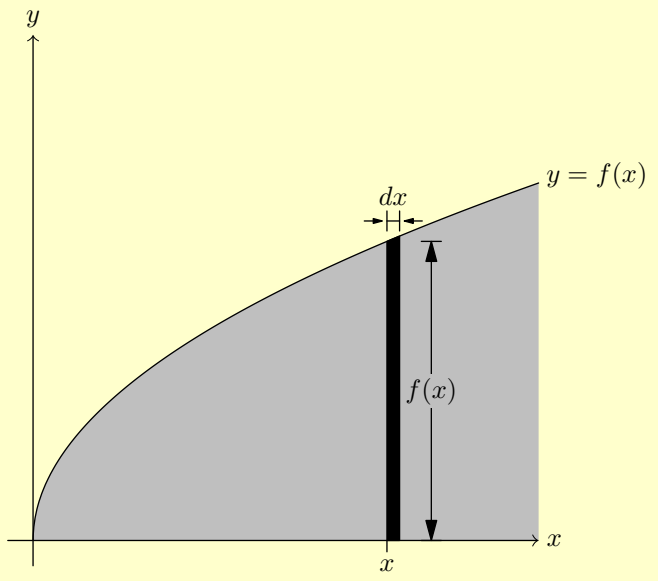
Janet asks how to define a single key that can set several others, as in TikZ styles. Vincent says that this is not possible in Asymptote, although he can see how it might be useful.

2 Drawing a two-dimensional image

2.1 Lines and sizing

Having determined that Asymptote is already installed on her computer, Janet decides to use it to draw a picture of the two-dimensional region that will be revolved. She could do this using TikZ, but Vincent recommends that she get some basic practice drawing with Asymptote before tackling a three-dimensional picture. Here is, roughly, what Janet would like:

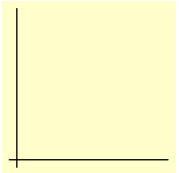
[4] Technically, `"Hello world"` is of type `string`, but strings can be treated as Labels in Asymptote.



First of all, she tries drawing the x-axis as a line from $(-0.1, 0)$ to $(2, 0)$, and the y-axis as a line from $(0, -.1)$ to $(0, 2)$;

```
settings.outformat="pdf";
draw((-.1,0) -- (2,0));
draw((0,-.1) -- (0,2));
```

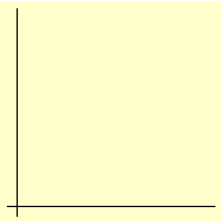
The result is a mark that is barely visible because it is so short. Vincent lets Janet know this is because, by default, Asymptote interprets one unit to mean one point—roughly 0.035 centimeters. Janet thinks that the TikZ default of 1 centimeter is much more reasonable. This can be arranged by adding the line `unitsize(1cm);` to the code:



```
settings.outformat="pdf";
unitsize(1cm);
draw((-.1,0) -- (2,0));
draw((0,-.1) -- (0,2));
```

The final product should be larger still, but is kept this size for now to save space.

There's one more kind of sizing option for Asymptote: the `size` command. In its simplest usage, this command takes a single length for an argument—in the code below, `3cm`—and makes the final picture as large as possible, keeping the same height-to-width ratio, such that neither the width nor the height exceeds the specified dimension (3 centimeters).



```
settings.outformat="pdf";
size(3cm);
draw((-.1,0) -- (2,0));
draw((0,-.1) -- (0,2));
```

The `size` command has several important variations:

- `size(real, real)` takes two dimensions—a maximum width and a maximum height, in that order. Thus, for instance, code that begins with the command `size(2cm, 3cm);` will ordinarily produce a picture that is either 2 centimeters wide (and $\leq$ 3 centimeters tall) or a picture that is 3 centimeters tall (and $\leq$ 2 centimeters wide). In either case, the height-to-width ratio is preserved.
- If either argument in `size(real, real)` is zero, it is ignored. Thus, for instance, a picture that includes the command `size(4cm, 0);` will ordinarily be scaled so that the width is exactly 4 centimeters. The height will be the "natural" height for this scaling factor.
- The command `size(real, real, keepAspect=false);` will scale the width and height independently so that the resulting picture has exactly the specified width and height, but the height-to-width ratio is allowed to change. Thus, for instance, if a circle is drawn on a picture that has this type of size command, the circle is likely to end up looking like an ellipse.

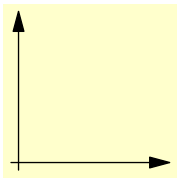
One of Janet's frustrations with TikZ has been that it is difficult to produce a picture that has exactly the desired width (or height). She is gratified to learn that this will be much easier with Asymptote.

2.2 Arrowheads

With the axis lines drawn, Janet thinks that they should have arrows indicating the directions. Vincent agrees that the axes should have arrows. Janet's students do not care about arrows.

Arrows can be added on the end of the line by using the optional parameter `arrow` in the draw command:

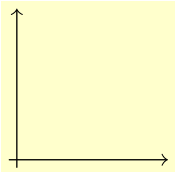
```
settings.outformat="pdf";
unitsize(1cm);
draw((-.1,0) -- (2,0), arrow=Arrow);
draw((0,-.1) -- (0,2), arrow = Arrow);
```



Vincent thinks this looks fairly nice. Janet wants to imitate the TEX-style arrowheads →, as she is used to being done in TikZ. Looking in the Asymptote manual, she and Vincent find the following styles for arrowheads:

Asymptote code	appearance
`draw((0,0)--(1,0),arrow=Arrow());`	
`draw((0,0)--(1,0),arrow=ArcArrow());`	
`draw((0,0)--(1,0),arrow=Arrow(SimpleHead));`	
`draw((0,0)--(1,0),arrow=ArcArrow(SimpleHead));`	
`draw((0,0)--(1,0),arrow=Arrow(HookHead));`	
`draw((0,0)--(1,0),arrow=ArcArrow(HookHead));`	
`draw((0,0)--(1,0),arrow=Arrow(TeXHead));`	

The last, the `TeXHead` style, is more what Janet has in mind:

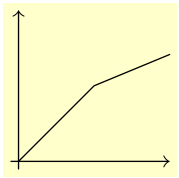


```
settings.outformat="pdf";
unitsize(1cm);
draw((-.1,0) -- (2,0), arrow=Arrow(TeXHead));
draw((0,-.1) -- (0,2), arrow = Arrow(TeXHead));
```

2.3 Curved paths

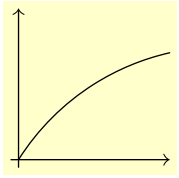
Next, Janet would like to draw the parabola function, which is supposed to look something like the graph of $y = \sqrt{x}$. Here's a first attempt, with a path through the three points $(0, 0)$, $(1, 1)$, and $(2, \sqrt{2})$:

```
settings.outformat="pdf";
unitsize(1cm);
draw((-.1,0) -- (2,0),
    arrow=Arrow(TeXHead));
draw((0,-.1) -- (0,2), arrow =
    Arrow(TeXHead));
draw((0,0) -- (1,1) -- (2,sqrt(2)));
```



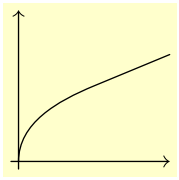
This does not look very nice at all. Substituting `..` for the connector `--` can at least make the path look smooth:

```
settings.outformat="pdf";
unitsize(1cm);
draw((-.1,0) -- (2,0),
    arrow=Arrow(TeXHead));
draw((0,-.1) -- (0,2), arrow =
    Arrow(TeXHead));
draw((0,0) .. (1,1) .. (2,sqrt(2)));
```



While this is a significant improvement, Janet would also like to make the tangent at the origin vertical. This can also be specified:

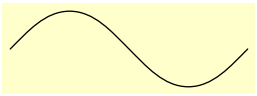
```
settings.outformat="pdf";
unitsize(1cm);
draw((-.1,0) -- (2,0),
    arrow=Arrow(TeXHead));
draw((0,-.1) -- (0,2), arrow =
    Arrow(TeXHead));
draw((0,0){up} .. (1,1) ..
    (2,sqrt(2)));
```



This looks more or less like what Janet had in mind.

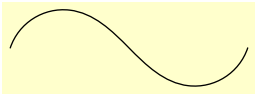
Note that `up` is really just short for `(0,1)`. If Janet wanted a direction other than `up` (or `down`, `right`, or `left`, which are similar), she could specify the tangent direction explicitly as an ordered pair. For instance, here is a rough approximation of a sine curve:

```
settings.outformat = "pdf";
unitsize(0.5cm);
draw((0,0){(1,1)} .. {right}(pi/2,1)
      .. {(1,-1)}(pi,0) ..
    {right}(3*pi/2,-1)
      .. {(1,1)}(2*pi, 0));
```



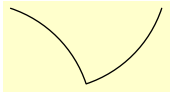
Without the tangent directions specified, Asymptote does a very nice job of connecting the dots to form a smooth curve, but it doesn't really look like a sine curve. The ends in particular are much too steep:

```
settings.outformat = "pdf";
unitsize(0.5cm);
draw((0,0) .. (pi/2,1) .. (pi,0)
      .. (3*pi/2,-1) .. (2*pi, 0));
```



Except at the first and last points, the tangent direction at a point can be specified either before or after the point—or, if a corner is desired, at both:

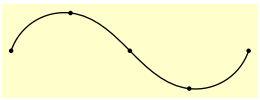
```
settings.outformat = "pdf";
unitsize(0.5cm);
draw((0,2) .. {(1,-3)}(2,0){(1,1/3)}
    .. (4,2));
```



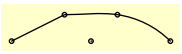
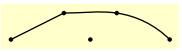
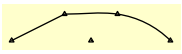
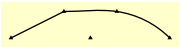
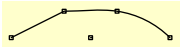
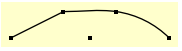
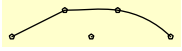
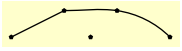
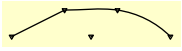
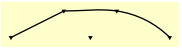
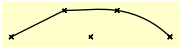
2.4 Markers on paths

If Janet wants to see what the points on a path were actually specified, she can use the `marker` option to the `draw` command:

```
settings.outformat="pdf";
unitsize(0.5cm);
draw((0,0) .. (pi/2,1) .. (pi,0)
      .. (3*pi/2,-1) .. (2*pi, 0),
      marker=MarkFill[0]);
```



Designing your own markers is not that difficult, but requires more knowledge than is currently available. Here are the built-in markers:

built-in option	description	appearance
`Mark[0]`	open circle	
`MarkFill[0]`	filled circle	
`Mark[1]`	open triangle	
`MarkFill[1]`	filled triangle	
`Mark[2]`	open square	
`MarkFill[2]`	filled square	
`Mark[3]`	open pentagon	
`MarkFill[3]`	filled pentagon	
`Mark[4]`	open triangle (upside down)	
`MarkFill[4]`	filled triangle (upside down)	
`Mark[5]`	x-mark	
`Mark[6]`	asterisk	

2.5 Circles and ellipses

The path

```
unitcircle
```

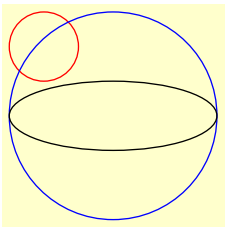
is a unit circle. The function

```
path circle(pair c, real r);
```

returns a circle centered at `c` with radius `r`. The function

```
path ellipse(pair c, real a, real b);
```

produces an ellipse centered at `c` with horizontal diameter `2a` and vertical diameter `2b`.



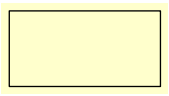
```
settings.outformat="pdf";
size(3cm);
draw(circle((0,1), 0.5), red);
draw(circle((1,0), 1.5), blue);
draw(ellipse((1,0), 1.5, 0.5));
```

2.6 Boxes and polygons

The function

```
path box(pair a, pair b);
```

returns a cyclic path that is a rectangle of which `a` and `b` are opposite corners:

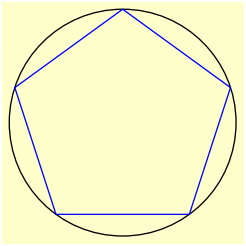


```
settings.outformat="pdf";
unitsize(1cm);
draw(box((0,0), (2,1)));
```

The function

```
path polygon(int n);
```

returns a cyclic path that is a regular polygon with `n` sides, all of whose corners lie on the unit circle:

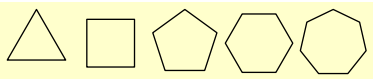


```
settings.outformat="pdf";
unitsize(1.5cm);
draw(unitcircle);
draw(polygon(5), blue);
```

2.7 Transformations: shifting, scaling, rotating, etc.

Janet observes that the `polygon` function above seems to be quite limited. What happens if she wants to draw a polygon at a different position, or a different size? Or upside down? Obviously, she could simply construct the path directly, but she thinks that the `polygon` function ought to allow for more flexibility.

After consulting the documentation, Vincent realizes that such flexibility is not necessary: Janet can still change the path after it has been created, but before it is drawn, using the `transform` type. Here's an example of using a `shift` transform to draw several polygons side by side:

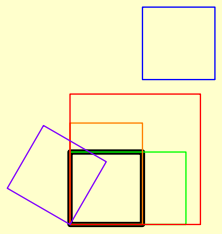


```
settings.outformat="pdf";
size(5cm);
for (int n = 3; n <= 7; ++n) {
    draw(shift(2.2*n, 0) *
    polygon(n));
}
```

Note also the use of a `for` loop to repeat the same code multiple times.

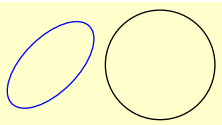
Warning: Since Vincent is an experienced programmer in C-like languages, his first instinct is to use `n++` instead of `++n` to increment `n`. Unfortunately, this *does not work* in Asymptote; the creators decided to omit it because the corresponding `n--` notation would interfere with the use of the notation `p -- q` to indicate a line segment.

Here is some code demonstrating a number of useful transforms:



```
settings.outformat = "pdf";
size(3cm,0);
path p = box((0,0), (1,1));
draw(p, black + linewidth(2.0pt));
draw(shift(1,2)*p, blue);
draw(xscale(1.6)*p, green);
draw(yscale(1.4)*p, orange);
draw(scale(1.8)*p, red);
draw(rotate(60)*p, purple); /*Rotate 60
    degrees*/
```

Transforms can be composed with one another using the `*` operator. For instance, to halve the height of a path, rotate it by 45°, and translate it two to the left (in that order), you can do the following:



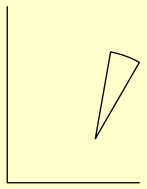
```
settings.outformat = "pdf";
size(3cm,0);
path p = unitcircle;
draw(p, black);
path q = shift(-2,0) * rotate(45) *
    yscale(0.5) * p;
draw(q, blue);
```

2.8 Arcs and margins

The function

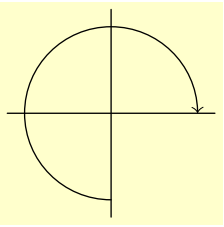
```
path arc(pair c, real r, real angle1, real angle2);
```

creates an arc centered at `c` with radius `r` from `angle1` to `angle2` specified in degrees:

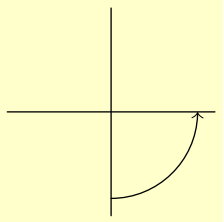


```
settings.outformat="pdf";
size(2cm,0);
draw((3,0)--(0,0)--(0,4));
draw((2,1) -- arc((2,1), 2, 60, 80) -- cycle);
```

The arc goes counterclockwise if `angle1 < angle2`, clockwise otherwise:



```
settings.outformat="pdf";
size(3cm,0);
draw((-1.2,0)--(1.2,0));
draw((0,-1.2)--(0,1.2));
/* An arc from 270 to 0 goes clockwise. */
draw(arc((0,0), r=1, angle1=270, angle2=0),
    arrow=Arrow(TeXHead));
```

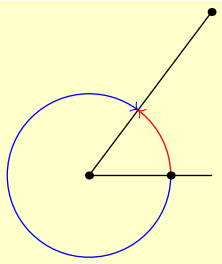


```
settings.outformat="pdf";
size(3cm,0);
draw((-1.2,0)--(1.2,0));
draw((0,-1.2)--(0,1.2));
/* An arc from -90 to 0 goes
    counterclockwise. The same effect could be
    achieved by drawing an arc from 270 to
    360. */
draw(arc((0,0), r=1, angle1=-90, angle2=0),
    arrow=Arrow(TeXHead));
```

There is another useful function for drawing arcs:

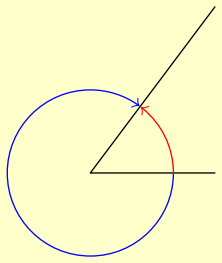
```
path arc(pair c, explicit pair z1, explicit pair z2,
    bool direction = CCW);
```

This function produces an arc centered at `c`, starting at the point `z1` and ending on the line from `c` to `z2`. The direction is specified by either `direction = CW` (clockwise) or the default `direction=CCW` (counterclockwise). This function can be quite convenient for specifying an arc from one line to another:



```
settings.outformat="pdf";
size(3cm,0);
draw((3,0) -- (0,0) -- (3,4));
draw(arc((0,0), (2,0), (3,4)),
    arrow=Arrow(TeXHead), red);
draw(arc((0,0), (2,0), (3,4), direction=CW),
    arrow=Arrow(TeXHead), blue);
dot((0,0));  dot((2,0));  dot((3,4));
```

While this looks good at first glance, Janet is distressed that upon magnification, the two arrow tips both cross into the black line rather than stopping at its edge. Here is the code to fix this using the optional parameter `margin=` for the `draw()` command:

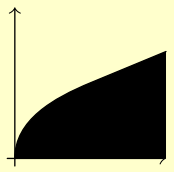


```
settings.outformat="pdf";
size(3cm,0);
draw((3,0) -- (0,0) -- (3,4));
real linewidth = linewidth(currentpen);

/* A path drawn with margin=ArrowMargins will
    be shortened at the end by 0.5 linewidth
    and at the beginning by the full
    linewidth. */
margin ArrowMargins = TrueMargin(linewidth, 0.5 linewidth);
draw(arc((0,0), (2,0), (3,4)), arrow=Arrow(TeXHead), red,
    margin=ArrowMargins);
draw(arc((0,0), (2,0), (3,4), direction=CW),
    arrow=Arrow(TeXHead), blue, margin=ArrowMargins);
```

The dots have been omitted to show the full effect, which will nevertheless be visible only on close inspection (probably with high zoom).

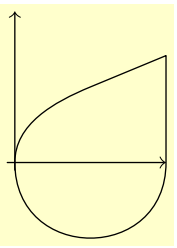
2.9 Filling a region

Next, Janet would like to fill the region under the half-parabola. This may be accomplished by creating a cyclic path and filling it with the `fill` command:



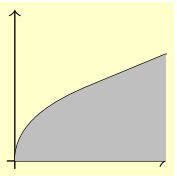
```
settings.outformat="pdf";
unitsize(1cm);
draw((-.1,0) -- (2,0), arrow=Arrow(TeXHead));
draw((0,-.1) -- (0,2), arrow = Arrow(TeXHead));
draw((0,0){up} .. (1,1) .. (2,sqrt(2)));
fill((0,0){up} .. (1,1) .. (2,sqrt(2))
      -- (2,0) -- cycle);
```

Note the use of `cycle` to close the path. Also note that the `..` and `--` operators can be combined to produce a path that is curved some places and straight others. If `.. cycle` were used instead of `--cycle`, the path would be closed smoothly:



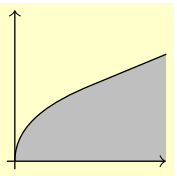
```
settings.outformat="pdf";
unitsize(1cm);
draw((-.1,0) -- (2,0), arrow=Arrow(TeXHead));
draw((0,-.1) -- (0,2), arrow = Arrow(TeXHead));
draw((0,0){up} .. (1,1) .. (2,sqrt(2))
      -- (2,0) .. cycle);
```

Returning to the originally desired picture, Janet really wants it filled with a gray color rather than black. This is not hard to achieve—she can just add an option to the `fill` command specifying what color she wants used.



```
        ⋮
fill((0,0){up} .. (1,1) .. (2,sqrt(2)) -- (2,0)
    -- cycle, mediumgray);
```

Janet almost immediately notices a problem: the filled area is covering other things, including half the arrowhead on the x-axis. This can be fixed by putting the `fill` command earlier, so that the other things get drawn on top of the filled area:

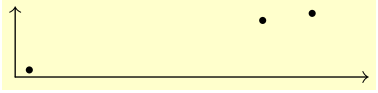


```
settings.outformat="pdf";
unitsize(1cm);
fill((0,0){up} .. (1,1) .. (2,sqrt(2))
     -- (2,0) -- cycle, mediumgray);
draw((-.1,0) -- (2,0), arrow=Arrow(TeXHead));
draw((0,-.1) -- (0,2), arrow = Arrow(TeXHead));
draw((0,0){up} .. (1,1) .. (2,sqrt(2)));
```

Janet thinks that looks much better. Vincent agrees.

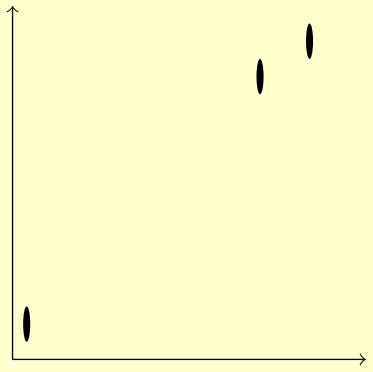
2.10 Drawing a dot at a point

One thing Janet imagines the `fill` command might be useful for is if she wants to draw a point—say, for a scatter plot:



```
settings.outformat="pdf";
size(5cm,5cm);
draw((0,0) -- (50,0), arrow=Arrow(TeXHead));
draw((0,0) -- (0,10), arrow=Arrow(TeXHead));
real r = 0.5;
fill(circle((2,1),r));
fill(circle((35,8),r));
fill(circle((42,9),r));
```

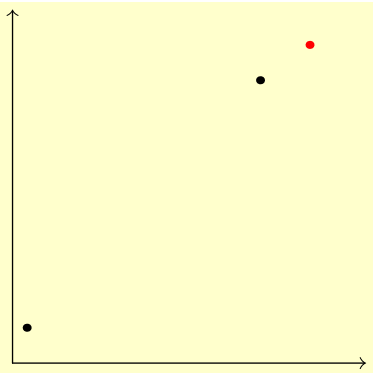
Vincent believes there may be trouble here, however, in case the plot needs to be rescaled. For data plots like this, the actual x and y scales are typically not in the same units, so there is no reason to keep the aspect ratio constant:



```
settings.outformat = "pdf";
size(5cm,5cm, keepAspect=false);
draw((0,0) -- (50,0),
    arrow=Arrow(TeXHead));
draw((0,0) -- (0,10),
    arrow=Arrow(TeXHead));
real r = 0.5;
fill(circle((2,1),r));
fill(circle((35,8),r));
fill(circle((42,9),r));
```

Unfortunately, this highlights a key weakness of drawing points by filling circles: the "points" obtained thus can change size and even shape when the picture is rescaled.

Fortunately, there is a command designed for precisely this sort of thing: the `dot` command, which draws a dot at a specified point that will remain the same size and shape even after rescaling.

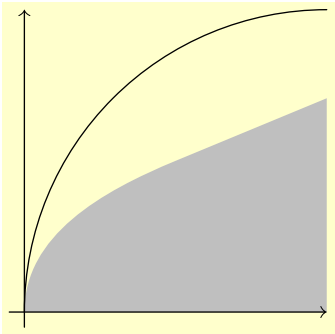


```
settings.outformat = "pdf";
size(5cm,5cm, keepAspect=false);
draw((0,0) -- (50,0),
    arrow=Arrow(TeXHead));
draw((0,0) -- (0,10),
    arrow=Arrow(TeXHead));
dot((2,1));
dot((35,8));
dot((42,9), red);
```

In the picture above, the last dot is drawn in red simply to demonstrate how to do such a thing.

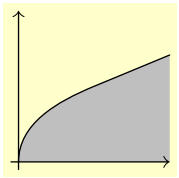
2.11 Named paths and variables

One thing that is starting to bother Vincent is that there is now a certain amount of duplicate code. If Janet were to decide that she wanted to use, say, a quarter-circle rather than a parabola, she'd have to remember to change the path in two different places, or she'd end up with something like this:



Janet thinks she can probably avoid such mistakes, but Vincent insists that even the most experienced computer programmers make mistakes like this unless they take measures to avoid redundant code. Fortunately, in Asymptote, such measures are not difficult: create a single path with a name (like `s`, for instance), and then use the name of the path rather than re-writing it entirely. Here's how this might work for the example in question:

```
settings.outformat="pdf";
unitsize(1cm);
path s = (0,0){up} .. (1,1) .. (2,sqrt(2));
fill(s -- (2,0) -- cycle, mediumgray);
draw((-.1,0) -- (2,0), arrow=Arrow(TeXHead));
draw((0,-.1) -- (0,2), arrow = Arrow(TeXHead));
draw(s);
```



Janet asks Vincent what the word "path" is doing on the third line; why would it not work just to write something like

```
s = (0,0){up} .. (1,1) .. (2,sqrt(2));
```

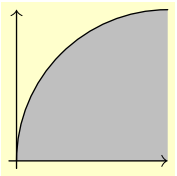
to define `s`? Vincent replies that this is a feature of C-like programming languages, including Asymptote. The letter `s` is something called a *variable*. This basically means that the programmer is allowed to assign—and later reassign—what the symbol `s` means. In this way, it is a lot like a macro in TEX or LATEX. However, unlike macros in TEX, variables in most programming languages have a specified type. Thus, for instance, the variable `s` above has type `path`. One characteristic of C-like languages is that the type of the variable must be specified the first time the variable is used. This is more or less how Asymptote knows that you are creating a new variable rather than re-assigning one that has already been created. If Janet were to omit the word `path` from the declaration

```
path s = (0,0){up} .. (1,1) .. (2,sqrt(2));
```

then Asymptote would assume she was trying to reassign an already existing `path` variable named `s`. If there were no such variable, or if the existing variable were not of type `path`, Asymptote would exit with an error message.

In any case, with the code set up this way, changing the definition of `s` will automatically change both the curve drawn and the region filled:

```
  ⋮
path s = (0,0){up} .. (2-sqrt(2), sqrt(2)) ..
    (2,2);
fill(s -- (2,0) -- cycle, mediumgray);
  ⋮
draw(s);
```

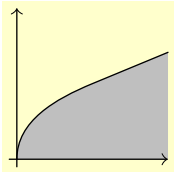


There's at least one more redundancy in the code as currently written: the arrowhead should be the same for both axes. To make sure this happens, Janet decides to create a variable called `axisarrow` and assign it the value `Arrow(TeXHead)`. Vincent tells her the type of this variable should be `arrowbar`.

Janet wonders why the type is called `arrowbar` rather than simply `arrow`. As it turns out, the creators of Asymptote decided that defining a bar on the end of a path (like the one on the left of the arrow $\mapsto$) is quite similar to defining an arrow tip. So, they combined the two into a single type and called it `arrowbar`. Janet will find herself putting both arrows and bars on the end of a line segment before the end of this tutorial.

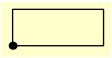
While not an issue of redundancy, the code could also be made more readable by making the ranges of the axes into named variables. The type `real`, short for "real number," is the appropriate type to use for this variable. Vincent notes that this is a departure from conventional C-like languages, which use `double`, primarily for historical reasons.

As long as Janet is deliberately making the code more readable, Vincent suggests that she should also add in some blank lines to group similar kinds of statements together. And since it does not really seem to make much difference whether the curve is drawn before or after the axes, Janet also switches the order to group related commands together. Note, however, that if she were to draw the path `s` before filling it, that *would* make a difference in the appearance of the picture. As a final note, Janet also takes advantage of the `xmax` variable in drawing and filling the path.



```
settings.outformat = "pdf";
unitsize(1cm);

real xmin = -0.1;
real xmax = 2;
real ymin = -0.1;
real ymax = 2;

path s = (0,0){up} .. (1,1) .. (xmax,sqrt(xmax));
fill(s -- (xmax,0) -- cycle, mediumgray);
draw(s);

arrowbar axisarrow = Arrow(TeXHead);
draw((xmin,0) -- (xmax,0), arrow=axisarrow);
draw((0,ymin) -- (0,ymax), arrow = axisarrow);
```

This is probably as good a place as any to mention that another important type in Asymptote is `pair`:

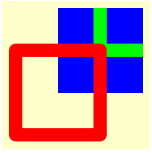


```
settings.outformat = "pdf";
size(1.5cm, 0);
pair botleft = (-1,0);
pair topright = (2.5,1.4);
draw(box(botleft, topright));
dot(botleft);
```

2.12 Clipping a picture

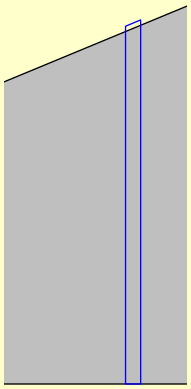
Janet will soon be doing some more detailed work on the image, which will benefit greatly from enlargement. In order to save space in this tutorial, we'll show how to clip graphics. The key is to use the `clip` command, together with a path specifying the area to be clipped. One key difference from TikZ that catches Janet somewhat by surprise is that in Asymptote, the `clip` command clips everything that came *before* it, but not what comes afterwards. (In TikZ, it's the other way around.) Here's an example, in which the blue square indicates the clipped area. The green path, which is drawn before the `clip` command, is clipped; the red path, which is drawn afterwards, is not.

```
settings.outformat="pdf";
size(2cm);
path thebox = box((0,0),(1,1));
fill(thebox, blue);
draw(shift(.5,.5)*thebox,green+linewidth(5pt));
clip(thebox);
draw(shift(-.5,-.5)*thebox,red+linewidth(5pt));
```



2.13 Path times and subpaths

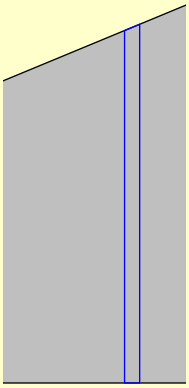
Now, Janet would like to draw the strip that will, in the 3d version, be revolved to make a disk. Here's a first attempt (at the 2d version):



```
settings.outformat="pdf";
unitsize(4cm);

real xmin = -0.1;
real xmax = 2;
real ymin = -0.1;
real ymax = 2;

path s = (0,0){up} .. (1,1) ..
    (xmax,sqrt(xmax));
fill(s -- (xmax,0) -- cycle, mediumgray);
draw(s);

arrowbar axisarrow = Arrow(TeXHead);
draw((xmin,0) -- (xmax,0), arrow=axisarrow);
draw((0,ymin) -- (0,ymax), arrow = axisarrow);

real x = 1.4;
real dx = .05;

draw((x,0) -- (x,sqrt(x)) -- (x+dx,sqrt(x+dx))
     -- (x+dx,0) --  cycle, blue);

clip(box((1,ymin),(1.6,1.5)));
```

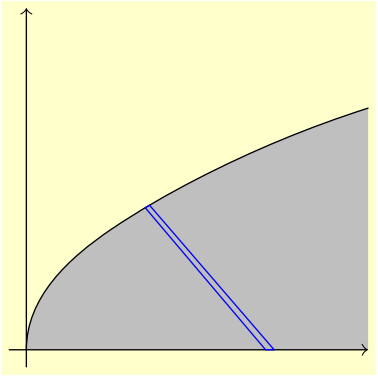
Janet immediately notices that the top of the strip is too high. She realizes that in fact it is the curve that is too low; it's supposed to resemble the graph of $\sqrt{x}$, but clearly it's a bit off. However, before figuring out how to improve the curve, Vincent wants to figure out how to make the top of the strip match the curve that was actually drawn.

A promising-looking command is `subpath(path p, real a, real b);`. According to the documentation, this "returns the subpath of `p` running from path time `a` to path time `b`". Vincent and Janet aren't sure what exactly "path time" means, but they decide to experiment rather than looking it up first:



```
settings.outformat="pdf";
unitsize(4cm);
real xmin = -0.1;
real xmax = 2;
real ymin = -0.1;
real ymax = 2;

path s = (0,0){up} .. (1,1) ..
    (xmax,sqrt(xmax));
fill(s -- (xmax,0) -- cycle, mediumgray);
draw(s);

arrowbar axisarrow = Arrow(TeXHead);
draw((xmin,0) -- (xmax,0), arrow=axisarrow);
draw((0,ymin) -- (0,ymax), arrow = axisarrow);

real x = 1.4;
real dx = .05;
path striptop = subpath(s,x,x+dx);
draw((x,0) -- striptop
     -- (x+dx,0) --  cycle, blue);

clip(box((1,ymin),(1.6,1.5)));
```

At first glance, it appears to work perfectly. Unfortunately, something dreadful happens when Janet inserts the point $(1/2, \sqrt{1/2})$ into the path `s` in an effort to make it more closely resemble the graph of $y = \sqrt{x}$:



```
settings.outformat="pdf";
size(5cm,0);

real xmin = -0.1;
real xmax = 2;
real ymin = -0.1;
real ymax = 2;

path s = (0,0){up} ..
    (1/2,sqrt(1/2)) .. (1,1) ..
    (xmax,sqrt(xmax));
fill(s -- (xmax,0) -- cycle,
    mediumgray);
draw(s);

arrowbar axisarrow = Arrow(TeXHead);
draw((xmin,0) -- (xmax,0), arrow=axisarrow);
draw((0,ymin) -- (0,ymax), arrow = axisarrow);

real x = 1.4;
real dx = .05;
path striptop = subpath(s,x,x+dx);
draw((x,0) -- striptop -- (x+dx,0) --  cycle, blue);
```

Looking through the documentation more closely, Janet and Vincent realize that "path time" is dependent on the number of "nodes" on a path, i.e., the number of points that are specified when the path is defined. Consequently, inserting an extra point to help guide the path drastically changes the path times, even though it does not actually change the shape of the path that much.

2.14 The Law of Janet

After this fiasco, Janet and Vincent formulate the following guideline, and jokingly call it the "Law of Janet":

The Law of Janet. *After a path is created, it should not be used in any way that depends on how it was created.* In particular, if one path is substituted for another with identical appearance, the rest of the code should still produce the same result.

2.15 Intersections and arrays and subpaths

At first glance, the Law of Janet appears to exclude any use of path times—which would be a pity, since a number of useful commands like `subpath` rely on them. However, after further examining the documentation, Vincent realizes that there are other commands that give the path times at which one path intersects

another. By using only path times produced by these commands, it is possible to use subpaths without violating the Law of Janet.

The particular command most useful at the moment is the function `real[] times(path p, real x)` which according to the documentation "returns all intersection times of path `p` with the vertical line through `(x,0)`."

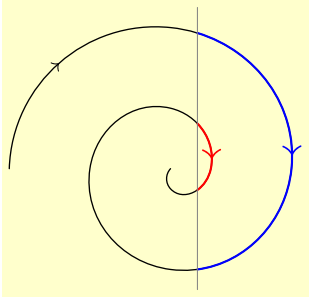
It should be noted that the return type of this function is not simply `real` (a real number), but `real[]`; the function returns an *array* of real numbers. To make sense of this, Janet gets a minimal introduction from Vincent on arrays.

Essentially, an array is an indexed list of objects of a specified type. For instance, a `real[]` (read "a real array") is an index list of `real`s. If a `real[]` is named, for instance, `a`, then the elements of this list can be accessed as `a[0]`, `a[1]`, ..., `a[n-1]`, where `n` is the length of the array. The indices themselves can be variables or expressions. For instance, if `a` is a `real[]`, the code

```
int n = a.length;
real r = a[n-1];
```

will set `r` equal to the last element of the list.

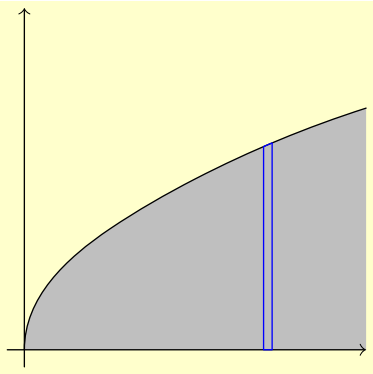
Arrays are useful when a function needs to be able to return more than one value. Since a function can return only one object, it puts all of the values into a single object—an array. In particular, the `times()` function above can return multiple different path times, which can then be plugged into the `subpath()` method. Here's an example:



```
settings.outformat="pdf";
size(4cm,0);

path p = (-2,0) .. (0,7/4) .. (6/4,0)
    .. (0,-5/4) .. (-4/4,0) .. (0,3/4)
    .. (2/4,0) .. (0,-1/4) .. (0,0);
draw(p, arrow=ArcArrow(TeXHead,
    position=0.5));

real[] isections = times(p,1/3);
draw(subpath(p,isections[0],isections[1]), blue+linewidth(0.8),
    arrow=MidArcArrow(TeXHead));
draw(subpath(p,isections[2],isections[3]), red+linewidth(0.8),
    arrow=MidArcArrow(TeXHead));
draw((1/3,-1.5) -- (1/3,2), gray + linewidth(0.2));
```

In the example above, the four points of intersection are specified by the path times `isections[0]`, `isections[1]`, `isections[2]`, and `isections[3]`; and they are arranged in the order they occur as the spiral is traced out.

For what Janet wants, there is only one point of intersection of the path with any vertical line, so what she wants is entry `[0]` of the array returned by `times(s,x)`:

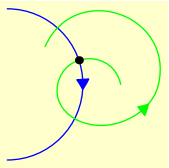


```
settings.outformat="pdf";
size(5cm,0);

real xmin = -0.1;
real xmax = 2;
real ymin = -0.1;
real ymax = 2;

path s = (0,0){up} ..
    (1/2,sqrt(1/2))  .. (1,1) ..
    (xmax,sqrt(xmax));
fill(s -- (xmax,0) -- cycle,
    mediumgray);
draw(s);

arrowbar axisarrow = Arrow(TeXHead);
draw((xmin,0) -- (xmax,0), arrow=axisarrow);
draw((0,ymin) -- (0,ymax), arrow = axisarrow);

real x = 1.4;
real dx = .05;
real t0 = times(s,x)[0];
real t1 = times(s,x+dx)[0];
path striptop = subpath(s,t0,t1);
draw((x,0) -- striptop -- (x+dx,0) --  cycle, blue);
```

There are times when it is desirable to know the intersection of a path with something more general than a vertical line. The most general version of this is to know the intersection points of two paths. The simplest function for this is

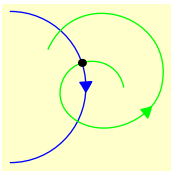
```
real[] intersect(path p, path q);
```

Assuming the two paths `p` and `q` intersect at least once, this function returns an array of length 2. Entry `[0]` gives the path time, along path `p`, of one point of intersection. Entry `[1]` gives the path time along path `q` of the same point. Thus, the two pieces of code below, which are identical except for the last line, give exactly the same result:



```
settings.outformat="pdf";
unitsize(1cm);
path p = (-1,1) .. (0,0) .. (-1,-1);
path q = (1/2,0) .. (-1/3,0) .. (1/2,-1/2) ..
    (1,0) .. (-1/2,1/2);
draw(p,blue, arrow=MidArcArrow());
draw(q,green, arrow=MidArcArrow());
```

```
real[] isections = intersect(p,q);
dot(point(p,isections[0]));
```

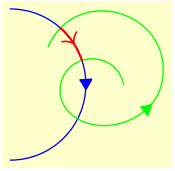


```
settings.outformat="pdf";
unitsize(1cm);
path p = (-1,1) .. (0,0) .. (-1,-1);
path q = (1/2,0) .. (-1/3,0) .. (1/2,-1/2) ..
    (1,0) .. (-1/2,1/2);
draw(p,blue, arrow=MidArcArrow());
draw(q,green, arrow=MidArcArrow());
real[] isections = intersect(p,q);
dot(point(q,isections[1]));
```

Note that the two paths shown above have more than one point of intersection. To get all of them, Janet can use the function

```
real[][] intersections(path p, path q);
```

which returns an array of `real[]`s, i.e., an array of arrays. If the `real[][]` returned is called, say, `a`, then `a[0]` is an array of two real numbers, representing the path times for `p` and for `q` of an intersection point. Significantly, when this function is used, the points are ordered according to which comes first on `p`. Thus, `subpath(p, a[0][0], a[1][0])` will be the subpath of `p` going from the first intersection point to the second intersection point:



```
settings.outformat="pdf";
unitsize(1cm);
path p = (-1,1) .. (0,0) .. (-1,-1);
path q = (1/2,0) .. (-1/3,0) .. (1/2,-1/2) ..
    (1,0) .. (-1/2,1/2);
draw(p,blue, arrow=MidArcArrow());
draw(q,green, arrow=MidArcArrow());
real[][] a = intersections(p,q);
draw(subpath(p,a[0][0], a[1][0]), red+linewidth(0.8),
    arrow=MidArrow(TeXHead));
```

If Janet were only interested in getting the intersection points, rather than path times to use for subpaths, she could use the convenience functions

```
pair intersectionpoint(path p, path q);
```

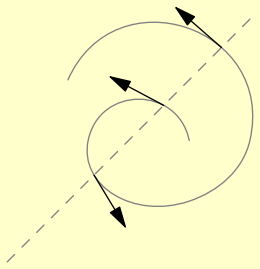
for a single point of intersection, and

```
pair[] intersectionpoints(path p, path q);
```

for a list of all intersection points (in no particular order, or at least none specified by the manual).

2.16 Tangent lines

As a calculus teacher, Janet needs a way to draw a tangent line to a curve, even though that is not needed for this particular diagram. In the past (when using TikZ), her strategy has been to specify the curve by an explicit formula, and differentiate that formula by hand to determine the slope of the tangent line. However, Asymptote has a better way: the function `dir(path p, real t)` returns the unit tangent vector to the path `p` at path time `t`. As previously discussed, path times cannot be used directly without violating the Law of Janet, but they can be constructed from intersections.

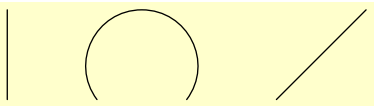


```
settings.outformat="pdf";
size(3.5cm, 0);
path p = (1/2,0) .. (-1/3,0) .. (1/2,-1/2)
    .. (1,0) .. (-1/2,1/2);
path l = (-1,-1) -- (1,1);
draw(l,dashed+gray);
draw(p, gray);
```

```
for (real[] pathtime : intersections(p,l)) {
    real t = pathtime[0];
    pair tangent = dir(p, t);
    draw(shift(point(p,t)) * scale(1/2) * ((0,0) -- tangent),
    arrow=Arrow);
}
```

2.17 Drawing disconnected paths

Technically speaking, an object of type `path` is always connected in Asymptote. Nevertheless, it is possible to form a "disconnected path"—actually an array of paths—using the "pen lift" operator `^^`:



```
settings.outformat="pdf";
size(5cm,0);
draw((0,0) -- (0,1) ^^ (1,0) ..
    (3/2,1) .. (2,0) ^^ (3,0) --
    (4,1));
```

The `draw()` command used here accepts an object of type `path[]` rather than `path`. The relevant drawing command[5] is

[5]Technically, there are two commands. The first accepts as its mandatory argument an `explicit path[]` rather than simply a `path[]`; this means that the command will *not* implicitly convert something else to a `path[]`, even if Asymptote knows how to do the conversion. A second `draw` command with mandatory argument a `guide[]` is also provided, which converts the `guide[]` to a `path[]` by an explicit conversion.

```
void draw(picture pic=currentpicture, path[] g,
          pen p=currentpen, Label legend=null,
          marker marker=nomarker);
```

The only required argument is an array of paths.

Warning: The optional arguments for the `draw(path[])` command are *not* the same as those for the more conventional `draw(path)` command. In particular, there are no arguments for adding labels, arrows, or bars.

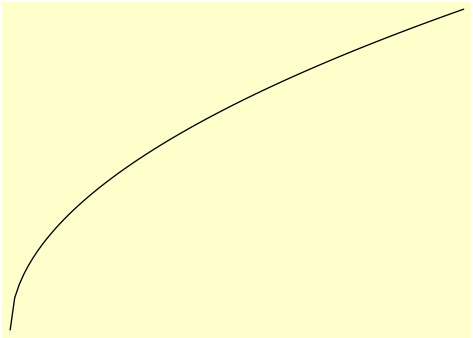
For the purposes of this tutorial, the only important optional argument is the pen `p`, which can be used to set the color and width of the paths drawn (as well as other attributes).

2.18 Graphing functions

Extrapolating a path from four points is very impressive, but Janet wonders if there is a way she could simply tell Asymptote to graph the function $y = \sqrt{x}$ for her. There is—in fact, graphing functions in a somewhat efficient manner is one of the strengths of Asymptote over a purely TeX-based system like TikZ. However, it requires that she use a piece of code that has already been written in Asymptote to do this for her. Fortunately, that is easy enough: in practice, it just means that the line `import graph;` must show up in her file before she can start making graphs. Janet thinks this is a lot like importing a LaTeX package; her husband Vincent thinks it is like using a code library. In any case, this is called importing a *module* in Asymptote.

Here's an example of how she might draw the square root function by graphing it:

```
1 settings.outformat="pdf";
2 unitsize(3cm);
3 import graph;
4 real f(real x) {
5     return sqrt(x);
6 }
7 path g = graph(f,0,2);
8 draw(g);
```

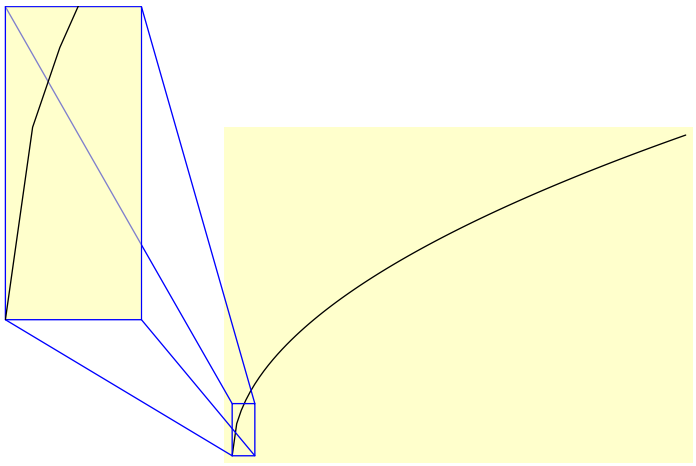


There are several features worth noticing here:

- Lines 4–6 create a function: for the rest of the code, whenever `t` is of type `real` (i.e., a real number), `f(t)` will be interpreted as the real number `sqrt(t)`.

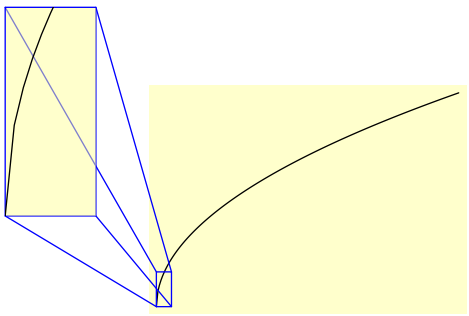
- Line 7 creates, *but does not draw*, the graph of the just-defined function $f\colon \mathbb{R} \to \mathbb{R}$ over the domain $x \in [0, 2]$. The mandatory arguments for the function `graph` are a function and the minimum and maximum x-coordinates to plot.

At first glance, the result does not look bad. However, looking more closely, Janet notices at least one corner, and it bothers her:



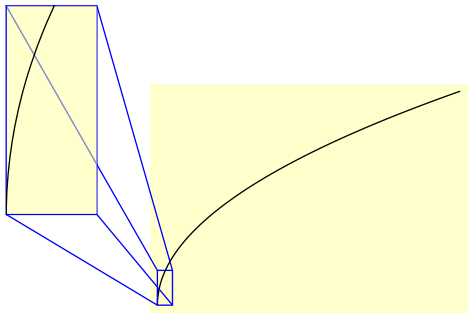
Vincent thinks it's hardly noticeable, but agrees to see what he can find in the manual. As it turns out, there are (at least) two ways to attempt to deal with this by adding optional arguments to the `graph` command. The first is simply to increase the number of points plotted, using a parameter called `n`:

```
settings.outformat="pdf";
unitsize(2cm);
import graph;
real f(real x) {
    return sqrt(x);
}
path g = graph(f,0,2,
    n=200);
draw(g);
```

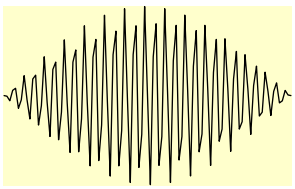


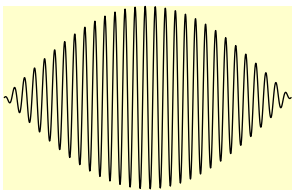
The second is to tell Asymptote to connect the points with a smooth curve rather than with line segments. This is done by adding the argument `operator ..` (The default operator, according to the manual, seems to be `operator --`.)

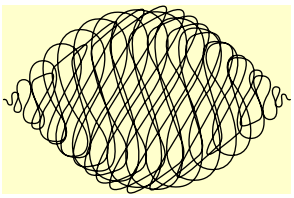
```
settings.outformat="pdf";
unitsize(2cm);
import graph;
real f(real x) {
    return sqrt(x);
}
path g = graph(f,0,2,
    operator ..);
draw(g);
```



In this case, the second method—telling Asymptote to draw a smooth curve—seems to work better. However, that is not always the case. Compare the following three renditions of the graph of $y = \sin x \cos 57x$:







The picture on the right, which was produced using `operator ..`, is clearly inferior to the picture in the middle, which was produced using `n=1000`. Here is the code for the three pictures:

```
settings.outformat
    = "pdf";
import graph;
unitsize(0.9cm);
real f(real x) {
    return sin(x) *
    cos(57*x); }
path g = graph(f,
    0, pi);
draw(g);
```

```
settings.outformat
    = "pdf";
import graph;
unitsize(0.9cm);
real f(real x) {
    return sin(x) *
    cos(57*x); }
path g = graph(f,
    0, pi, n=1000);
draw(g);
```

```
settings.outformat
    = "pdf";
import graph;
unitsize(0.9cm);
real f(real x) {
    return sin(x) *
    cos(57*x); }
path g =
    graph(f,0,pi,
    operator..);
draw(g);
```

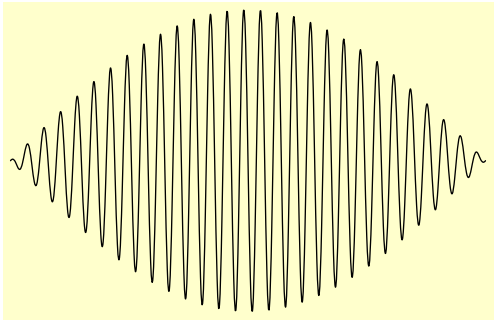
It is, of course, possible to use the option `operator ..` together with the `n=` option, but it is not always wise.

Note: `operator ..` can be problematic for graphing functions because it is designed to take full advantage of the two-dimensional drawing space, whereas functions are supposed to be somewhat limited in how they can move left and right. A better choice in this case is called `Hermite`, which creates curves without producing the awful scribble.

```
settings.outformat = "pdf";
```

```
import graph;
unitsize(2cm);
real f(real x) { return sin(x)*cos(57*x); }
path g = graph(f, 0, pi, n=200, Hermite);
draw(g);
```

The result:



2.19 Parametric graphs

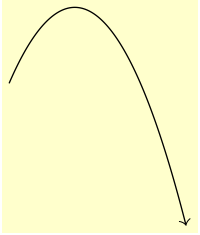
Janet's brother Jason teaches physics. In a particular example problem, a ball is thrown so that its height (in meters) after t seconds is given by

$$\begin{aligned} y(t) &= -\tfrac{1}{2}gt^2 + v_{y,0}t + y_0 \\ &= -4.5t^2 + 3.0t + 1.0 \ ; \end{aligned}$$

the horizontal distance is given by

$$\begin{aligned} x(t) &= v_x t + x_0 \\ &= 1.3t \ . \end{aligned}$$

Jason would like to be able to draw this path (for the first 0.9 seconds) without having to rewrite y in terms of x. He can do this using parametric graphs in Asymptote:



```
settings.outformat="pdf";
unitsize(2cm);
import graph;
pair F(real t) {
    return (1.3*t,  -4.5*t^2 + 3.0*t + 1.0);
}
path g = graph(F, 0, 0.9);
draw(g, arrow=Arrow(TeXHead));
```

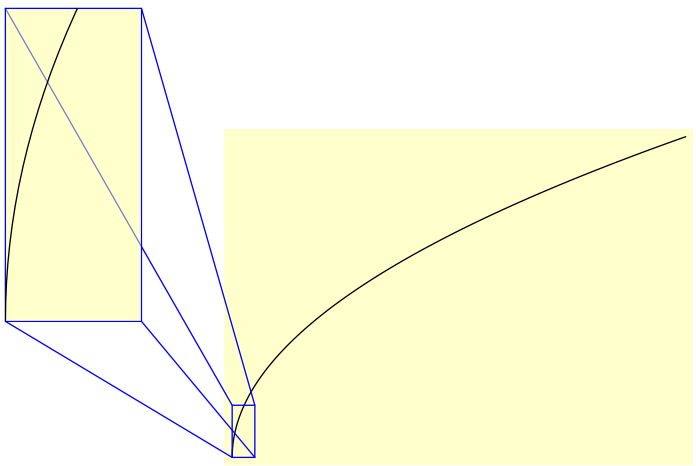
As a calculus teacher, Janet observes that the jagged corner visible in the square root graph on p. 29 is probably caused by the use of points with the x-distances evenly spaced. Since the slope of the function is very high near zero (and infinite at zero), small changes in x produce large changes in y and, consequently, long—and obtrusive—line segments. She wonders if this could be fixed by graphing x as a function of y—i.e., by graphing $x = y^2$ as y ranges from 0 to $\sqrt{2}$, rather than graphing $y = \sqrt{x}$ as x ranges from 0 to 2.

Even more generally, she would like to be able to draw a parametric graph, in which x and y are both given as functions of a third parameter t. Thus, for instance, the graph of

$$(x, y) = (t^2, t)$$

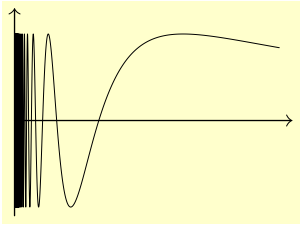
with evenly spaced choices of t might give a nicer-looking result than simply graphing $y = \sqrt{x}$. The `graph` module of Asymptote allows this more sophisticated kind of graphing as well:

```
settings.outformat="pdf";
unitsize(3cm);
import graph;
pair f(real t) { return (t^2, t); }
path g = graph(f, 0, sqrt(2));
draw(g);
```



Even though this graph is actually made up of line segments, the spacing is better so that the corners are far less noticeable.

Here's one last example that uses parametric graphing to plot the function $f(x) = \sin(1/x)$, the "topologist's sine curve," which is notoriously difficult to graph:



```
settings.outformat="pdf";
size(4cm,3cm, keepAspect=false);
import graph;
pair f(real t) {
     return (1/t, sin(t));
 }
draw(graph(f, 1, 10^4, n=5*10^5),
    thin());
draw((0,-1.1)--(0,1.3), arrow=Arrow(TeXHead));
draw((0,0)--(1.05,0), arrow=Arrow(TeXHead));
```

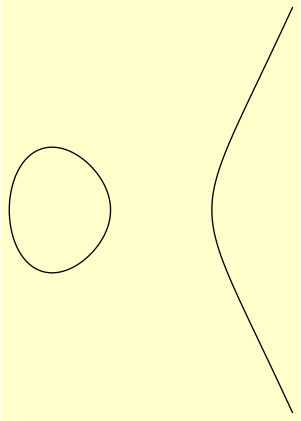
Note the use of the pen `thin()`, which shows up on a screen as being exactly one pixel thick (at every zoom level), making it possible to zoom in and see more of the oscillations before they completely obscure one another as $x \to 0$. For some reason, this particular image also seems to render much more quickly on the computer screen with the `thin()` pen than with a pen whose line width is specified; however, it may not print very nicely.

2.20 Implicitly defined curves; building arrays

Asymptote also has the capability to graph curves that are defined implicitly. This requires the module `contour` rather than `graph`. The relevant function is

```
contour(real f(real, real), pair a, pair b, real[] c);
```

it returns a `guide[][]` (i.e., an array of arrays of `guide`s). If this returned `guide[][]` is saved as a variable names `thegraphs`, then for each `i`, the entry `thegraphs[i]` is a `guide[]` representing the (possibly disconnected) graph of the equation $f(x, y) = \texttt{c}[\texttt{i}]$. For example, the graph of $y^2 = x^3 - x$ (equivalently, $y^2 - (x^3 - x^2) = 0$) restricted to the rectangular region with corners $(-2, -2)$ and $(2, 2)$ may be obtained as follows:



```
settings.outformat="pdf";
size(4cm,0);
import contour;
real f(real x, real y) { return y^2 -
    (x^3 - x); }
guide[][] thegraphs = contour(f,
    a=(-2,-2), b=(2,2), new real[] {0});
/* The next line draws the first (and
    only) entry in thegraphs. This entry
    is itself an array, since it
    represents a disconnected path. */
draw(thegraphs[0]);
```

Warning: If you draw the paths returned by the `contour()` function, you may unwittingly assume that the drawing will automatically include the points `a` and `b`. This assumption is *false*. Drawing the paths alone will only include the smallest rectangle that contains the graphs. If you want to include the entire rectangle with corners `a` and `b`, you must do something extra, such as drawing axes or drawing the disconnected path `a ^^ b` using the pen `invisible`.

Note the use of the expression `new real[] {0}` to build an array containing exactly one entry, namely the real number `0`. This is a general technique for building arrays. For instance, the expression `new pair[] {(0,0), (1,0), (0,1), (0,1)}` produces an array with the four entries $(0,0)$, $(1,0)$, $(0,1)$, $(0,1)$. Likewise, the expression

```
(0,0)--(0,1) ^^ (1,0)..(3/2,1)..(2,0) ^^ (3,0)--(4,1)
```

is essentially[6] equivalent to the expression

```
new path[] {(0,0)--(0,1), (1,0)..(3/2,1)..(2,0), (3,0)--(4,1)}
```

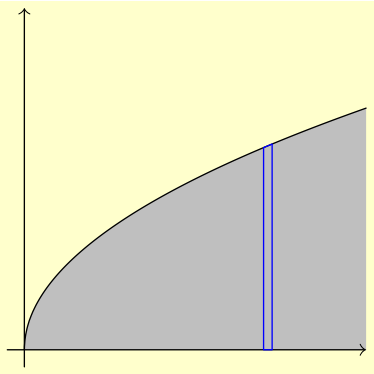
An alternative way to build an array is to declare the array and then add the items to the end one by one using the `array.push()` command. For example, the following code creates an array consisting of all nonzero integers from -5 to 15 (inclusive):

```
int[] myarray;
for (int i = -5; i <= 15; ++i) {
    if (i != 0) myarray.push(i);
}
```

2.21 The filldraw command

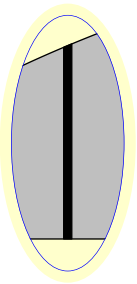
Returning to the original problem, Janet and Vincent have now progressed as far as being able to draw the following:

[6] To make it even more equivalent, change `new path[]` to `new guide[]`. The difference between paths and guides is explained in the official Asymptote documentation in the section "paths and guides" of the Programming chapter.



```
settings.outformat="pdf";
size(5cm,0);
import graph;

real xmin = -0.1;
real xmax = 2;
real ymin = -0.1;
real ymax = 2;

path s = graph(sqrt, 0, 2,
    operator..);
fill(s -- (xmax,0) -- cycle,
    mediumgray);
draw(s);
arrowbar axisarrow = Arrow(TeXHead);
draw((xmin,0) -- (xmax,0), arrow=axisarrow);
draw((0,ymin) -- (0,ymax), arrow = axisarrow);

real x = 1.4;
real dx = .05;
real t0 = times(s,x)[0];
real t1 = times(s,x+dx)[0];
path striptop = subpath(s,t0,t1);
draw((x,0) -- striptop -- (x+dx,0) --  cycle, blue);
```

What Janet actually wanted, for the blue box, was a box filled in black—and perhaps also outlined, for good measure. This can be accomplished by replacing the `draw` command in the final line of the code above with `filldraw`:



```
  ⋮
filldraw((x,0) -- striptop -- (x+dx,0) --  cycle,
    black);
  ⋮
//code to clip picture goes here
```

2.22 Adding text

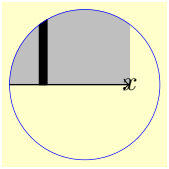
There are a number of different ways to add text to an Asymptote diagram. Probably the simplest is to use the command

```
void label(Label L, pair position);
```

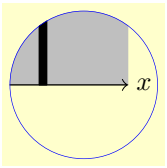
which has optional arguments

type	name	default value
`picture`	`pic`	`currentpicture`
`align`	`align`	`NoAlign`
`pen`	`p`	`currentpen`
`filltype`	`filltype`	`NoFill`

Thus, to place the text "`$x$`" at the point `(xmax,0)`, Janet can add the command `label("$x$",(xmax,0))` to the end of her Asymptote code. Here's the result after clipping:

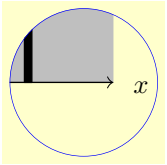


First, Janet notes that the string "`$x$`" was translated exactly into its LaTeX equivalent, which she appreciates. However, there is a more pressing issue: she did not want the text positioned exactly on top of the end of the x-axis, but to the right. For this, she can add the option "`align=E`":




```
label("$x$", (xmax,0), align=E);
```

This is much more what she had in mind. Other standard alignments include `N`, `S`, `W`, `NE`, `SE`, ... It should also be noted that these are really abbreviations for ordered pairs: `E`, for instance, really means `(0,1)`. As such, alignments can be added to each other and multiplied by real numbers:




```
label("$x$", (xmax,0), align=2.5E + S/2);
```

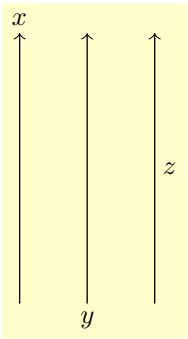
Labels can also be added directly to paths when they are drawn, using the optional parameter `Label L` of the `draw()` command. When this is done, it is typically necessary to construct a Label explicitly using the `Label()` function

```
Label Label(string s);
```

with optional parameters

type	name	default value
`string`	`size`	`""`
`position`	`position`	*no default*[7]
`align`	`align`	`NoAlign`
`pen`	`p`	`nullpen`
`embed`	`embed`	`Rotate`
`filltype`	`filltype`	`NoFill`

The most important optional parameter here is `position=`. Typical values include `position=EndPoint`, `position=MidPoint`, or `position=BeginPoint`. More generally, `position=Relative(r)` will take the real number `r` $\in [0,1]$ to specify the location of the Label as a fraction of the total arclength of the path. For instance, `MidPoint` is defined to be `Relative(0.5)`.




```
settings.outformat = "pdf";
size(2.5cm, 0);
Label Lx= Label("$x$", position=EndPoint);
Label Ly = Label("$y$", position=BeginPoint);
Label Lz = Label("$z$", position=MidPoint);
draw((0,0) -- (0,4), arrow=Arrow(TeXHead),
    L=Lx);
draw((1,0) -- (1,4), arrow=Arrow(TeXHead),
    L=Ly);
draw((2,0) -- (2,4), arrow=Arrow(TeXHead),
    L=Lz);
```

The default alignments are reasonably sensible. However, they can be changed by using the optional paramater `align` in creating the label with `Label`. For instance, the lower of the following two images is more like the one Janet is going to want in her diagram:

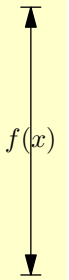

```
settings.outformat="pdf";
unitsize(0.3cm);
Label L = Label("$dx$", position=MidPoint);
draw((0,0) -- (1,0), L=L, bar=Bars);
```


```
settings.outformat="pdf";
unitsize(0.3cm);
Label L = Label("$dx$", position=MidPoint, align=2N);
draw((0,0) -- (1,0), L=L, bar=Bars);
```

[7] An "optional parameter with no default" is obtained by *overloading* the function, i.e., by defining multiple functions with the same name but different arguments.

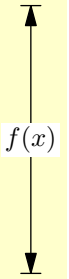
Note the use of the option `bar=Bars` on the draw command to produce bars perpendicular to the ends of the path. The other options for this parameter are `None` (the default), `BeginBar`, `EndBar`, and `Bar` (which is the same as `EndBar`).

Another label that Janet wants to add to her diagram is one indicating the height of the $dx \times f(x)$ strip. For this, she wants a vertical line with arrows and bars on both ends, and with the label $f(x)$ right on top of the midpoint. For the latter effect, the option `align=(0,0)` in the construction of the label is called for:




```
settings.outformat="pdf";
unitsize(3cm);
defaultpen(fontsize(10pt));
Label L = Label("$f(x)$", align=(0,0),
    position=MidPoint);
draw((0,0) -- (0,sqrt(1.4)), L=L, arrow=Arrows(),
    bar=Bars);
```

There is, unfortunately, one more problem: the part of the line behind the label $f(x)$ needs to be hidden for readability. Or, to put it another way, a box around the label needs to be filled, with the same color as the background, to hide the line behind it. Janet can accomplish this by using the option `filltype=Fill(white)` in the construction of the label, where `white` should be replaced by whatever the background color really is.



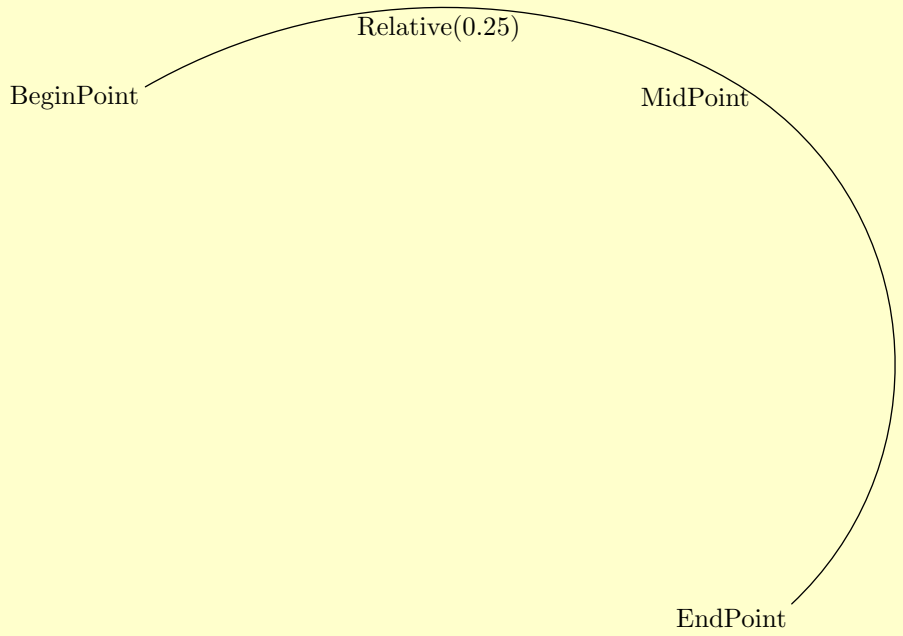

```
settings.outformat="pdf";
unitsize(3cm);
defaultpen(fontsize(10pt));
Label L = Label("$f(x)$", align=(0,0),
    position=MidPoint, filltype=Fill(white));
draw((0,0) -- (0,sqrt(1.4)), L=L, arrow=Arrows(),
    bar=Bars);
```

2.23 Adding multiple labels to a single path

To add multiple labels to the same path, construct the labels separately using `Label` and then apply the method `label(Label L, path g)`:

```
settings.outformat="pdf";
defaultpen(fontsize(10pt));
size(12cm, 0);
path p =(0,0) .. (4,.3) .. (5,-.3) .. (5,-4);
draw(p);
Label L1 = Label("BeginPoint", position=BeginPoint);
```

```
label(L1, p);
Label L2 = Label("MidPoint", position=MidPoint);
label(L2, p);
Label L3 = Label("EndPoint", position=EndPoint);
Label L4 = Label("Relative(0.25)", position=Relative(0.25));
label(L3, p);
label(L4, p);
```

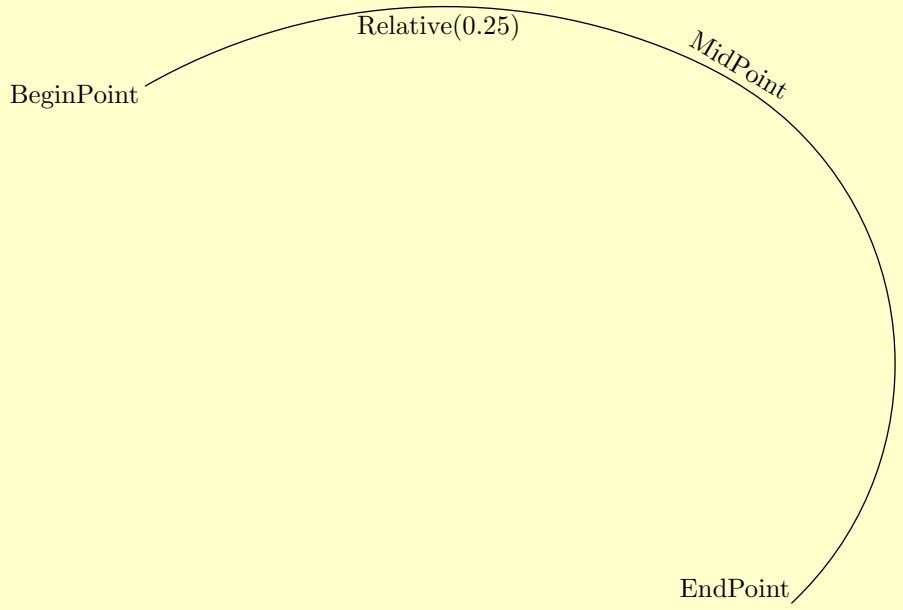


Janet thinks it is unfortunate that it takes two[8] lines of code to add each label to the path, and also that there is no obvious way to make the text of the label slope along the path. After consulting the source code in `plain_Label.asy`, she and Vincent manage to cobble together a new command, which they call `pathlabel`, that seems to alleviate both these issues:

```
void pathlabel(picture pic = currentpicture, Label L, path g,
    real position=0.5, align align=NoAlign, bool sloped=false,
    pen p=currentpen, filltype filltype=NoFill) {
  Label L2 = Label(L, align, p, filltype,
    position=Relative(position));
  if (sloped) {
    pair direction = dir(g, reltime(g, position));
    real angle = degrees(atan2(direction.y, direction.x));
    L2 = rotate(angle)*L2;
```

[8]This can be abbreviated to a single arguably more confusing line of code by defining the `Label` within the `label` command, e.g., `label(Label("MidPoint",position=MidPoint),p);`

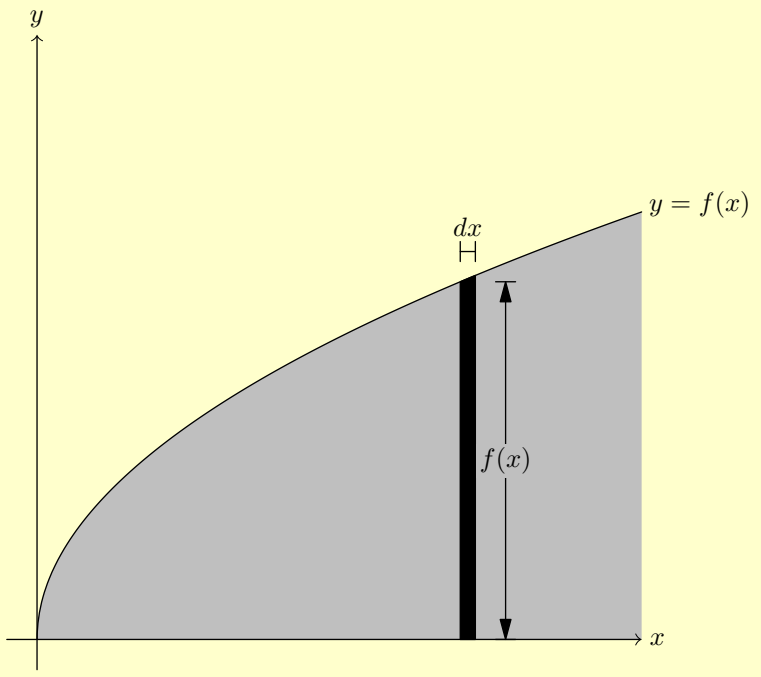
```
  }
  label(pic, L2, g);
}

size(12cm, 0);
defaultpen(fontsize(10pt));
settings.outformat="pdf";
path p =(0,0) .. (4,.3) .. (5,-.3) .. (5,-4);
draw(p);
pathlabel("BeginPoint", p, position=0);
pathlabel("Relative(0.25)", p, position=0.25);
pathlabel("MidPoint", p, position=0.5, align=Relative(W),
    sloped=true);
pathlabel("EndPoint", p, align=Relative(E), position=1.0);
```



Note also the use of the option `align=Relative(pair)`, which tells Asymptote to align the label relative to the path, with "north" pointing in the tangent direction to the path. In particular, `align=Relative(E)` will put the label to the right of the path direction (i.e., below the path, if it is going from left to right), while `align=Relative(W)` will put the label to the left of the path direction (above the path, if it is going from left to right).

2.24 Drawing objects of a fixed (unscalable) size

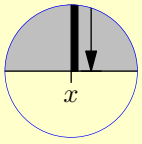
If we combine all that we have done so far toward the desired image, we get the following:



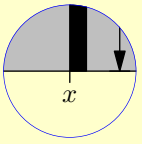

```
settings.outformat="pdf";
unitsize(4cm);
defaultpen(fontsize(10pt));
import graph;

real xmin = -0.1;
real xmax = 2;
real ymin = -0.1;
real ymax = 2;

real f(real x) { return sqrt(x); }
path s = graph(f, 0, 2, operator..);
pen fillpen = mediumgray;
fill(s -- (xmax,0) -- cycle, fillpen);
draw(s, L=Label("$y=f(x)$", position=EndPoint));

real x = 1.4;
real dx = .05;
real t0 = times(s,x)[0];
real t1 = times(s,x+dx)[0];
path striptop = subpath(s,t0,t1);
filldraw((x,0) -- striptop -- (x+dx,0) --  cycle, black);
```

```
real barheight = f(x) + .1;
Label dxlabel = Label("$dx$", position=MidPoint, align=2N);
draw((x,barheight) -- (x+dx, barheight), L=dxlabel, bar=Bars);

real barx = x + dx + 0.1;
Label fxlabel = Label("$f(x)$", align=(0,0), position=MidPoint,
    filltype=Fill(fillpen));
draw((barx,0) -- (barx, f(x)), L=fxlabel, arrow=Arrows(),
    bar=Bars);

arrowbar axisarrow = Arrow(TeXHead);
Label xlabel = Label("$x$", position=EndPoint);
draw((xmin,0) -- (xmax,0), arrow=axisarrow, L=xlabel);
Label ylabel = Label("$y$", position=EndPoint);
draw((0,ymin) -- (0,ymax), arrow = axisarrow, L=ylabel);
```

This is getting close to the desired result, but there are a few missing bits. First of all, Janet wants a tick mark on x-axis which labels the horizontal position of the strip as x. Moreover, this tick should remain the same size—say, 1.5 millimeters long—even if the image is rescaled. Fortunately, there is a variant of the `draw` command that draws a scale-proof object at a scalable location (like $(x, 0)$, which will change if the image is rescaled). This variant is called when the first parameter passed to the `draw` command is an ordered pair. Note that in the two examples below, the tick mark is the same size even though the scaling is different:

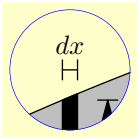


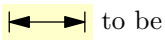
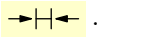

```
size(2cm);

    ⋮

path tick = (0,0) --
    (0,-0.15cm);
Label ticklabel =
    Label("$x$",
    position=EndPoint);
draw((x,0), tick,
    L=ticklabel);
path clippath=circle((x,0),
    0.5);
draw(clippath, blue);
clip(clippath);
```

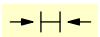



```
size(2cm);

    ⋮

path tick = (0,0) --
    (0,-0.15cm);
Label ticklabel =
    Label("$x$",
    position=EndPoint);
draw((x,0), tick,
    L=ticklabel);
path clippath=circle((x,0),
    0.2);
draw(clippath, blue);
clip(clippath);
```

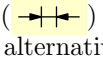
Another detail Janet finds unsatisfying is the way that the dx length is indicated:

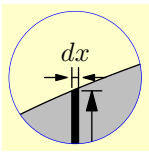


These bars are clearly too close together for a construction like  to be reasonable. To have this look "nicer," Janet would like to imitate a style she saw in a textbook in which arrows pointing in from the either side:  . Furthermore, the lengths of these arrows should be given in absolute measurements that do not change with scaling; say, each arrow should be 3 millimeters long. This could be done using the `draw` command discussed above. However, for something like this, it is easier to use the `arrow` command: `arrow(pair b, pair dir, real length=arrowlength)` will draw an arrow pointing to `b` from direction `dir`, with length `length` specified in absolute (unscalable) units.

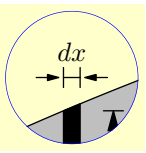


```
settings.outformat="pdf";
size(1cm,0);
draw((0,0) -- (1,0), bar=Bars);
arrow((0,0), W, length=0.3cm);
arrow((1,0), E, length=0.3cm);
```

Unfortunately, Janet thinks that there is a bit too much space at the arrow tips. Vincent consults the documentation and realizes that this is a consequence of the default option `margin=EndMargin`, which is designed for an arrow pointing to a label. Predefined alternatives include using `arrow(..., margin=NoMargin);` () and `arrow(..., margin=DotMargin);` ( ). Of these, the best alternative for this situation seems to be `margin=DotMargin`, which was designed for an arrow pointing to a dot created with the `dot` command. As desired, the resulting arrows remain the same length when the picture is scaled differently:



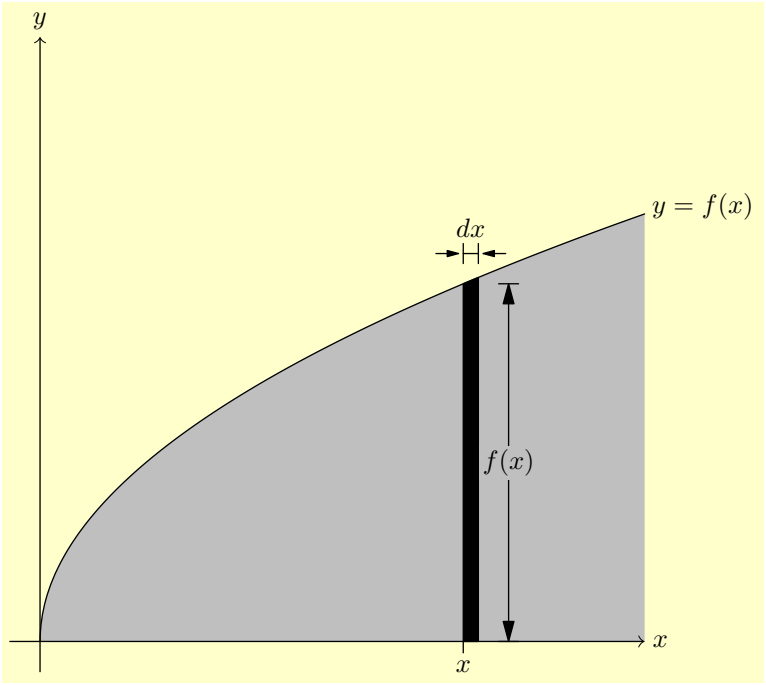

```
size(2cm);
    ⋮

real myarrowlength = 0.3cm;
margin arrowmargin =
    DotMargin;
arrow((x,barheight), W,
    length=myarrowlength,
    margin=arrowmargin);
arrow((x+dx,barheight), E,
    length=myarrowlength,
    margin=arrowmargin);

path clippath = circle(
    (x+dx/2,barheight), 0.5);
draw(clippath, blue);
clip(clippath);
```



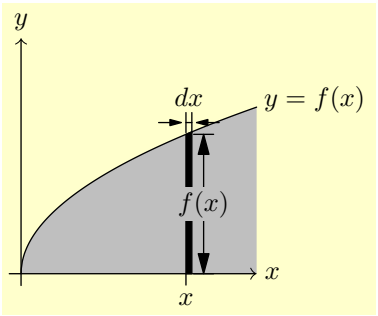

```
size(2cm);
    ⋮

real myarrowlength = 0.3cm;
margin arrowmargin =
    DotMargin;
arrow((x,barheight), W,
    length=myarrowlength,
    margin=arrowmargin);
arrow((x+dx,barheight), E,
    length=myarrowlength,
    margin=arrowmargin);

path clippath = circle(
    (x+dx/2,barheight), 0.2);
draw(clippath, blue);
clip(clippath);
```

2.25 Drawing objects shifted by an unscalable amount

Let's review the full, unclipped version of what Janet has accomplished so far:



Janet at first thinks that the two-dimensional drawing is now finished. However, she believes she will need to make the scale a bit smaller for the final drawing, so she decides to try a different size just to make sure the picture scales correctly:




```
settings.outformat = "pdf";
size(5cm, 0);

    ⋮
```

While the result is not awful, there are a couple of issues. Most obviously, the label $f(x)$ goes over the bar. Less problematic, but still irritating to Janet, is the way the arrow to the lower right of the dx label goes into the shaded region, rather than remaining strictly above it.

Vincent points out that in both cases, the problem was that specific distances were scaled that should not have been: the dx label should have remained the same height above the black strip no matter how the scaling changed, and the $f(x)$ label should have remained the same distance to the right. Unfortunately, after extensively consulting the documentation, neither Janet nor Vincent can find a

completely satisfactory way to include an unscalable length in the positioning of an object.

Finally, after weeks of considering the problem on-and-off and even consulting the source code for the basic Asymptote modules, Vincent manages to create a drawing command that does what is required. First, he adds the following definition[9] to the beginning of the file:

```
void drawshifted(path g, pair trueshift, picture pic =
    currentpicture, Label label="", pen pen=currentpen,
    arrowbar arrow=None, arrowbar bar=None, margin
    margin=NoMargin, marker marker=nomarker)
{
  pic.add(new void(frame f, transform t) {
      picture opic;
      draw(opic, L=label, shift(trueshift)*t*g, p=pen,
    arrow=arrow, bar=bar,
          margin=margin, marker=marker);
      add(f,opic.fit());
    });
  pic.addBox(min(g), max(g), trueshift+min(pen),
    trueshift+max(pen));
}
```

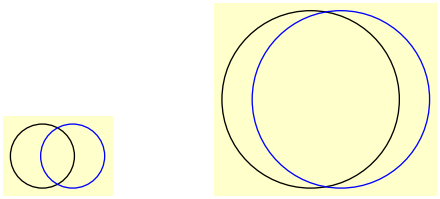
With this definition in place, the command, for instance, `drawshifted(g, (0.4cm, 0))` should draw the path `g` shifted exactly 3 millimeters to the right of its "natural" position; and this shift of 4 millimeters will remain the same, regardless of the scaling. Consider, for instance, the following code:

```
1  void drawshifted(path g, pair trueshift, picture pic =
       currentpicture, Label label="", pen pen=currentpen,
       arrowbar arrow=None, arrowbar bar=None, margin
       margin=NoMargin, marker marker=nomarker)
2  {
3    pic.add(new void(frame f, transform t) {
4        picture opic;
5        draw(opic, L=label, shift(trueshift)*t*g, p=pen,
       arrow=arrow, bar=bar,
6            margin=margin, marker=marker);
7        add(f,opic.fit());
8      });
9    pic.addBox(min(g), max(g), trueshift+min(pen),
       trueshift+max(pen));
10 }
```

[9] The original definition here was erroneous; the error was pointed out by user `mauvia` in the discussion thread at `https://sourceforge.net/p/asymptote/discussion/409349/thread/d0bc619b/?limit=25`, who also provided the core of the partial correction.

```
12 settings.outformat="pdf";
13 size(***);
14 draw(unitcircle);
15 drawshifted(unitcircle, trueshift=(0.4cm,0), pen=blue);
```

No matter how the scaling is set by changing the measurement in the `size` command (replacing the `***` on line 13), the blue circle will always be exactly four millimeters to the right of the black circle:



Warning: If the `drawshifted()` method above is used with fixed-size elements such as a label, arrow, and/or bar, then any subsequent occurrences of the functions `min(currentpicture)` and `max(currentpicture)` will ignore these fixed-size elements and may consequently be incorrect.

Recall that the vertical line with the $f(x)$ label was created with the following code:

```
real barx = x + dx + 0.1;
Label fxlabel = Label("$f(x)$", align=(0,0), position=MidPoint,
    filltype=Fill(fillpen));
draw((barx,0) -- (barx, f(x)), L=fxlabel, arrow=Arrows(),
    bar=Bars);
```

The vertical strip has coordinate x on the left and $x + dx$ on the right; in this code, the label is shifted right by a distance of 0.1 scalable units. To use instead an unscalable shift of 4.2 millimeters, Janet replaces the code above with the following:

```
/* Note: the following code will only work after the
    drawshifted() command has been defined as above. */
real barx = x + dx;
pair barshiftx = (0.42cm, 0);
Label fxlabel = Label("$f(x)$", align=(0,0), position=MidPoint,
    filltype=Fill(fillpen));
drawshifted((barx,0) -- (barx, f(x)), trueshift=barshiftx,
    label=fxlabel, arrow=Arrows(), bar=Bars);
```

The dx label  is a bit trickier, because it includes both a scalable part (the line with bars on both ends) and an unscalable part (the arrows pointing in). Janet's solution is to use the `drawshifted()` command for the scalable part, and the `draw()` variant described earlier (p. 42) to draw the arrows. See lines 40–52 of the code on page 49.

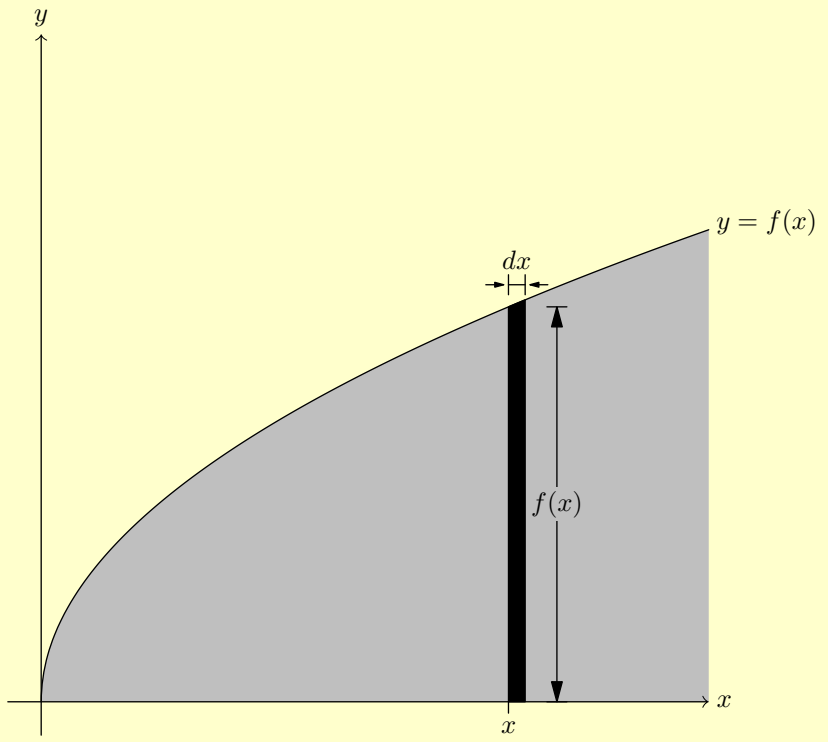
2.26 The two-dimensional picture: final result

The code:

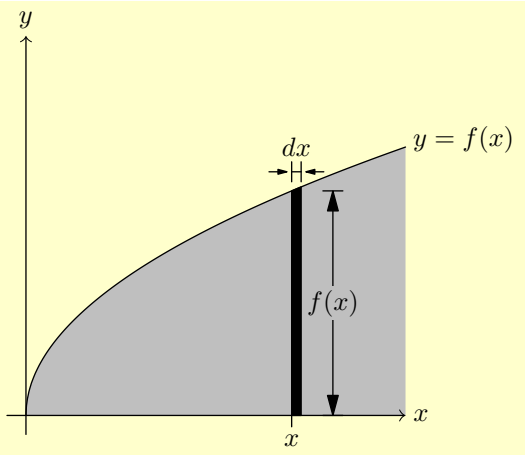
```
//Basic settings
settings.outformat="pdf";
size(11cm, 0);
defaultpen(fontsize(10pt));
import graph;

//Define the command drawshifted, to be used later
void drawshifted(path g, pair trueshift, picture pic =
    currentpicture, Label label="", pen pen=currentpen,
    arrowbar arrow=None, arrowbar bar=None, margin
    margin=NoMargin, marker marker=nomarker)
{
  pic.add(new void(frame f, transform t) {
      picture opic;
      draw(opic, L=label, shift(trueshift)*t*g, p=pen,
    arrow=arrow, bar=bar,
            margin=margin, marker=marker);
      add(f,opic.fit());
    });
  pic.addBox(min(g), max(g), trueshift+min(pen),
    trueshift+max(pen));
}

//Save some important numbers.
real xmin = -0.1;
real xmax = 2;
real ymin = -0.1;
real ymax = 2;

//Draw the graph and fill the area under it.
real f(real x) { return sqrt(x); }
path s = graph(f, 0, 2, operator..);
pen fillpen = mediumgray;
fill(s -- (xmax,0) -- cycle, fillpen);
draw(s, L=Label("$y=f(x)$", position=EndPoint));

//Fill the strip of width dx
real x = 1.4;
real dx = .05;
real t0 = times(s,x)[0];
real t1 = times(s,x+dx)[0];
path striptop = subpath(s,t0,t1);
filldraw((x,0) -- striptop -- (x+dx,0) --  cycle, black);
```

```
40 //Draw the bars labeling the width dx
41 real barheight = f(x+dx);
42 pair barshifty = (0, 0.2cm);
43 Label dxlabel = Label("$dx$", position=MidPoint, align=2N);
44 drawshifted((x,barheight) -- (x+dx, barheight),
       trueshift=barshifty, label=dxlabel, bar=Bars);

46 //Draw the arrows pointing inward toward the dx label
47 real myarrowlength = 0.3cm;
48 margin arrowmargin = DotMargin;
49 path leftarrow = shift(barshifty) * ((-myarrowlength, 0) --
       (0,0));
50 path rightarrow = shift(barshifty) * ((myarrowlength, 0) --
       (0,0));
51 draw((x, barheight), leftarrow, arrow=Arrow(),
       margin=arrowmargin);
52 draw((x+dx, barheight), rightarrow, arrow=Arrow(),
       margin=arrowmargin);

54 //Draw the bar labeling the height f(x)
55 real barx = x + dx;
56 pair barshiftx = (0.42cm, 0);
57 Label fxlabel = Label("$f(x)$", align=(0,0), position=MidPoint,
       filltype=Fill(fillpen));
58 drawshifted((barx,0) -- (barx, f(x)), trueshift=barshiftx,
       label=fxlabel, arrow=Arrows(), bar=Bars);

60 //Draw the axes on top of everything that has gone before
61 arrowbar axisarrow = Arrow(TeXHead);
62 Label xlabel = Label("$x$", position=EndPoint);
63 draw((xmin,0) -- (xmax,0), arrow=axisarrow, L=xlabel);
64 Label ylabel = Label("$y$", position=EndPoint);
65 draw((0,ymin) -- (0,ymax), arrow = axisarrow, L=ylabel);

67 //Draw the tick mark on the x-axis
68 path tick = (0,0) -- (0,-0.15cm);
69 Label ticklabel = Label("$x$", position=EndPoint);
70 draw((x,0), tick, L=ticklabel);
```

The result:

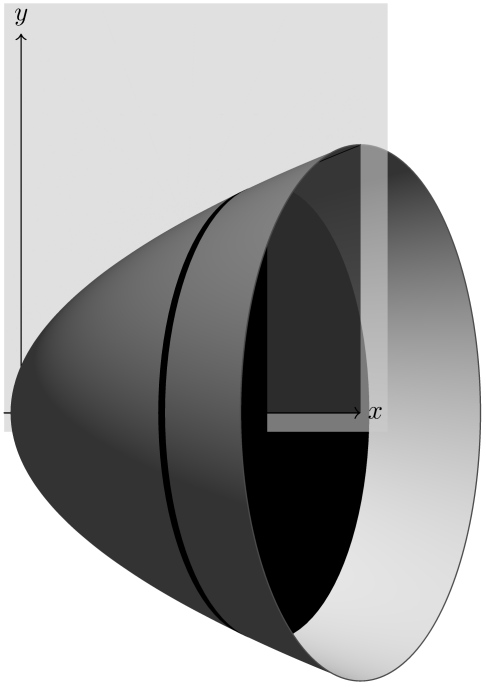


Here's a rescaled version, obtained by substituting `size(7cm,0)` for line 3, to show that rescaling actually works properly:



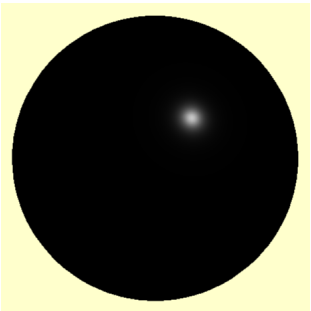
3 Three-dimensional images

With the two-dimensional image done, Janet turns her attention to the image of the surface of revolution. Here is the sort of thing she has in mind:



3.1 Hello Sphere

Here is perhaps the simplest reasonable three-dimensional image that can be drawn (as a pdf file) using Asymptote:

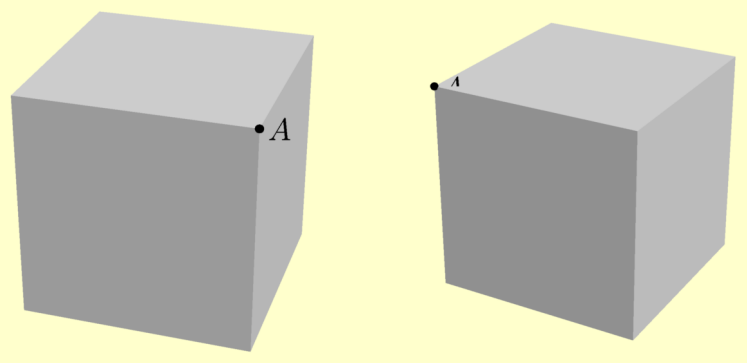


```
1 settings.outformat = "pdf";
2 settings.prc = false;
3 size(5cm,0);
4 import three;

6 draw(unitsphere);
```

Here is a line-by-line explanation for what is going on (especially in the preamble):

1 `settings.outformat = "pdf";` This line configures Asymptote to output a pdf file rather than the default eps file, as discussed on page 4.

2 `settings.prc = false;` By default, when Asymptote does 3d stuff and outputs a pdf file, the 3d information will be embedded in the pdf file as prc data. This means that if the pdf file is viewed using Adobe Reader, it will be possible to rotate the image interactively. Vincent thinks this is really cool. Janet thinks that for some complex mathematical objects, it may be impossible to see all the important features from any one point of view; in such cases, an interactive image is a great boon. Unfortunately, there is also an important drawback to interactive images: if an image is rotated, then labels can move away from the objects they are labeling or can be obscured when other objects move in front of them. For instance, the image below looks good when viewed from one point of view, but when it is rotated the letter A gets submerged in the cube:



Thus, it is important to decide early on whether an image will be interactive or not. Either choice comes with its own challenge:

- If you are creating an interactive image, then you have to make sure that the image (especially the labels) look good from every angle.
- If you are creating a non-interactive image, then you need to make sure that all the important aspects of the image are visible from a single point of view.

Janet decides to go with a non-interactive image because of the following technical considerations:

- Janet wants to be able to print out and make copies of a handout containing her image. (This is also why she has largely been working in grayscale.)
- The interactive functionality is only available when the pdf file is viewed with Adobe Reader, which is not the default pdf viewer on Janet's Mac. The rotating functionality is also lost whenever an Asymptote image is included in a TEX file via the naïve `\includegraphics` command.
- Although this has not been discussed in this tutorial, it is possible to create images in a LATEX document by including Asymptote code directly in the `tex` file; in fact, this method has been used to produce most of the

images in this tutorial. When this is done, especially for many images, the most efficient compilation method by far is to use `latexmk` as described in the Asymptote manual. And unfortunately, `latexmk` seems to choke when given an interactive 3d image from Asymptote, at least on Janet's computer.

Warning: when using `settings.prc=true;`—The rest of this tutorial will proceed under the assumption that the image to be produced is from a fixed (non-interactive) viewpoint. However, users wishing to experiment with interactive images should be aware of the following technical issue. Due to a bug in the version of Asymptote that ships with MacTeX 2013, if `settings.prc` is set to `true` (as it is by default), then sometimes the image shows up blank unless the viewer activates the interactive functionality. In particular, for either of the two cases described above, the image appears completely blank. Fortunately, this can be fixed by adding the line `settings.embed=true;` (which is supposed to be true by default).

3 `size(5cm,0);` This command behaves exactly as it does for two-dimensional drawing: the scaling factor will be set such that the final image is five centimeters wide (if possible). Without some sizing command like this, the image would be of a sphere with radius one point, which would be tiny. With the command `size(5cm,2cm);`, the image would be scaled such that either the width was 5 cm and the height ≤ 2 cm, or the width was ≤ 5 cm and the height was 2 cm. With the command `unitsize(1cm);`, the image would be a sphere with radius 1 cm.

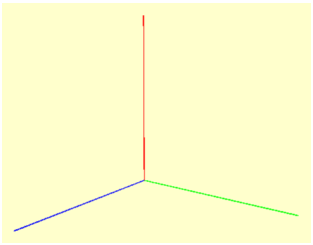
4 `import three;` Much of the code for dealing with three-dimensional objects is written in the Asymptote language. Like the code for graphing functions (see p. 28), this code is not loaded by default, but is contained in a module—in this case, the module `three`. This code is not loaded by default; the line `import three;` makes these features available.

Warning: Any changes to `settings.outformat`, `settings.prc`, or `settings.render` must take place *before* importing the module `three` or any module (such as `graph3`) that imports it.

6 `draw(unitsphere);` There is a pre-constructed object `unitsphere` of type `surface` that represents a sphere of radius one. This is the command to draw it.

3.2 Drawing lines in 3d

As a first step, Janet decides to try drawing the x- and y-axes. And, just to make the picture slightly more interesting, she adds a z-axis. These are drawn in different colors for now, so that we can tell which is which.



```
settings.outformat="pdf";
settings.prc = false;
import three;
size(4cm,0);

draw((0,0,0)--(2,0,0), blue);  //x-axis
draw((0,0,0)--(0,2,0), green); //y-axis
draw((0,0,0)--(0,0,2), red);   //z-axis
```

The code can be shortened slightly by using a few handy built-in aliases for the vectors $\vec{0}$, $\vec{\imath}$, $\vec{\jmath}$, and $\vec{k}$:

vector	alias
$(0,0,0)$	`O` (letter oh)
$(1,0,0)$	`X`
$(0,1,0)$	`Y`
$(0,0,1)$	`Z`

Thus, the triple `(2,-4,5)` could equivalently be expressed as `2X - 4Y + 5Z`. Note that `triple`s can be manipulated like vectors, via addition and scalar multiplication.

Using these aliases, the three `draw` commands above can be shortened to

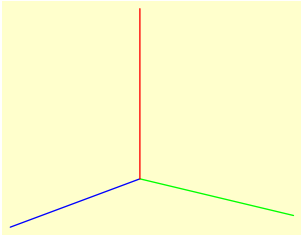
```
draw(O--2X, blue);  //x-axis
draw(O--2Y, green); //y-axis
draw(O--2Z, red);   //z-axis
```

3.3 Vector graphics in 3d

One thing that Janet immediately notices is that the image appears somewhat grainy.[10] Her first thought is surprise that Asymptote, the "vector graphics programming language," is producing rasterized graphics at all. As she explains to Vincent, a rasterized picture is saved by saving the color of each pixel. By contrast, a vector graphic actually saves a bunch of line segments and other information exactly. Vector graphics have a number of advantages over rasterized graphics; for one, they tend to look much better at high zoom levels, where a rasterized picture starts to look like a bunch of colored squares. In some renderers, they also look much better when zoomed out, since the renderer will know it still has to draw a line even if that line is less than a pixel thick.

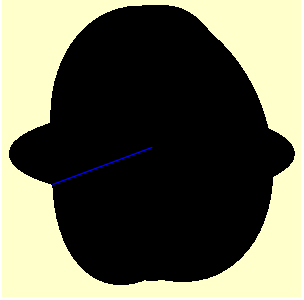
The setting in Asymptote that controls resolution is `settings.render`. It is possible to produce three-dimensional vector graphics using Asymptote by setting `render=0`:

[10] Depending on your pdf viewer, it may look just fine, or it may appear grainy or fuzzy or even be missing bits until you zoom in.



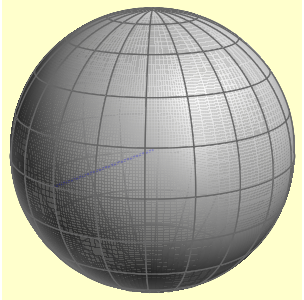
```
settings.outformat="pdf";
settings.prc = false;
settings.render = 0;
import three;
size(4cm,0);

draw(O--2X, blue); //x-axis
draw(O--2Y, green); //y-axis
draw(O--2Z, red);  //z-axis
```

Unfortunately, Asymptote's algorithm for turning three-dimensional graphics into two-dimensional vector drawings has some serious problems. Here is a naïve attempt to draw a sphere with a radius (which ought to be completely contained in the sphere, and thus invisible):



```
settings.outformat="pdf";
settings.prc=false;
settings.render=0;
import three;
size(4cm,0);

draw(unitsphere);
// The following line should be
    contained within the sphere.
draw(O--X, blue);
```

If you know how to work around these limitations, it is possible to produce some fairly nice vector graphics:



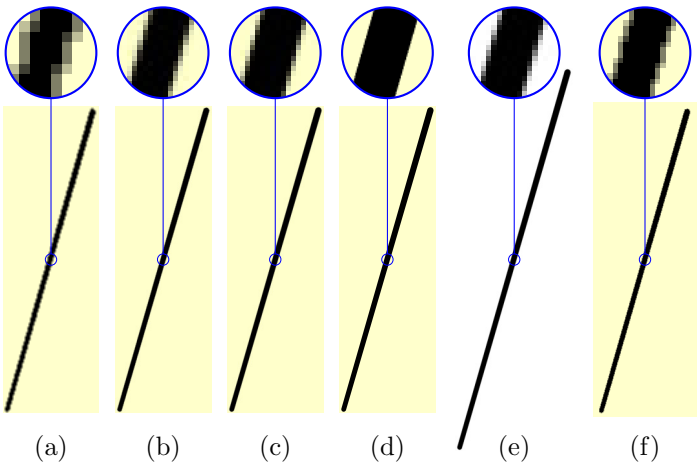
```
settings.outformat="pdf";
settings.prc=false;
settings.render=0;
import graph3;
size(4cm,0);
path3 myarc = rotate(18,Z) * Arc(c=0,
    normal=X, v1=-Z, v2=Z, n=10);
surface backHemisphere =
    surface(myarc, angle1=0,
    angle2=180, c=0, axis=Z, n=10);
surface frontHemisphere = surface(myarc, angle1=180,
    angle2=360, c=0, axis=Z, n=10);
draw(backHemisphere, surfacepen=material(white+opacity(0.8),
    emissivepen=0.1*white), meshpen=gray(0.4));
draw(O--X, blue+linewidth(1pt));
```

```
draw(frontHemisphere, surfacepen=material(white+opacity(0.8),
    emissivepen=0.1*white), meshpen=gray(0.4));
```

Such workarounds are time-consuming, limited, and I sincerely hope destined for obsolescence. Thus, I shall not take the time to explain the ones I know.

3.4 High-resolution rasterized images

The more versatile alternative is to use a rasterized image, but at a higher resolution. The setting that controls how high the resolution is computed is called `settings.render`. If you set `settings.render = 1;`, it computes one pixel per postscript point[11]; if you set `settings.render = 16;`, it computes 16 pixels per postscript point (which actually means that the rasterized image contains $16^2 = 256$ times as many pixels, and is likely to take a lot longer to compile and be significantly larger). Unfortunately, Janet finds that there is a limit to how far she can improve the resolution using this setting alone. For instance, the first three images below were produced with code that was identical except for making `settings.render` equal to 1, 4, and 16, respectively; arguably, the one with `settings.render=4;` looks better than the one with `settings.render=16;`.



(a) (b) (c) (d) (e) (f)

There are (at least) two ways to improve the actual resolution beyond the apparent "limit" illustrated in (c).

1. Changing the format from `pdf` to the format `png`, which is used for rasterized images, will cause high values of `settings.render` to produce high-resolution images, as seen in (d). This is the simpler method, and usually the recommended one. The main drawback to this method is that with a `png` file, any two-dimensional parts to the image will be rasterized

[11] one postscript point $= \frac{1}{72}$ inches ≈ 0.0353 centimeters

as well. Thus, this method is not ideal for producing images with mixed two- and three-dimensional output, such as the one on page 3.

2. Alternatively, adding the line

```
shipout(scale(4.0)*currentpicture.fit());
```

to the end of the code will cause the entire Asymptote picture (including any text) to be scaled up by a factor of `4.0`. (Other factors can, of course, be substituted.) This increases the threshold for higher values of `settings.render` to produce better results, as seen in (e). [Note that scaling up the image without increasing `settings.render` does not do any good, as seen in (f).] The main disadvantage of this method, aside from its complexity, is that the resulting image must be scaled back down. Thus, for instance, (e) and (f) were each included in a `.tex` file using the line `\includegraphics[scale=0.25]{filename.pdf}` (with `filename` replaced by the actual file name). It would also have worked in this case to replace `[scale=0.25]` by `[width=1.2cm]`; this is, in fact, the only viable option for producing high-resolution rasterized images with the Asymptote code written into a LaTeX file via the `asymptote` package.

A second disadvantage is that this method can make small changes to the dimensions of the resulting picture; in the images above, the two rightmost images are both taller and wider than the preceeding four images. This is unlikely to cause problems as long as you are consistent in which method you use.

Here is the code that was used to produce the images above. The lines of code under discussion are in red.

(a)

```
settings.outformat="pdf";
settings.render=1;
settings.prc = false;
import three;
size(1cm,0);
draw((0,0,0) -- (1,1,1),
    linewidth(2pt));
```

(b)

```
settings.outformat="pdf";
settings.render=4;
settings.prc = false;
import three;
size(1cm,0);
draw((0,0,0) -- (1,1,1),
    linewidth(2pt));
```

(c)

```
settings.outformat="pdf";
settings.render=16;
settings.prc = false;
import three;
size(1cm,0);
draw((0,0,0) -- (1,1,1),
    linewidth(2pt));
```

(d)

```
settings.outformat="png";
settings.render=16;

import three;
size(1cm,0);
draw((0,0,0) -- (1,1,1),
    linewidth(2pt));
```

(e)

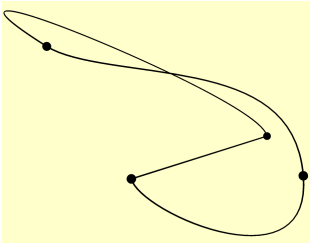
```
settings.outformat="pdf";
settings.render=16;
settings.prc = false;
import three;
size(1cm,0);
draw((0,0,0) -- (1,1,1),
    linewidth(2pt));
shipout(scale(4.0) *
    currentpicture.fit());
```

(f)

```
settings.outformat="pdf";

settings.prc = false;
import three;
size(1cm,0);
draw((0,0,0) -- (1,1,1),
    linewidth(2pt));
shipout(scale(4.0) *
    currentpicture.fit());
```

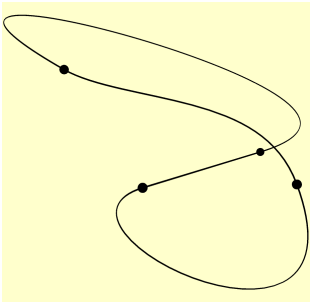
3.5 Three-dimensional paths

The built-in type for a three-dimensional path in Asymptote is `path3`. An intermediate type is `guide3`; functions will often return a `guide3` when you want a `path3`, but this is fine since Asymptote can automatically convert the former to the latter. Much as in the case of two-dimensional paths, the `--` operator connects two points by a line segment, while `..` uses a smooth curve. Either can be followed by the another, already constructed path or by the keyword `cycle`, which closes the curve. Another useful function is `dot(triple)`, which draws a small sphere at the indicated point; in the following code, this is used together with the pen lift operator `^^` to dot several points in one fell swoop.



```
settings.outformat="png";
settings.render=8;
import three;
size(4cm,0);
draw(-X -- X .. Y .. X-Y+Z .. cycle);
dot(-X ^^ X ^^ Y ^^ X-Y+Z);
```

Another interesting operator is `---`. Like the operator `--`, it draws a line segment between two points; unlike that operator, it tries to keep the path smooth beyond those two points if possible. Note that this operator works in the two-dimensional context also.



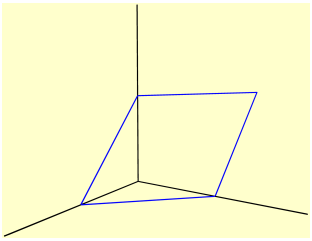
```
settings.outformat="png";
settings.render=8;
import three;
size(4cm,0);
draw(-X --- X .. Y .. X-Y+Z .. cycle);
dot(-X ^^ X ^^ Y ^^ X-Y+Z);
```

3.5.1 Parallelograms and 3d boxes

To draw the outline of a parallelogram with sides given by the vectors `u` and `v` starting at the point `O`, use the function

```
path3 plane(triple u, triple v, triple O=O);
```

Here's an example:



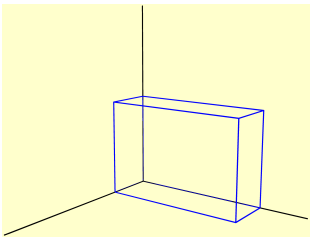
```
settings.outformat="png";
settings.render=16;
import three;
size(4cm,0);
draw(O--2X ^^ O--2Y ^^ O--2Z, black);
draw(plane(O=X, Y-X, Z-X), blue);
```

Somewhat confusingly, a parameter named `O` need not have the value `O=(0,0,0)`, although that is its default value.

To draw the outline of a rectangular solid with opposite vertices at `v1` and `v2`, use the function

```
path3[] box(triple v2, triple v2);
```

Here's an example:



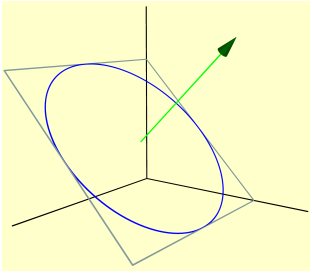
```
settings.outformat="png";
settings.render=16;
import three;
size(4cm,0);
draw(O--2X ^^ O--2Y ^^ O--2Z);
draw(box(O, (0.5, 1.5, 1)), blue);
```

3.5.2 Circles and arcs

The function

```
path3 circle(triple c, real r, triple normal=Z);
```

creates an approximate circle in three-dimensional space with center `c` and radius `r` that lies in the plane normal to `normal`:

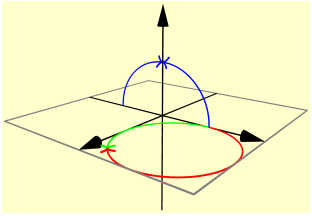


```
settings.outformat="png";
settings.render=16;
import three;
size(4cm,0);
draw(O--2X ^^ O--2Y ^^ O--2Z);
triple circleCenter = (Y+Z)/sqrt(2) +
    X;
path3 mycircle =
    circle(c=circleCenter, r=1,
    normal=Y+Z);
draw(plane(O=sqrt(2)*Z, 2X, 2*unit(Y-Z)), gray + 0.1cyan);
draw(mycircle, blue);
draw(shift(circleCenter) * (O -- Y+Z), green, arrow=Arrow3());
```

The most convenient function to create arcs in three dimensions is

```
path3 arc(triple c, triple v1, triple v2, triple normal=O);
```

which draws an arc centered at `c` from `v1` to the line through `c` and `v2`. By default, the arc drawn will be $\leq 180°$. If it is exactly 180°—i.e., if `v1`, `v2`, and `c` are all collinear—then the normal is necessary to determine the plane of the arc. Otherwise, the normal vector is optional, but can be used to force the complementary arc (greater than 180°).



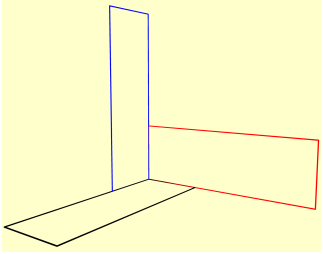
```
settings.outformat="png";
settings.render=16;
import three;
size(4cm);

draw(-2X--2X,
    arrow=Arrow3(emissive(black)));
draw(-2Y--2Y, arrow=Arrow3(emissive(black)));
draw(-2Z--2Z, arrow=Arrow3(emissive(black)));
draw(path3(box((-2,-2),(2,2))), gray);
draw(arc(c=O, Y, Z), blue, arrow = Arrow3(TeXHead2,
    emissive(blue)));
draw(arc(c=O, -Y, Z), blue, arrow = Arrow3(TeXHead2,
    emissive(blue)));
draw(arc(c=(1,1,0), Y, 2X, normal=Z), green, arrow =
    Arrow3(TeXHead2(normal=Z), emissive(green)));
draw(arc(c=(1,1,0), Y, 2X, normal=-Z), red, arrow =
    Arrow3(TeXHead2(normal=Z), emissive(red)));
```

3.5.3 Planar curves

To convert a two-dimensional `path` variable into a three-dimensional `path3`, use the function

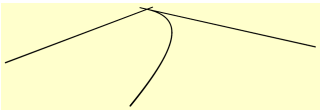
```
path3 path3(path p, triple plane(pair) = XYplane);
```

By default, this function treats maps the path into the xy-plane by applying $(x, y) \mapsto (x, y, 0)$. If the optional argument is changed to `ZXplane` or `YZplane`, then (x, y) maps to $(y, 0, x)$ or $(0, x, y)$, respectively.



```
settings.outformat = "png";
settings.render=16;
import three;
size(4.15cm, 0);

path p = box((0,-0), (3,1));
draw(path3(p), black);
draw(path3(p, plane=ZXplane), blue);
draw(path3(p, plane=YZplane), red);
```

Janet can use this to start lifting her two-dimensional diagram to three dimensions:

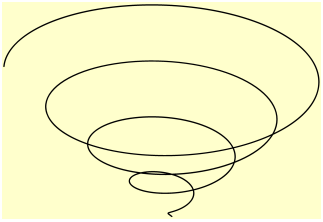


```
//Basic settings
settings.outformat = "png";
settings.render = 8;
import graph;

import three;
size(4.15 cm, 0);

//Save some important numbers.
real xmin = -0.1;
real xmax = 2;
real ymin = -0.1;
real ymax = 2;

//Construct the graph.
real f(real x) { return sqrt(x); }
path s = graph(f, 0, 2, operator..);

//Draw the graph and the axes.
draw(path3(s));
draw(xmin*X -- xmax*X);
draw(ymin*Y -- ymax*Y);
```

She is dissatisfied with the result, since the axes are lying down in a horizontal plane rather than standing upright. This will be corrected later, by playing with the point of view—specifically the `up` parameter, which controls which way is up.

3.5.4 Parametric curves

For drawing complicated three-dimensional curves in Asymptote, assembling them point-by-point is not generally feasible. Instead, it is better to describe the curve by a parametric function and then plot it using the `graph` function from the `graph3` module:



```
settings.outformat="png";
settings.render=16;
import graph3;
size(4.15cm, 0);
currentprojection =
    orthographic(0,2,1);

triple f(real t) {
    return (t*cos(t), t*sin(t), t);
}

path3 spiral = graph(f, 0, 8pi, operator ..);
draw(spiral);
```

3.6 Surfaces of revolution

The most interesting surface Janet needs to draw is not a sphere, but a surface obtained by revolving the graph of $y = \sqrt{x}$ about the x-axis. She is prepared to work out how to define it as a parametric surface, but is gratified to learn that there is a simpler way: Asymptote has a facility for producing surfaces of revolution directly.

Warning: The `surface()` function that I am about to introduce to create surfaces of revolution is undocumented. Thus, it could conceivably cease to work without notice in future versions of Asymptote. If this should happen, it should still be possible to create surfaces of revolution in a slightly longer way using the `solids` module, which is documented.

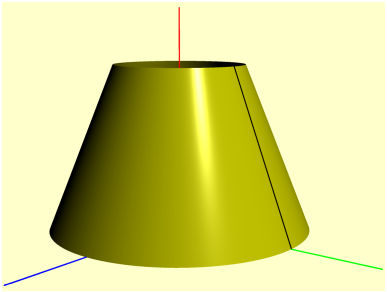
The documentation recommends creating surfaces of revolution using functions from the module `solids`, which allows additional options such as drawing silhouettes. However, if all you want to do is create a surface of revolution (with no silhouettes, skeletons, etc.), then the following function from the module `three` is typically simpler to use. The function is defined in the file `three_surface.asy`,

which is automatically included when the `three` module is loaded. Here is the specification[12] of the function, including only the required parameters:

```
surface(triple c, path3 g, triple axis);
```

The type `triple`, as discussed earlier, represents an ordered triple (a, b, c); depending on context, this can be a point or a vector in three-dimensional space $\mathbb{R}^3$. The type `path3` represents a path in three dimensions; objects of this type behave very similarly to paths in two dimensions (of type `path`). In particular, a line segment in three dimensions like `O -- 2X` is an object of type `path`.

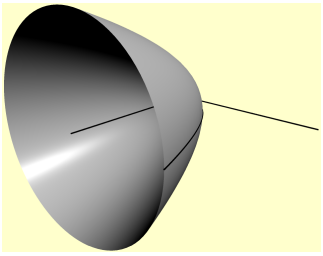
The effect of this function is to revolve the path `g` about the line `c -- c+axis`, i.e., the line through the point c in the direction $\vec{axis}$, and return the resulting surface of revolution. The surface may then be drawn with the `draw()` function.



```
settings.outformat="png";
settings.render=8;
import three;

size(5cm,0);
//create segment
path3 segment = (0,1.2,0) --
    (0,0.6,1.5);
//create surface of revolution
surface lampshade =
    surface(segment, c=O, axis=Z);
//draw surface
draw(lampshade, yellow);
//draw revolved segment for reference
draw(segment, black);
//draw axes for reference
draw(O--2X, blue); //x-axis
draw(O--2Y, green); //y-axis
draw(O--2Z, red); //z-axis
```

Here's a first stab at using this to construct the surface of revolution in Janet's diagram. (Recall that `s` was the name of the two-dimensional graph of $y = \sqrt{x}$.)

[12] This is a special kind of function called a *constructor*. Its return type is the same as its name, in this case `surface`.



```
    ⋮

path3 p3 = path3(s);
draw(p3);
surface solidsurface = surface(p3,
    c=0, axis=X);
draw(solidsurface, white);
draw(xmin*X -- xmax*X);
draw(ymin*Y -- ymax*Y);
```

3.6.1 optional parameters

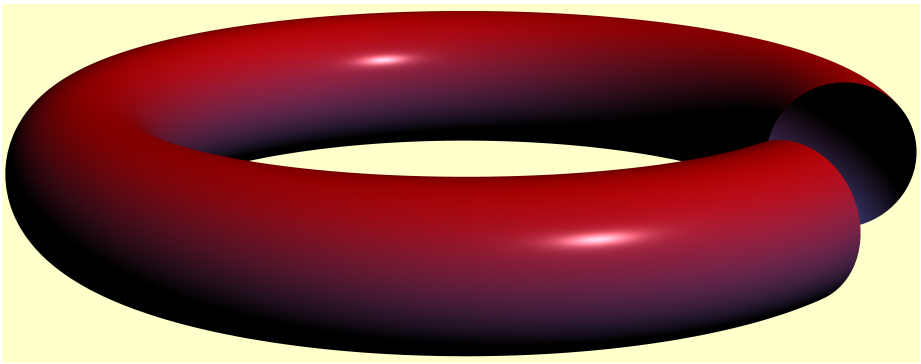
Since this function seems to be undocumented, here are the optional parameters, as defined by the source code, together with Vincent's best guesses about what they mean.

type	name	default value
`int`	`n`	`nslice` (usually 12)
`real`	`angle1`	`0`
`real`	`angle2`	`360`
`pen(int,real)`	`color`	`null`

The function creates a surface by revolving the given path from `angle1` to `angle2` (in degrees) about the specified axis. The parameter `n` can be used to refine the mesh, making the surface more accurate (for higher values of `n`) or faster to produce and draw (for lower values of `n`).

The parameter `color()` is a function that can be used to vary color within the surface, overriding any color parameter passed to the `draw()` command when applied to the surface. The first parameter of `color()` represents distance along the path (in the sense of "path times"), while the second parameter represents the number of slices of the revolution.

Here's an example that uses all of these optional parameters:



```
settings.outformat = "png";
settings.render=16;
size(345.0pt,0);
import graph3;
currentprojection = perspective(30*dir(75,0));
real r1=5, r0=1;
int nu = 36, nv = 36;
path3 crossSection = Circle(r=r0, c=(r1,0,0), normal=Y, n= nu);
pen colorFunction(int u, real theta) {
    real z = sin(u/nu * 2pi);
    real t = (z + 1) / 2;
    return t*red + (1-t)*lightblue;
}
surface torus = surface(crossSection, c=(0,0,0), axis=Z, n=nv,
    angle1=90, angle2=410, color=colorFunction);
draw(torus);
```

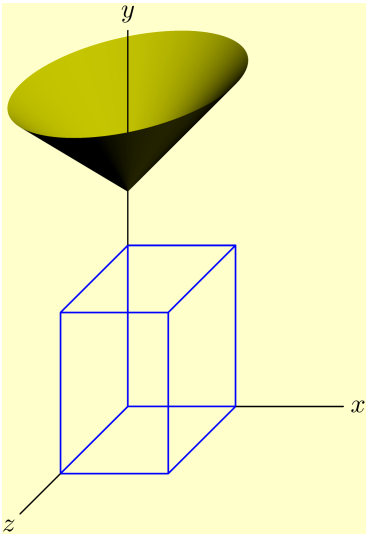
3.7 Points of view; projections

For non-interactive images, choosing a good point of view is something of an art. The way to select a point of view in Asymptote is to set the value of the predefined variable `currentprojection`, of type `projection`. Here are the most relevant options:

3.7.1 Oblique projection

Janet is used to a very simple setup for drawing three-dimensional objects on a two-dimensional chalkboard (or paper): the x and y axes are drawn as usual, and the z axis is drawn pointing down and to the left with a slope of 1. The z-axis is supposed to be imagined as sticking out of the page. This setup is especially nice for drawing surfaces of revolution on the board, since one can draw the curve exactly as usual in the xy-plane and then talk about revolving it about the x or y axis.

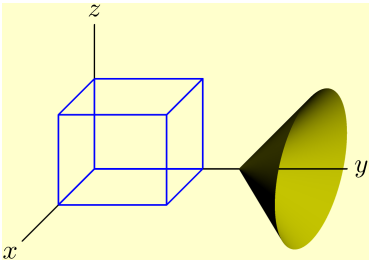
Asymptote does support this setup: it is called an oblique projection. Unfortunately, this surface of revolution about the y-axis looks much stranger here than it does on the chalkboard:




```
settings.outformat="png";
settings.render=8;
defaultpen(fontsize(10pt));
import three;
size(5cm,0);
currentprojection = oblique;
draw(O -- 2X, L=Label("$x$",
    position=EndPoint));
draw(O -- 3.5Y, L=Label("$y$",
    position=EndPoint));
draw(O -- 2Z, L=Label("$z$",
    position=EndPoint));
draw(box(O, (1,1.5,1.25)),
    blue+linewidth(0.6pt));
draw(surface(2Y -- 3Y+X, c=O,
    axis=Y), yellow);
```

Janet wonders if this is a bug in Asymptote. Vincent does some research and finds out that the problem is something inherent in the oblique projection: fundamentally, it is an unrealistic simplification of the projection of three-dimensional space onto a two-dimensional plane. For some objects it looks okay, but some things—especially surfaces of revolution (including some arrowheads in Asymptote)—just look weird.

In spite of the weirdness visible here, oblique projections are occasionally useful, so it is worthwhile to note that Asymptote also has projections called `obliqueX` and `obliqueY`, with the x and y axes (respectively) sticking "out of the page." The `obliqueZ` projection is the same as just plain `oblique`.




```
    ⋮
currentprojection = obliqueX;
    ⋮
```

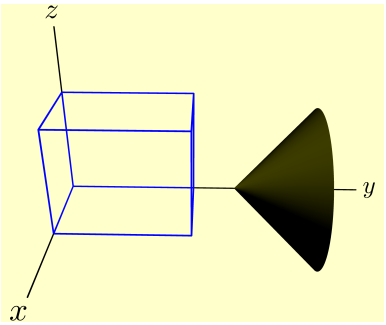
3.7.2 Perspective

On the other end of the spectrum lies vanishing point perspective, which is "realistic": closer objects appear larger. The line

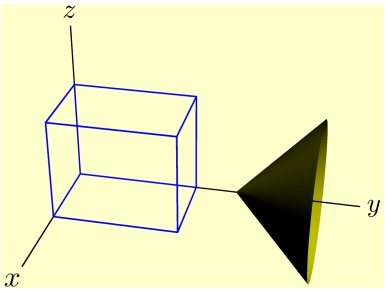
```
currentprojection = perspective(5,2,3);
```

tells Asymptote to draw the image as though it were produced by a camera located at the point $(5, 2, 3)$. The following three images show the same scene

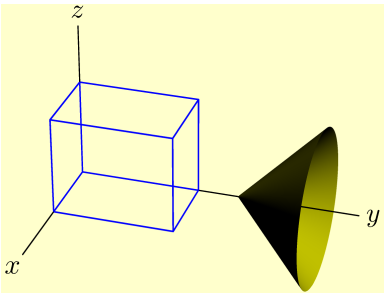
in perspective as the camera moves farther and farther away from the origin. Note that in the first image especially, text that is closer to the camera is visibly larger than text that is farther away. If this behavior is undesired, it can be prevented by including the two lines `Embedded.targetsize = true;` `Billboard.targetsize = true;` after the imports, although this will look funny if an interactive view is desired.




```
settings.outformat="png";
settings.render=16;
defaultpen(fontsize(10pt));
import three;
size(5cm,0);
currentprojection =
    perspective(5,2,3);
draw(O -- 2X, L=Label("$x$",
    position=EndPoint));
draw(O -- 3.5Y, L=Label("$y$",
    position=EndPoint));
draw(O -- 2Z, L=Label("$z$", position=EndPoint));
draw(box(O, (1,1.5,1.25)), blue+linewidth(0.6pt));
draw(surface(2Y -- 3Y+X, c=O, axis=Y), yellow);
```

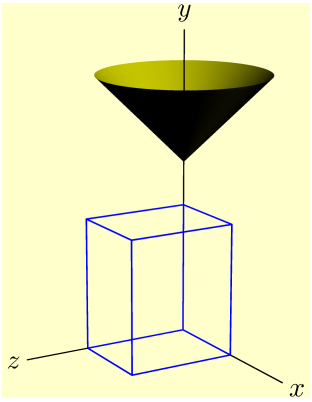



```
  ⋮
currentprojection =
    perspective(2*(5,2,3));
  ⋮
```



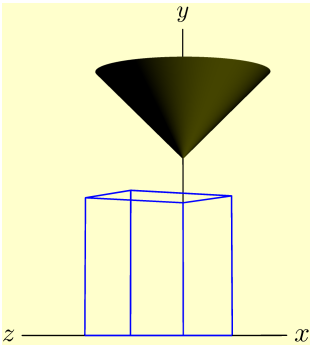

```
  ⋮
currentprojection =
    perspective(4*(5,2,3));
  ⋮
```

The optional argument `up=triple` tells Asymptote to rotate the camera (without changing where it is pointing) so that the specified vector will appear to point up:




```
  ⋮
size(4cm,0);
currentprojection =
    perspective(3*(5,2,3), up=Y);
  ⋮
```

One of the features Janet wants is that the x-axis should appear exactly horizontal and the y-axis exactly vertical. Theoretically, modifying the previous example to put the camera in the xz-plane while keeping `up=Y` ought to do this.




```
  ⋮
currentprojection =
    perspective(3*(5,0,3), up=Y);
  ⋮
```

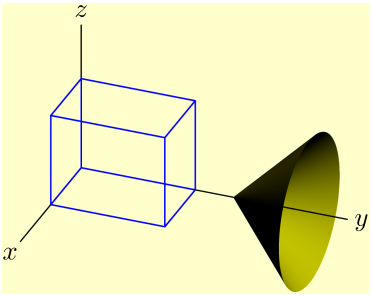
This basically works, but the effect is slightly spoiled when the axes appear to get bigger as they grow closer.

3.7.3 Orthographic projection

After the line

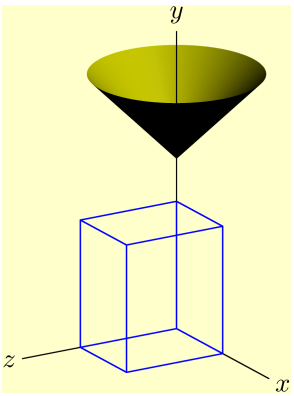
```
currentprojection=orthographic((5,2,3));
```

the image is drawn as if it were a zoomed-in picture taken by a camera very far away in the direction of the vector $(5, 2, 3)$:



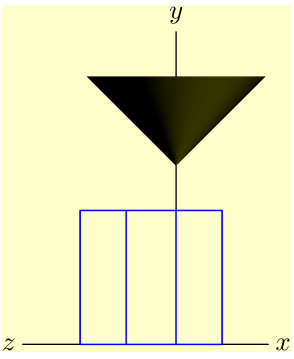

```
settings.outformat="png";
settings.render=4;
defaultpen(fontsize(10pt));
import three;
size(5cm,0);
currentprojection =
    orthographic((5,2,3));
draw(O -- 2X, L=Label("$x$",
    position=EndPoint));
draw(O -- 3.5Y, L=Label("$y$", position=EndPoint));
draw(O -- 2Z, L=Label("$z$", position=EndPoint));
draw(box(O, (1,1.5,1.25)), blue+linewidth(0.6pt));
draw(surface(2Y -- 3Y+X, c=O, axis=Y), yellow);
```

The optional argument `triple up` tells Asymptote to rotate the camera in place so that the specified vector will appear to point up:




```
  ⋮
size(4cm,0);
currentprojection = orthographic(5,2,3,
    up=Y);
  ⋮
```

Setting the y-coordinate equal to zero makes the x and z axes precisely horizontal:

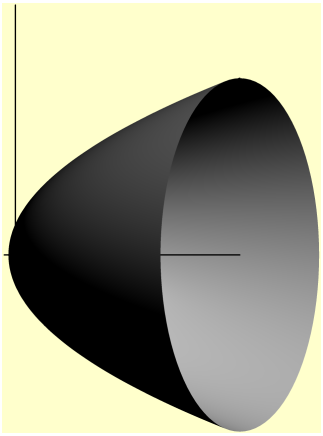



```
  ⋮
currentprojection = orthographic(5,0,3,
    up=Y);
  ⋮
```

3.7.4 General recommendation

After learning some of the techniques yet to be discussed, Janet recommends that some form of orthographic projection be used by default. Orthographic projections provide some degree of realism while allowing for certain tricks that involve moving an object directly toward or away from the camera without changing the object's size. However, she acknowledges that both perspective and oblique projections have their uses and should not be disdained.

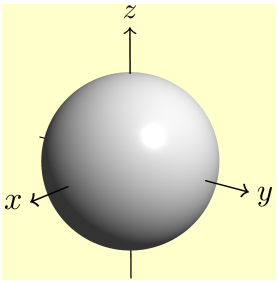
For her particular image, Janet chooses an orthographic projection. Crucially, she sets `up=Y` to make the image orient itself correctly.



```
  ⋮
currentprojection =
    orthographic(5,0,10, up=Y);
  ⋮
```

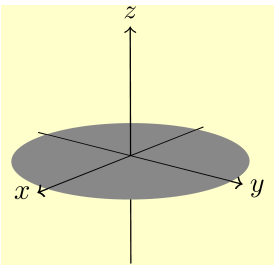
3.8 Predefined solids

There are a number of predefined surfaces that can be used to draw basic shapes and solids. We've already met one in 3.1—the unit sphere:

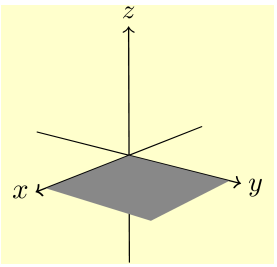



```
settings.outformat="png";
settings.render=16;
import three;
size(3.55cm, 0);
draw(-1.5X -- 1.5X,
    arrow=Arrow3(TeXHead2), L=Label("$x$",
    position=EndPoint, align=W));
draw(-1.5Y -- 1.5Y,
    arrow=Arrow3(TeXHead2), L=Label("$y$",
    position=EndPoint));
draw(-1.5Z -- 1.5Z, arrow=Arrow3(TeXHead2), L=Label("$z$",
    position=EndPoint));
draw(unitsphere, surfacepen = material(white, emissivepen =
    gray(0.2)));
```

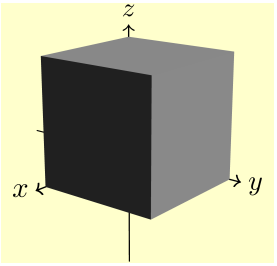
Here are some additional examples.



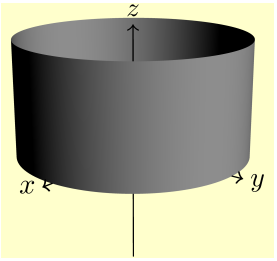

```
  ⋮
draw(-1.1X -- 1.1X,
    arrow=Arrow3(TeXHead2), L=Label("$x$",
    position=EndPoint, align=W));
draw(-1.1Y -- 1.1Y,
    arrow=Arrow3(TeXHead2), L=Label("$y$",
    position=EndPoint));
draw(-Z -- 1.1Z, arrow=Arrow3(TeXHead2), L=Label("$z$",
    position=EndPoint));
draw(unitdisk, surfacepen=white);
```



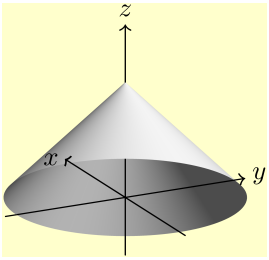

```
  ⋮
draw(unitplane, surfacepen=white);
```



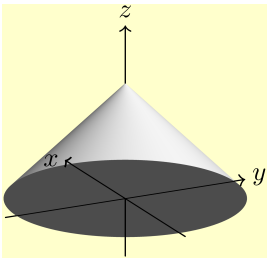

```
  ⋮
draw(unitcube, surfacepen=white);
```



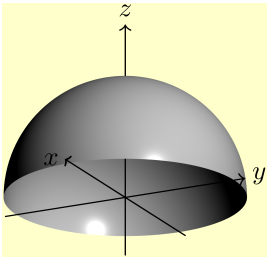

```
  ⋮
draw(unitcylinder, surfacepen=white);
```




```
  ⋮
draw(-.5Z -- 1.5Z, arrow=Arrow3(TeXHead2),
    L=Label("$z$", position=EndPoint));
currentprojection = orthographic(4,2,-1.5);
draw(unitcone, surfacepen =
    material(white, emissivepen =
    gray(0.3)));
```




```
  ⋮
draw(unitsolidcone, surfacepen =
    material(white, emissivepen =
    gray(0.3)));
```



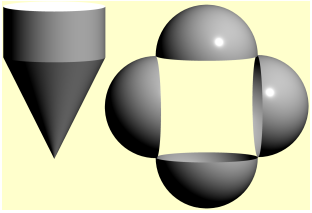

```
  ⋮
draw(unithemisphere, surfacepen = white);
```

3.9 Three-dimensional transforms

By themselves, the predefined surfaces are quite limited. However, they can be made much more flexible using three-dimensional transforms. For instance, to construct a sphere with radius `r` (of type `real`) centered at the ordered triple `c`, we can write

```
surface s = shift(c) * scale3(r) * unitsphere;
```

Three-dimensional transforms have type `transform3`. Like two-dimensional transforms (of type `transform`), they can be applied (on the left) and composed using the `*` operator. Here's an illustration of some useful transforms:



```
settings.outformat="png";
settings.render=16;
import three;
size(4cm);
currentprojection =
    orthographic(1,10,1);

for (int theta = 0; theta < 360; theta += 90) {
  /* Rotate by 'angle' degrees about the line u--v */
  draw( rotate(angle=theta, u=(0,0,-1), v=(0,1,-1)) *
    unithemisphere, surfacepen=white);
}

/* Rotate by 180 degrees about the y-axis , then shift three
    units along the x-axis and double the height*/
draw( scale(1,1,2) * shift(3X) * rotate(180, Y) * unitcone,
    surfacepen=white);

/* illustrating more shifts */
draw(shift(3,0,0) * unitcylinder, surfacepen = white);
draw(shift(3,0,1) * unitdisk, surfacepen = emissive(white));
```

One point that is perhaps not adequately brought out by this example are the subtleties of scaling. Here's a table that may help matters.

function	resulting transform
`scale3(real r)`	scaling factor `r`
`scale(real a, real b, real c)`	scale by `a` in the x-direction, `b` in the y-direction, and `c` in the z-direction
`scale(triple t)`	equivalent to `scale(t.x, t.y, t.z)`
`xscale3(real r)`	equivalent to `scale(r, 1, 1)`
`yscale3(real r)`	equivalent to `scale(1, r, 1)`
`zscale3(real r)`	equivalent to `scale(1, 1, r)`

Note that `scale(real r)` will always give a *two*-dimensional transform; if you try to apply it to a three-dimensional object, you will get an error.

There is one other function for producing a `transform3`: `reflect(triple u, triple v, triple w)` produces a reflection about the plane through the three points $\mathbf{u}, \mathbf{v}, \mathbf{w}$. Three reflections are of particular note:

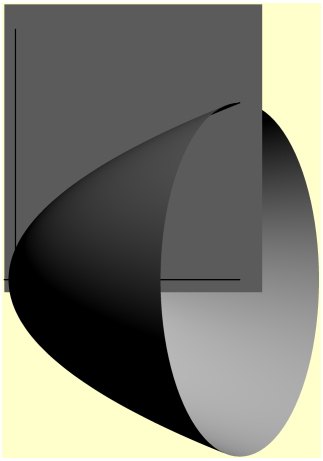
transform	effect
`reflect(O, Z, X+Y)`	switch x and y
`reflect(O, Y, X+Z)`	switch x and z
`reflect(O, X, Y+Z)`	switch y and z

Any permutation of x, y, z can be expressed as a product of these three. Here are two important use cases:

- To switch the xy and xz planes, apply `reflect(O, X, Y+Z)`.
- To switch the xy and yz planes, apply `reflect(O, Z, X+Y) * reflect(O, X, Y+Z)`.

3.10 Simple planar surfaces

Janet wants to add a symbol to her three-dimensional diagram that looks something like a piece of paper representing the plane of the original, two-dimensional drawing. She could do this by applying appropriate three-dimensional transformations to a `unitplane`. Vincent tells her there is a simpler way to construct planar surfaces—use a version of the `surface()` method that takes a single, two-dimensional `path` as a parameter. The result is a surface in the xy-plane whose boundary is the provided `path`.



```
//Basic settings
settings.outformat = "png";
settings.render = 8;
defaultpen(fontsize(10pt));
import graph;
import three;
size(4.15cm, 0);

currentprojection =
    orthographic(5,0,10, up=Y);

//Save some important numbers.
real xmin = -0.1;
real xmax = 2;
real ymin = -0.1;
real ymax = 2;
real margin = 0.2;

//Construct the graph.
real f(real x) { return sqrt(x); }
path s = graph(f, 0, 2, operator..);
```

```
//Draw the graph and the axes.
path3 p3 = path3(s);
draw(p3);
surface solidsurface = surface(p3, c=0, axis=X);
draw(solidsurface, white);
draw(xmin*X -- xmax*X);
draw(ymin*Y -- ymax*Y);

//Draw the plane.
path planeoutline = box((xmin, ymin), (xmax+margin,
    ymax+margin));
draw(surface(planeoutline), surfacepen=white);
```

Note: There is also a `surface` constructor that accepts a three-dimensional cyclic `path3` and attempts to construct a surface with that path as the outline. In the author's experience, this does not turn out nearly as well as the two-dimensional version.

3.11 Lighting

Janet is perplexed and frustrated that even though she said the plane should be drawn "white," it came out looking dark gray. Vincent tells her that this is because of the lighting: the plane is showing up as dark because the light is hitting it at a bad angle. This can be changed using the `light` parameter of the `draw()` command. The built-in options are `Viewport`, `White`, `Headlamp` (the default), and `nolight`. The effect of each on a sphere is shown below:

```
  ⋮
draw(unitsphere, white, light=Viewport);
```

```
  ⋮
draw(unitsphere, white, light=White);
```

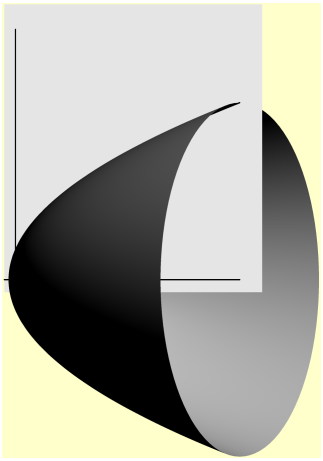
```
  ⋮
draw(unitsphere, white, light=Headlamp);
```

```
  ⋮
draw(unitsphere, white, light=nolight);
```

In this case, Janet thinks the first three lights behave similarly. Vincent agrees: in his experience, playing around with the lighting is rarely helpful. But there are exceptions; See, for instance, the picture of a spiral cone on p. 93, and see what happens when you compile it without the line `currentlight = White;`.

⚠ **Warning:** The `White` light, like snow, is actually very slightly blue. A typical human will not consciously perceive the difference, but a book printing company will want you to pay for color printing on images that use this lighting (or, more likely, convert them to grayscale).

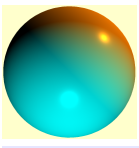
The crucial example for our current purposes is the fourth, which has `light = nolight`. The effect of `nolight` is to make Asymptote take the color parameter (`surfacepen`) literally and throw out all lighting considerations. For a sphere, this looks bad indeed. But for a planar surface, it is often exactly what is desired.



```
path planeoutline = box((xmin, ymin),
    (xmax+margin, ymax+margin));
draw(surface(planeoutline),
    surfacepen=lightgray,
    light=nolight);
```

Note that in the end, Janet decided that a `lightgray` rectangle gave a more realistic impression of a piece of paper than a pure white one.

Two more notes. First, the default value of `light` is actually `currentlight`, which can be changed. Second, striking effects can be produced by creating your own lighting:



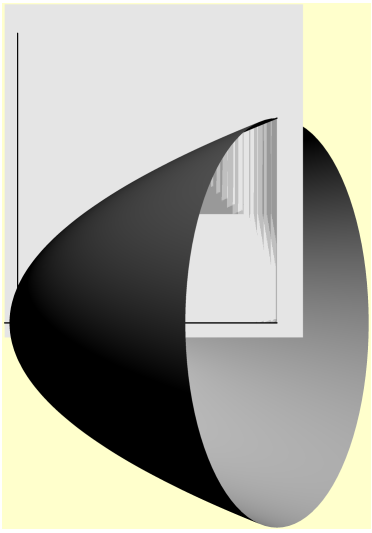
```
settings.outformat = "png";
settings.render = 8;
import three;
size(1.7cm, 0);
currentlight = light(diffuse = new pen[] {cyan, orange},
                     specular = new pen[] {black, white},
```

```
                        position = new triple[] {-Y+Z, X+Y});
draw(unitsphere, surfacepen=white);
```

However, such effects are complex to implement and more often distracting than helpful. In the opinion of both Janet and Vincent, a better alternative is usually provided by playing with the `surfacepen` parameter; see 3.18.1 on p. 89.

3.12 Planar surfaces with holes

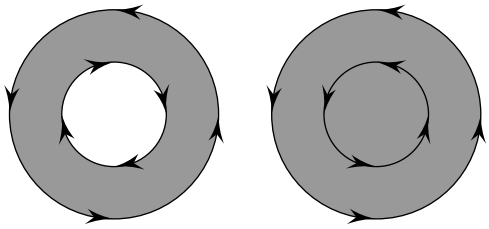
Next, Janet would like to add a gray area under the curve, imitating the two-dimensional picture. Unfortunately, when she tries adding it as a second planar surface, the result is . . . strange:



```
  ⋮

size(4.8cm, 0);

  ⋮

//Construct the graph and the area
    under it.
real f(real x) { return sqrt(x); }
path s = graph(f, 0, 2, operator..);
path fillregion = s -- (xmax,0) --
    cycle;

  ⋮

//Draw the plane.
path planeoutline = box((xmin,
    ymin), (xmax+margin,
    ymax+margin));
draw(surface(planeoutline), surfacepen=lightgray,
    light=nolight);
//Fill the area under the graph.
draw(surface(fillregion), surfacepen=gray(0.6), light=nolight);
```

Even more peculiarly, the picture changes in unexpected ways when small changes are made to the `size` parameter.

The trouble, as Vincent explains, is that Janet has set up two surfaces occupying exactly the same space. In two dimensions (or with `settings.render=0`), the surface drawn second would be the one to show up on top. But in three dimensions, which surface is displayed is determined entirely by which one is in front of the other (closer to the camera). With two surfaces in exactly the same place, Asymptote has no good way to decide which surface is in front of



(a) Good (b) Bad: both counterclockwise

the other, so the results can be unpredictable. Here's another illustration of the effect:

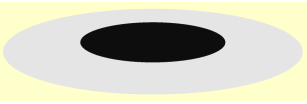
```
settings.outformat = "png";
settings.render = 8;
import three;

size(3cm, 0);
draw(surface(scale(2)*unitcircle), lightgray, nolight);
draw(surface(unitcircle), darkgray, nolight);
```

When Asymptote cannot decide which surface it is supposed to be drawing, it comes up with an unpredictable mishmash.

The preferred solution is to tell Asymptote to leave a hole in one surface, which will be filled by the other surface. To leave a hole in a surface, pass the `surface()` function a disconnected path (a.k.a. `path[]`) consisting of the outline of the surface, together with the outline of the hole *in the opposite direction* (see the figure). Here's an example using this technique to draw an annulus. Note that reversing the outer circle makes it go clockwise, so that the inner circle is going in the opposite direction:


```
settings.outformat = "png";
settings.render = 8;
import three;

size(4cm, 0);
surface s = surface(reverse(scale(2)*unitcircle) ^^ unitcircle);
draw(s, lightgray, light=nolight);
```

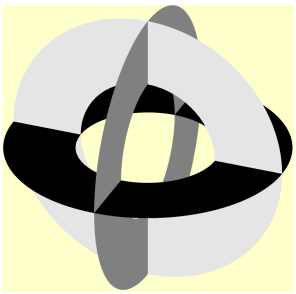
And here's how to use it to draw a light gray disk with a dark gray center:



```
settings.outformat = "png";
settings.render = 8;
import three;
```

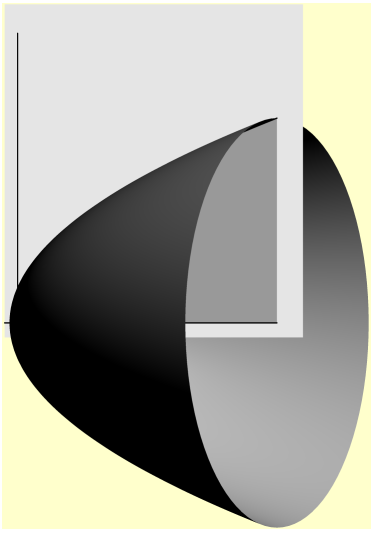
```
size(4cm, 0);
draw(surface(scale(2)*unitcircle ^^ reverse(unitcircle)),
    lightgray, nolight);
draw(surface(unitcircle), darkgray, nolight);
```

To place a planar surface in a plane other than the xy-plane, move it around using three-dimensional transforms (see 3.9, p. 72). In particular, there are transformations described in that section explicitly to move a surface from the xy plane into the xz or yz plane:



```
settings.outformat = "png";
settings.render = 8;
import three;
size(3.8cm, 0);
currentprojection = orthographic(5,2,3);

// counterclockwise ^^ clockwise
surface annulus = surface(unitcircle ^^
    reverse(scale(2)*unitcircle));
// xy plane:
draw(annulus, black, nolight);
// Switch xy and xz planes:
draw(reflect(O, X, Y+Z) * annulus, gray, nolight);
// Switch xy and yz planes:
draw(reflect(O, Z, X+Y) * reflect(O, X, Y+Z) * annulus,
    lightgray, nolight);
```

Transformations aside, when Janet adds a whole in the plane, the area under the curve looks the way it is supposed to:



```
  ⋮

//Construct the graph and the area
    under it.
real f(real x) { return sqrt(x); }
path s = graph(f, 0, 2, operator..);
//clockwise
path fillregion = s -- (xmax,0) --
    cycle;

  ⋮

//Draw the plane.
path planeoutline = box((xmin,
    ymin), (xmax+margin,
    ymax+margin));
```

```
//counterclockwise ^^ clockwise
draw(surface(planeoutline ^^ fillregion), surfacepen=lightgray,
    light=nolight);
//Fill the area under the graph.
draw(surface(fillregion), surfacepen=gray(0.6), light=nolight);
```

3.13 Arrowheads in three dimensions

At this point, Janet would like to go ahead and add the arrowheads to the axes. As in the two-dimensional case, the basic method here is to pass an arrowhead to the `arrow=` parameter of the `draw()` command. Unfortunately, arrowheads in three dimensions are somewhat more complex than in two dimensions because of lighting issues. There are basically two approaches: trying to make arrowheads look "fancy" and three-dimensional, or trying to make arrows look "plain" and two-dimensional.

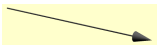
3.13.1 Fancy 3d arrowheads

In principle, a default three-dimensional arrow is the simplest instruction, requiring only the parameter `arrow=Arrow3()`.

```
settings.outformat = "png";
settings.render = 8;
import three;
size(2cm, 0);
draw(O -- Y, arrow=Arrow3(), p=gray(0.6));
```

This often works, but it has the consequence that the arrowhead often appears significantly darker than the line or path it is on. If this effect is undesirable, there are at least two ways to compensate:

1. Pass the parameter `light=currentlight` to the `draw()` command. This way, the line will be shaded (as a cylinder) to match the arrowhead.



```
draw(O -- Y, arrow=Arrow3(), p=gray(0.6),
    light=currentlight);
```

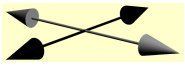
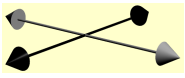
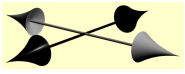
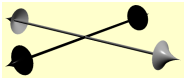
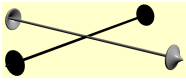
2. Pass a lighter color to the `Arrow3()` function (or whatever variant you are using). This way, the arrowhead will be lightened, potentially matching the line more closely.

```
draw(O -- Y, p=gray(0.6), arrow =
    Arrow3(arrowheadpen=material(gray(0.4),
    emissivepen=gray(0.3))));
```

   The `arrowheadpen=` parameter is actually a `material`; using the information of 3.18.1 (p. 89), you can exert much finer control over the appearance of the arrowhead. In particular, you can lighten the darkest part of the shadow.

The first option is the simplest, but it has the disadvantage of altering the color of the line in sometimes confusing ways. The second option may be necessary if you want to make sure several different lines pointed in different directions have the same color.

Here are the basic available styles for arrowheads with three-dimensional appearance:

`arrow=`	appearance
`Arrow3()`	
`ArcArrow3()`	
`Arrow3(HookHead3)`	
`ArcArrow3(HookHead3)`	
`Arrow3(TeXHead3)`	

Here is the code used to produce the image for `Arrow3()`:

```
settings.outformat = "png";
settings.render=8;
import three;
unitsize(1.5cm);
draw(-X -- X, arrow=Arrows3(), p=linewidth(1pt));
draw(-Y -- Y, arrow=Arrows3(), p=lightgray+linewidth(1pt),
    light=currentlight);
```

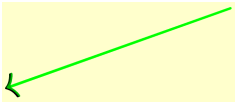
There are several things to note here:

- The line width was doubled (the default is 0.5 points) to make the arrowheads larger and easier to see.
- The function `Arrows3()` is a variation on `Arrow3()` that puts arrowheads at both ends of the path. `ArcArrows3()`, `Arrows()`, and `ArcArrows()` are similar.
- The `light=currentlight` option was used for simplicity. However, it was not needed for the black line, since the principle desired effect was to darken the line to match the arrowhead. (Black lines are already as dark as possible.)

3.13.2 Plain arrowheads in 3d

Janet thinks that the three-dimensional shaded arrowheads are distracting. What she wants is something that imitates the appearance of two-dimensional arrowheads. Vincent tells her that Asymptote does have the capability to do this. (For the record, Vincent thinks the shaded arrowheads are awesome.)

The main difficulty with the "plain" arrowheads is that, by default, Asymptote will still try to shade them. This can look rather bizarre:



```
settings.outformat = "png";
settings.render = 8;
import three;
size(3cm, 0);
draw(O -- X, arrow=Arrow3(TeXHead2), p=green+linewidth(1pt));
```

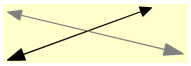
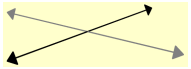
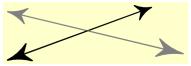
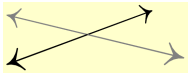
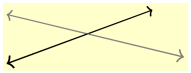
In this example, the "shading" makes the arrowhead much darker than it should be.

Janet thinks that this should be solved by setting the arrowhead to have lighting `nolight` so that it will follow the suggested pen color without any shading. Vincent agrees in principle; but there is no easy way to control the lighting of an arrowhead. Fortunately, there is an alternative: setting the `arrowheadpen=` parameter to (say) `emissive(green)` will have the same effect as setting it to `green` and specifying `nolight` (if that were possible):

```
settings.outformat = "png";
settings.render = 8;
import three;

size(3cm, 0);
draw(O -- X, arrow=Arrow3(TeXHead2, emissive(green)),
    p=green+linewidth(1pt));
```

For surfaces in general, an `emissive` material is an alternative to setting `light=nolight` with essentially the same effect; for more details, see 3.18.1 (p. 89).

Here are the basic options for a "plain" arrowhead in three dimensions:

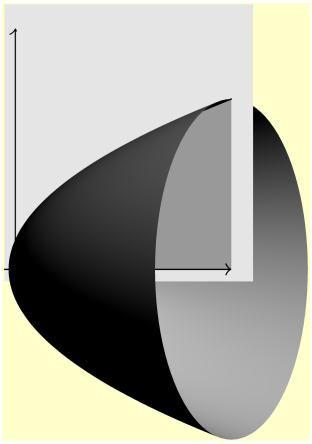
`arrow=`	appearance
`Arrow3(DefaultHead2, emissive(`⟨*color*⟩`))`	
`ArcArrow3(DefaultHead2, emissive(`⟨*color*⟩`))`	
`Arrow3(HookHead2, emissive(`⟨*color*⟩`))`	
`ArcArrow3(HookHead2, emissive(`⟨*color*⟩`))`	
`Arrow3(TeXHead2, emissive(`⟨*color*⟩`))`	

Here is the code used to produce the image for `Arrow3(DefaultHead2, emissive(`⟨*color*⟩`))`:

```
settings.outformat = "png";
settings.render=8;
import three;
unitsize(1.5cm);
draw(-X -- X, arrow=Arrows3(DefaultHead2, emissive(black)));
draw(-Y -- Y, arrow=Arrows3(DefaultHead2, emissive(gray)),
    p=gray);
```

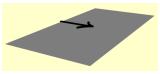
Again, `Arrows3()` is a variation on `Arrow3()` that puts arrowheads at both ends of the path. `ArcArrows3()`, `Arrows()`, and `ArcArrows()` are similar.

For her diagram, Janet opts for the `TeXHead2` option. Unfortunately, it does not work quite as expected—half the top arrowhead is missing:



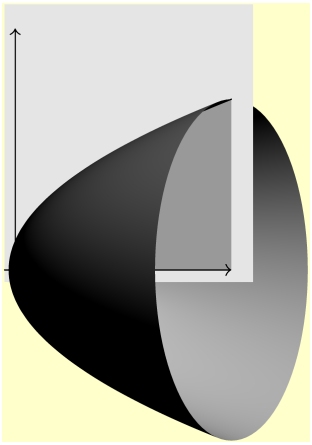
```
  ⋮

draw(xmin*X -- xmax*X,
    arrow=Arrow3(TeXHead2,
    emissive(black)));
draw(ymin*Y -- ymax*Y,
    arrow=Arrow3(TeXHead2,
    emissive(black)));

  ⋮
```

To embed arrowheads in a plane, Janet can use `DefaultHead2()`, `TeXHead2()`, etc. as *functions* with a `normal=` parameter:



```
settings.outformat = "png";
settings.render=8;

import three;
size(2cm, 0);
draw(surface(box((-2,0),(2,2))), lightgray);
draw(O -- Y, p=linewidth(1pt), arrow=Arrow3(TeXHead2(normal=Z),
    emissive(black)));
```

Without the `normal=` parameter, the arrowhead will be oriented to best approximate a two-dimensional arrowhead as seen from the camera. (If the camera is moved—say, in interactive viewing mode—it won't look good.) Setting the `normal=` to a vector normal to the plane rotates the arrowhead to lie within the plane. Even if the perspective change is not as obvious as above, this prevents the plane from blocking out part of the arrowhead:



```
  ⋮

draw(xmin*X -- xmax*X,
    arrow=Arrow3(TeXHead2(normal=Z),
    emissive(black)));
draw(ymin*Y -- ymax*Y,
    arrow=Arrow3(TeXHead2(normal=Z),
    emissive(black)));

  ⋮
```

3.13.3 3d arrows in interactive mode

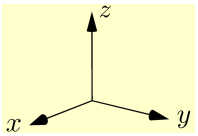
The plain arrows (except when embedded in a plane) are designed to be viewed from a single angle, so they tend not to look good in interactive mode. In general, the "fancy" three-dimensional arrowheads hold up better when the person viewing the picture starts changing the point of view.

An additional note: Janet thinks that the shading, gleaming "fancy" arrowheads, while impressive, are also distracting. If she wishes to use these arrowheads without the fancy shading, she can give them an `emissive(`⟨*color*⟩`)` parameter, just as with the "plain" arrowheads. There's a corresponding warning, however: without the fancy shading, a three-dimensional arrowhead pointed straight at the camera will look like a circle.

3.14 Labels in three dimensions

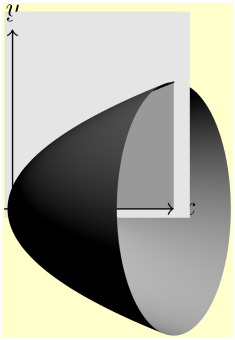
The basic methods of adding labels in three dimensions are the same as those in two dimensions (2.22, p. 35). However, there are—inevitably—some complications.

As in the two-dimensional case, there are three basic commands to choose from. The simplest is often to label a path as you draw it using the `L=` parameter of the `draw()` command:




```
settings.outformat = "png";
settings.render = 8;
defaultpen(fontsize(10pt));
import three;
size(2.5cm, 0);
draw(O -- X, arrow=Arrow3, L=Label("$x$", position=EndPoint,
    align=W));
draw(O -- Y, arrow=Arrow3, L=Label("$y$", position=EndPoint));
draw(O -- Z, arrow=Arrow3, L=Label("$z$", position=EndPoint));
```

Not the `Label()` function (p. 36). There is also a `label()` function that can be used to add a label to a `path3`, in case the path has already been drawn. But there is a problem when Janet tries to use these to label her axes:




```
draw(xmin*X -- xmax*X,
     arrow=Arrow3(TeXHead2(normal=Z),
                  emissive(black)),
     L=Label("$x$", align=E,
             position=EndPoint));
draw(ymin*Y -- ymax*Y,
     arrow=Arrow3(TeXHead2(normal=Z),
                  emissive(black)),
     L=Label("$y$", align=N,
             position=EndPoint));
```

The axis labels are partially hidden behind the plane! Once again, Janet has run afoul of the fact that in three dimensions, drawing precedence is determined entirely by distance from the camera. The label, unlike the plane, is turned to face the camera, which involves moving at least part of it farther away from the camera. The part that gets rotated away from the camera gets hidden by the plane.

The solution here is, unfortunately, a bit of a hack: layering can be simulated by moving the labels directly toward (or away from) the camera. The preferred way to do this is using yet another variation on the `label()` command:

```
void label(Label L, triple position);
```

with the following optional parameters:

type	name	default value
`picture`	`pic`	`currentpicture`
`align`	`align`	`NoAlign`
`pen`	`p`	`currentpen`
`light`	`light`	`nolight`
`string`	`name`	`""`
`render`	`render`	`defaultrender`
`interaction`	`interaction`	`LabelInteraction()`

In particular, the `position=` parameter will allow Janet to place a label at a point far away from any path.

Warning: Another way to move labels around is to applying a `transform3` to a `Label` object. However, this has the side effect of aligning the label toward the xy-plane rather than toward the camera position. This side effect is a huge bother unless it is deliberately sought, in which case it can be quite handy (see 3.17, p. 89).

3.15 Layering: moving objects closer to the camera

When the current projection is orthographic, moving it a given distance of units toward the camera is quite simple: just add `unit(currentprojection.camera)` (or apply `shift(unit(currentprojection.camera))`).

[Note that the `unit()` function takes a `pair` or a `triple` and returns the length one vector in the same direction; e.g., `unit((1,1))` is a decimal approximation of $(\frac{1}{2}\sqrt{2}, \frac{1}{2}\sqrt{2})$.]

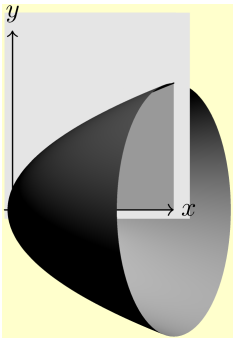
For determining the direction toward the camera when allowing for a perspective projection, Vincent comes up with the following:

```
triple cameradirection(triple pt, projection
    P=currentprojection) {
  if (P.infinity) {
    return unit(P.camera);
  } else {
```

```
    return unit(P.camera - pt);
  }
}

triple towardcamera(triple pt, real distance=1, projection
    P=currentprojection) {
  return pt + distance * cameradirection(pt, P);
}
```

The first function returns the direction from a point to the camera. The second function returns the result of moving a point a specified distance closer to the camera. Note that both of these work better for orthographic than perspective projections. Apart from all other considerations, in a perspective projection, Asymptote will actually move the camera if it gets "too close" to the objects being viewed, which can cause functions like this to behave in undesirable ways.

In any case, here is what happens when Janet incorporates these new functions and uses them, together with the `label()` command described previously:




```
  ⋮

//Direction of a point toward the camera.
triple cameradirection(triple pt, projection
    P=currentprojection) {
  if (P.infinity) {
    return unit(P.camera);
  } else {
    return unit(P.camera - pt);
  }
}


//Move a point closer to the camera.
triple towardcamera(triple pt, real distance=1, projection
    P=currentprojection) {
  return pt + distance * cameradirection(pt, P);
}

  ⋮

//Label the axes.
label("$x$", align=E, position=towardcamera(xmax*X));
label("$y$", align=N, position=towardcamera(ymax*Y));
```

The effect is much improved.

3.16 Complex scaling with `unitsize()`

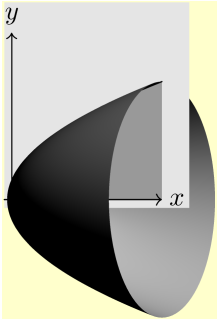
Unfortunately, Janet notices something that bothers her about this: the plane does not "contain" the labels. The issue would be fixed by making the picture a bit bigger—but if she made it too much bigger, the plane would be too big. Basically, no matter how the image is scaled, the margin at the edge of the plane needs to be about 0.4 cm.

Vincent scratches his head, but cannot devise a solution similar to what was done for the two-dimensional case. Instead, he and Janet decide to start using `unitsize()` instead of `size()` to control the scaling of the image. This means that setting the overall size of the image has to be done by trial and error, but Janet and Vincent think it's worth it to have more flexible scaling.

```
real unit = ***;
real truecm = cm / unit;
unitsize(unit);
```

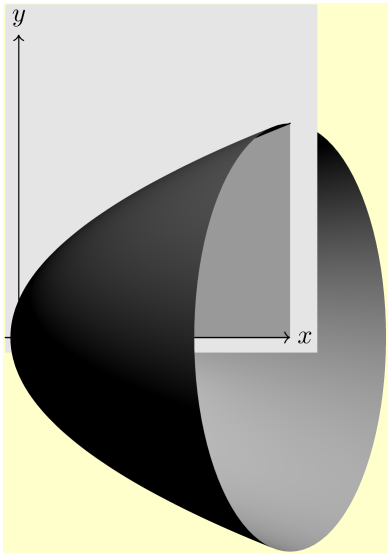
Once these are established, it is easy enough to combine scaled and unscaled lengths. For instance, `3 + 1 truecm` represents three (scaled) units plus one centimeter (which does not scale).

Using this technique, Janet is able to create a version of the image that reserves just enough space on the edge of the plane to accommodate the labels, regardless of the scaling.




```
  ⋮

real unit = 1.1cm;
real truecm = cm / unit;
unitsize(unit);

  ⋮

real xmin = -0.1;
real xmax = 2;
real ymin = -0.1;
real ymax = 2;
real margin = 0.4 truecm;

  ⋮

//Draw the plane.
path planeoutline = box((xmin, ymin), (xmax+margin,
    ymax+margin));

  ⋮
```

Compare with what happens when `unit` is set to `2cm` instead of `1.1cm`. The entire picture scales up except for the margin in the plane that contains the x and y labels, which remains exactly 0.4 cm. This is precisely the effect that Janet wanted.



3.17 Writing on surfaces

3.18 Subtleties in drawing surfaces

In this subsection, we will be concerned with various techniques to control the appearance of surfaces once constructed and positioned. The `draw` command used for surfaces has the following optional parameters we will be considering here:

type	name	default value
`material`	`surfacepen`	`currentpen`
`pen`	`meshpen`	`nullpen`
`light`	`light`	`currentlight`
`light`	`meshlight`	`nolight`

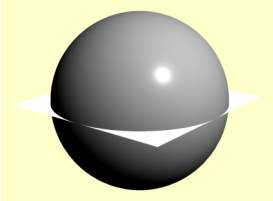
3.18.1 Shading, lighting, and material

The coloring of a surface is based on the interaction of a `material` and a light source (the parameters `surfacepen` and `light`, respectively).

Minimal rules of thumb First, Vincent introduces Janet to the following rules of thumb about using these two parameters:

- To draw flat surface in a particular color (say `lightgray`), set `surfacepen = lightgray` and `light=nolight`. If the `light` parameter is unavailable (as in specifying arrowheads), an equivalent setting is `surfacepen = emissive(lightgray)`.
- To draw a non-flat surface in a particular color, set `surfacepen` equal to a lighter version of that color. For instance, to get a light gray sphere or cube, set `surfacepen = white`.
- The parameter `light=` can usually be left alone. For more details, see 3.11, p. 75.

For instance, here's a picture of a light gray sphere with a white plane poking out of it:



```
settings.outformat = "pdf";
settings.render = 8;
import three;
size(3.5cm, 0);

draw(unitsphere, surfacepen=white);
draw(surface(box((-1,-1),(1,1))),
    surfacepen=emissive(white));
```

Janet is unsatisfied with the second rule of thumb and would like to know how to have finer control. For instance, what if she wants a surface in a lighter color than `surfacepen = white` can provide?

The answer to that question is, unfortunately, somewhat complex. Here's what Vincent is able to come up with.

First of all, the `surfacepen=` parameter that determines color is actually of type `material`, which is more complex than the type `pen` that determines color in two-dimensional drawings. (Actually `pen` determines other things too, such as line width, font size, and dashing pattern; but that's a tangent.)

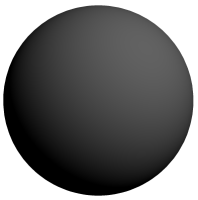
Here's the constructor for a `material` object:

```
material(pen diffusepen=black, pen ambientpen=black,
         pen emissivepen=black, pen specularpen=mediumgray,
         real opacity=opacity(diffusepen),
         real shininess=defaultshininess);
```

All parameters are optional. For almost all purposes, the only parameters that need to be dealt with are `diffusepen=`, `emissivepen=`, and `specularpen=`. Here's what they do:

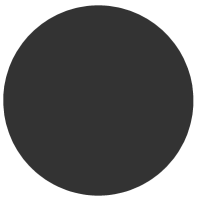
- The diffuse pen controls the way the object interacts with light sources. Portions of the surface aimed directly at the light source show up exactly this color; the color gradually fades to black as the surface approaches a

90 degree angle to the light source. It corresponds roughly to the diffuse component of the Phong reflection model[13].



```
settings.render = 8;
settings.outformat = "png";
import three;
size(2.5cm, 0);
draw(unitsphere,
    surfacepen=material(diffusepen=gray,
    emissivepen=black, specularpen=black));
```

- The emissive pen controls object coloring that is completely independent of the lighting. A surface colored with `emissivepen=red` and no other pens will show up completely red no matter what direction it is aimed. It corresponds roughly to the ambient component of the Phong reflection model.



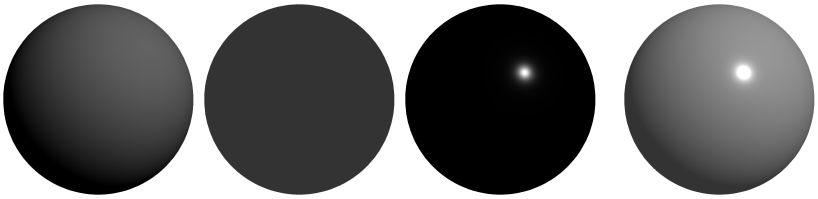
```
settings.render = 8;
settings.outformat = "png";
import three;
size(2.5cm, 0);
draw(unitsphere,
    surfacepen=material(diffusepen=black,
    emissivepen=gray(0.2),
    specularpen=black));
```

- The specular pen controls the extent to which the object gleams like a mirror. This pen only affects portions of the surface aimed directly at the light source (or close to it). It corresponds roughly to the specular component of the Phong reflection model.

```
settings.render = 8;
settings.outformat = "png";
import three;
size(2.5cm, 0);
draw(unitsphere,
    surfacepen=material(diffusepen=black,
    emissivepen=black, specularpen=white));
```

The three components are added together to form the final result:

[13] `https://en.wikipedia.org/wiki/Phong_reflection_model`



Note that the emissive component is the only one that contributes to the deepest shadow.

3.18.2 Transparency

3.18.3 Gridlines

4 Building surfaces

4.1 Predefined solids

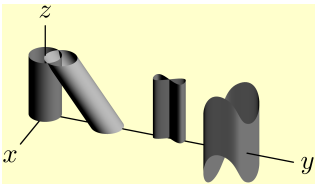
See 3.8, p. 70.

4.2 Cylinders and cones over an arbitrary base

A cylinder with an arbitrary base can be constructed using the function

```
surface extrude(path3 p, triple axis = Z);
```

The second parameter `axis` indicates the height of the cylinder, as a vector.




```
settings.outformat="png";
settings.render=16;
import three;
size(4.3, 0);

currentprojection =
    orthographic(5,2,3);
surface cyl1 = extrude(circle(c=0, r=1/2, normal=Z), axis=2Z);
surface cyl2 = extrude(circle(c=2Y, r=1/2, normal=Z), axis = 2Z
    - 1.5Y);
surface cyl3 = extrude(shift(4Y) * (-0.5Y {X} .. {X} 0.5Y ..
    cycle), axis=2Z);
surface cyl4 = extrude( shift(6Y) * ((1/2,0,0) .. (0,1/2,-1) ..
    (-1/2,0,0) .. (0,-1/2, -1) .. cycle), axis = 2Z);

draw(cyl1, white);
draw(cyl2, white);
draw(cyl3, white);
draw(cyl4, white);
```

```
draw(O -- 3Z, L=Label("$z$", position=EndPoint));
draw(O -- 8Y, L=Label("$y$", position=EndPoint));
draw(O -- 2X, L=Label("$x$", position=EndPoint));
```

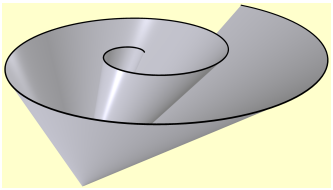
Another useful function for constructing surfaces is

```
surface extrude(path3 p, path3 q);
```

Basically, this function builds a surface that connects the two paths `p` and `q`; more precisely, it connects two points that occur at the same path time. This can cause problems if the two paths do not have the same length (in the sense of path times); in general, this function should be used only with care. However, there is one special case that is easy to use: if `v` is a point in space (i.e., a `triple`) and `g` is a `path3`, then the expression

```
extrude(g, v -- cycle)
```

constructs a cone with base `g` and vertex `v`.



```
settings.outformat="png";
settings.render=16;
import three;
size(4.3cm, 0);

currentlight = White;
path3 spiral = path3((-2,0) .. (0,7/4) .. (6/4,0) .. (0,-5/4)
    .. (-4/4,0) .. (0,3/4) .. (2/4,0) .. (0,-1/4) .. (0,0));
triple vertex = (0,-1, -2);
draw(extrude(spiral, vertex -- cycle), material(gray,
    emissivepen=gray));
draw(spiral);
```

4.3 Surfaces of revolution

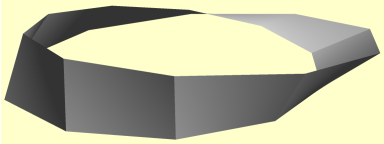
Surfaces of revolution are discussed in 3.6, p. 62.

4.4 Parametric surfaces

Parametric graphing is by far the most flexible way to produce a surface. Of all the kinds of surfaces discussed so far, the only ones that cannot easily be imitated by a single parametric surface are the cube and the solid cone: the solid cone would require two parametric surfaces, while the unit cube would require six (one for each face). The primary disadvantage of using parametric surfaces is that they require a certain amount of mathematics beforehand to determine the

right formulas. Janet thinks this is fine; Vincent is a bit intimidated, although he will find that complex programming can often be substituted for complex mathematics.

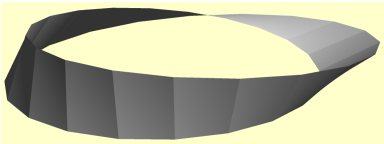
Parametric graphing requires the `graph3` module (which automatically imports the `three` module). Here's an example in which parametric equations are used to draw a Möbius band:



```
settings.outformat="png";
settings.prc=false;
settings.render=8;
size(5cm,0);

import graph3;

triple F(pair uv) {
  real t = uv.x;
  real r = uv.y;
  return (cos(t) + r*cos(t)*sin(t/2),
          sin(t) + r*sin(t)*sin(t/2),
          r*cos(t/2));
}
real r = 0.2;
surface moeb = surface(F, (0,-r), (2pi,r));
draw(moeb, surfacepen=material(white, emissivepen=0.2 white));
```

The key function from the `graph` module is called `surface()`. Its required parameters are a function of type `triple(pair)` and two ordered pairs `a` and `b`. The ordered pairs specify the domain over which the function should be graphed; in the code above, the function is graphed over the domain $0 \leq u \leq 2\pi$, $-r \leq v \leq r$.

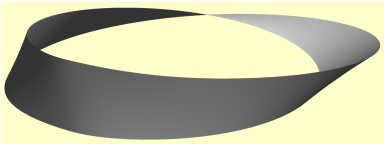
Additional optional parameters include `nu` and `nv`, which set the resolution of the graph. Both of these parameters default to the variable `nmesh`, which is normally 10. Changing only `nu` will automatically change `nv` as well; to change only `nu`, you should pass in `nv=nmesh`. Here's the result of using these parameters to produce a somewhat nicer picture of the Möbius band:



```
  ⋮
surface moeb = surface(F, (0,-r),
    (2pi,r), nu=20, nv=1);
  ⋮
```

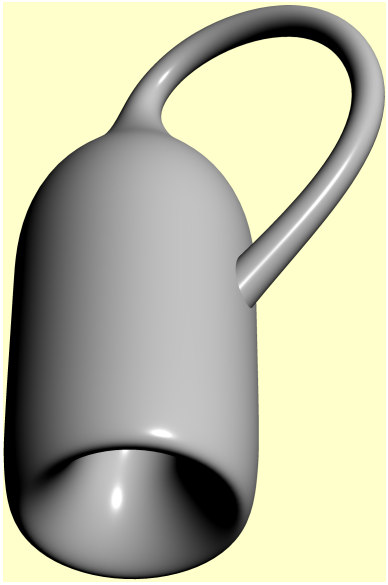
While a definite improvement, this picture still does not appear smooth. The way to get Asymptote to plot a smooth surface is to add the word `Spline` after

all the other parameters[14]. The parameters `nu` and `nv` can still be adjusted, but in this case that is unnecessary:



```
  ⋮
surface moeb = surface(F, (0,-r),
    (2pi,r), Spline);
  ⋮
```

While Janet is very comfortable with this, Vincent dislikes the amount of algebra required to obtain the parametric equations for the Möbius band. Here is an example of a parametric function that is defined by programming rather than by formulas:



```
settings.outformat="png";
settings.render=8;
settings.prc=false;

size(5cm,0);

import graph3;
currentprojection =
    perspective(10,5,-8);

path center_path = (0,0) --- (0,3)
    .. (1.5,3) .. (0,0.3) --- (0,0);

real bottomradius = 0.6;
real topradius = 0.1;
```

```
path radius_graph = (0,bottomradius) {up} ::
    (0.3,1.2*bottomradius) --- (0.6, 1.2*bottomradius) ::
    (1.2,topradius) --- (2,topradius) :: {up} (3,bottomradius);
radius_graph = xscale(1/3) * radius_graph;
radius_graph = (shift(-1,0)*radius_graph) & radius_graph &
    (shift(1,0)*radius_graph);

real radius(real t) {
  return point(radius_graph, times(radius_graph, t)[0]).y;
}
```

[14]Exception: if you are using a `bool(pair)` condition to exclude part of the rectangular domain, that condition should be specified after `Spline`.

```
triple F(pair w) {
  real t = w.x % 2.0;
  bool reverse = (t >= 1.0);
  t %= 1.0;
  real relt = reltime(center_path, t);
  real theta = w.y;
  triple center = YZplane(point(center_path, relt));
  pair tangent = dir(center_path, relt);
  if (reverse) tangent *= -1;
  triple normal = X;
  triple binormal = cross(YZplane(tangent), normal);
  triple v = normal*cos(theta) + binormal*sin(theta);

  return center + radius(t)*v;
}

surface kleinbottle = surface(F, (0.5,0), (1.5, 2pi), nu=32,
    nv=16, Spline);

draw(kleinbottle, white);
```

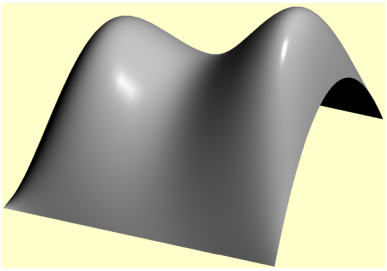
Note, in particular, that the helper function `radius()` is not defined by a formula; instead, its graph is created as a `path` (which is never drawn), and the `times()` function is used to find where that graph intersects a given vertical line in order to evaluate the function. This trick is expensive for the computer, but can be useful when you have a good idea what a function's graph should look like and don't want to have to come up with a formula for it.

4.5 Graphs of functions of two variables

Consider the function

$$f(x,y) = \left(\tfrac{6}{5} - \tfrac{1}{2}x^2\right) \cdot \left(-\tfrac{1}{2}y^4 + \tfrac{1}{15}y^3 + y^2 + \tfrac{1}{5}y + 1\right)$$

which Janet designed for a calculus test; it has two local maxima and one saddle point within the domain $\left(-\sqrt{\frac{12}{5}}, \sqrt{\frac{12}{5}}\right) \times \left(-\sqrt{\frac{12}{5}}, \sqrt{\frac{12}{5}}\right)$. Here's how she can graph it using the `graph3` module:



```
settings.outformat="png";
settings.render=4;
settings.prc=false;
size(5cm,0);
import graph3;
currentprojection =
    orthographic(10,4,7);
```

```
real f(pair xy) {
  real x = xy.x; real y = xy.y;
  return (6/5 - x^2/2) * (-y^4/2 + y^3/15 + y^2 + y/5 + 1);
}
real xmax = sqrt(12/5);  real xmin = -xmax;
real ymax = sqrt(12/5);  real ymin = -ymax;
surface s = surface(f, (xmin,ymin), (xmax,ymax), Spline);
draw(s, surfacepen=white);
```

Again, the key function from the `graph3` module is called `surface()`. Its required parameters are a function of type `real(pair)` and two ordered pairs `a` and `b`. The ordered pairs specify the corners of the rectangular domain over which the function should be graphed.

Additional optional parameters include `nx` and `ny`, which set the resolution of the graph. Both of these parameters default to the variable `nmesh`, which is normally 10. Changing only `nx` will automatically change `ny` as well; to change only `nx`, you should pass in `ny=nmesh`.

Passing in the parameter `Spline`, as above, causes a smooth surface to be drawn. Sometimes the smooth surface may appear wavy; the simplest way to counteract this is to increase `nx` and/or `ny`. (Sometimes it may be necessary to increase `nx` and/or `ny` to the point that the surface appears smooth even without the `Spline` option.)

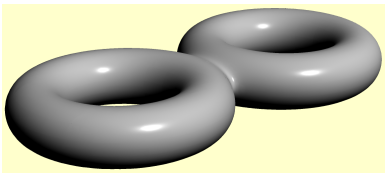
4.6 Implicitly defined surfaces

Implicitly defined surfaces may be drawn using either the older module `contour3` or, courtesy of yours truly, the newer module `smoothcontour3`. The newer module produces nicer results if the function is differentiable (and sometimes if it is not—no guarantees), but typically takes longer to compute.

4.6.1 The smoothcontour3 module

The `smoothcontour3` module is so new that many Asymptote installations will not include it. (It was first included in version 2.33.) In Janet's case, the code below would not work for her until she copied the `smoothcontour3.asy` file from `http://github.com/charlesstaats/smoothcontour3` into the same directory as the `asy` file she was trying to compile. Here's how to graph

an implicitly defined function using the `smoothcontour3` module. (This example was created by the author for use both in this tutorial and in package documentation, including in the official Asymptote manual. The example draws from `http://math.stackexchange.com/a/349914/455` and `http://mathematica.stackexchange.com/a/20913`.)



```
settings.outformat="png";
settings.render=8;
import smoothcontour3;

size(5cm, 0);
currentprojection=perspective((18,20,10));

real tuberadius = 0.69;

// Convert to cylindrical coordinates to draw
// a circle revolved about the z axis.
real toruscontour(real x, real y, real z) {
  real r = sqrt(x^2 + y^2);
  return (r-2)^2 + z^2 - tuberadius^2;
}

// Take the union of the two tangent tori (by taking
// the product of the functions defining them). Then
// add (or subtract) a bit of noise to smooth things
// out.
real f(real x, real y, real z) {
  real f1 = toruscontour(x - 2 - tuberadius, y, z);
  real f2 = toruscontour(x + 2 + tuberadius, y, z);
  return f1 * f2 - 0.1;
}

// The smoothed function extends a bit farther than the union of
// the two tori, so include a bit of extra space in the box.
triple max = (2*(2+tuberadius), 2+tuberadius, tuberadius) +
    (0.1, 0.1, 0.1);

// Draw the implicit surface.
surface s = implicitsurface(f, -max, max, overlapedges=true,
                            nx=20, nz=5);
draw(s, surfacepen=white);
```

The key function is called `implicitsurface()`. Its required parameters are a function of type either `real(triple)` or `real(real, real, real)` and two ordered triples `a` and `b`. The ordered triples specify the rectangular solid over

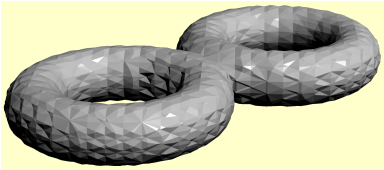
which the graph should be plotted.

Additional optional parameters include `nx`, `ny`, and `nz`, which set the initial resolution of the graph. The algorithm is adaptive, so the mesh can and probably will be finer in some places. But it is helpful if, for instance, the initial rectangular solids are close to being cubes. Setting one of these three parameters does not affect the other two; to set them all at once, use the optional parameter `n`.

The optional parameter `overlapedges` deserves special note. As an artifact, the rendering engine will often show gaps between patches of the surface, even if mathematically the edges are exactly aligned. Setting `overlapedges=true` can compensate, at least partially. For additional details, see the package documentation.

4.6.2 The contour3 module

For completeness, Janet also tries out the older `contour3` module (which does not have to be downloaded separately). Here is how it plots the same function:



```
settings.outformat="png";
settings.render=8;

import contour3;
size(5cm, 0);
```

⋮

```
// Draw the implicit surface.
surface s = surface(contour3(f, -max, max, nx=20, ny=10, nz=5));
draw(s, surfacepen=white);
```

The compilation was vastly faster than that of the previous image. However, when Janet attempts to increase the mesh resolution to approximate a smooth appearance, she finds that the compilation runs out of memory long before the appearance reaches anything like the smoothness of the example using `smoothcontour3`.

There are two key functions for this module, called `contour3()` and `surface()`. The required parameters of `contour3()` are a function of type `real(real, real, real)` and two ordered triples `a` and !b!, the diagonally opposite corners of the plotting domain. The optional parameters `nx`, `ny`, and `nz` control the mesh size. The default is 10; but note that changes in `nx` are reflected in `ny` and `nz` if they are not set.

The `surface()` function in question takes exactly one parameter: whatever[15] was produced by `contour3()`.

[15] This is actually a `vertex[][]`, where `vertex` is a type defined in the `contour3` module.

4.7 Cropping surfaces

A The complete code

Here is the complete code for the image on page 3 (including the yellow background).

```
1  //Function to return a brace path
2  real innerangle = radians(60);
3  real outerangle = radians(70);
4  real midangle = radians(0);
5  path brace(pair a, pair b, real amplitude = .14*length(b-a)) {
6    transform t = identity();
7    real length = length(b-a);
8    real sign = 1;
9    if (amplitude < 0) {
10     //    amplitude *= -1;
11     sign = -1;
12   }
13   path brace = (0,0){expi(sign*outerangle)} ::
       {expi(sign*midangle)}(length/4, amplitude/2)
14           :: {expi(sign*innerangle)} (length/2, amplitude)
       {expi(-sign*innerangle)}
15   :: {expi(-sign*midangle)}(3*length/4, amplitude/2) ::
       {expi(-sign*outerangle)} (length,0);
16   real angle = degrees(atan2((b-a).y, (b-a).x));
17   t = rotate(angle)*t;
18   t = shift(a) * t;
19   return t * brace;
20 }

22 //Define the command drawshifted, to be used later
23 void drawshifted(path g, pair trueshift, picture pic =
       currentpicture, Label label="", pen pen=currentpen,
       arrowbar arrow=None, arrowbar bar=None, margin
       margin=NoMargin, marker marker=nomarker)
24 {
25   pic.add(new void(frame f, transform t) {
26       picture opic;
27       draw(opic, L=label, shift(trueshift)*t*g, p=pen,
       arrow=arrow, bar=bar,
28          margin=margin, marker=marker);
29       add(f,opic.fit());
30     });
31   pic.addBox(min(g), max(g), trueshift+min(pen),
       trueshift+max(pen));
```

```
}

usepackage("amsmath");

real yellowPart = 0.2;
real unit = 2cm;
real truecm = cm / unit;
unitsize(unit);
pen backgroundpen = yellowPart*yellow + (1-yellowPart)*white;
frame finish() {
  currentlight.background = backgroundpen;
  frame toreturn = bbox(backgroundpen, Fill);
  currentpicture = new picture;
  unitsize(unit);
  return toreturn;
}

/*------------------------------*/

//Basic settings
settings.outformat="pdf";
defaultpen(fontsize(10pt));
import graph;

//Save some important numbers.
real xmin = -0.1;
real xmax = 2;
real ymin = -0.1;
real ymax = 2;

//Draw the graph and fill the area under it.
real f(real x) { return sqrt(x); }
path s = graph(f, 0, 2, operator..);
path fillregion = s -- (xmax,0) -- cycle;
pen fillpen = mediumgray;
fill(fillregion, fillpen);
draw(s, L=Label("$y=f(x)$", position=EndPoint));

//Fill the strip of width dx
real x = 1.4;
real dx = .05;
real t0 = times(s,x)[0];
real t1 = times(s,x+dx)[0];
path striptop = subpath(s,t0,t1);
filldraw((x,0) -- striptop -- (x+dx,0) --  cycle, black);
```

```
78 //Draw the bars labeling the width dx
79 real barheight = f(x+dx);
80 pair barshifty = (0, 0.2cm);
81 Label dxlabel = Label("$dx$", position=MidPoint, align=2N);
82 drawshifted((x,barheight) -- (x+dx, barheight),
       trueshift=barshifty, label=dxlabel, bar=Bars);

84 //Draw the arrows pointing inward toward the dx label
85 real myarrowlength = 0.3cm;
86 margin arrowmargin = DotMargin;
87 path leftarrow = shift(barshifty) * ((-myarrowlength, 0) --
       (0,0));
88 path rightarrow = shift(barshifty) * ((myarrowlength, 0) --
       (0,0));
89 draw((x, barheight), leftarrow, arrow=Arrow(),
       margin=arrowmargin);
90 draw((x+dx, barheight), rightarrow, arrow=Arrow(),
       margin=arrowmargin);

92 //Draw the bar labeling the height f(x)
93 real barx = x + dx;
94 pair barshiftx = (0.42cm, 0);
95 Label fxlabel = Label("$f(x)$", align=(0,0), position=MidPoint,
       filltype=Fill(fillpen));
96 drawshifted((barx,0) -- (barx, f(x)), trueshift=barshiftx,
       label=fxlabel, arrow=Arrows(), bar=Bars);

98 //Draw the axes on top of everything that has gone before
99 arrowbar axisarrow = Arrow(TeXHead);
100 Label xlabel = Label("$x$", position=EndPoint);
101 draw((xmin,0) -- (xmax,0), arrow=axisarrow, L=xlabel);
102 Label ylabel = Label("$y$", position=EndPoint);
103 draw((0,ymin) -- (0,ymax), arrow = axisarrow, L=ylabel);

105 //Draw the tick mark on the x-axis
106 path tick = (0,0) -- (0,-0.15cm);
107 Label ticklabel = Label("$x$", position=EndPoint);
108 draw((x,0), tick, L=ticklabel);

110 frame pic2dFrame = finish();

112 /* ------------------------------------------------------ */

114 settings.prc = false;
115 settings.render=16;
116 import three;
```

```
currentprojection = orthographic(5,0,10, up=Y);
//currentprojection=oblique;
//currentprojection=perspective(6,0,10,up=Y);

pen color = white;
material surfacepen = material(diffusepen=color+opacity(1.0),
    emissivepen=0.2*color);
material planepen = material(diffusepen=opacity(0.6),
    emissivepen=0.8*color);
pen diskpen = black+opacity(1.0);

path3 p3 = path3(s);
draw(p3);

surface FilledRegion = surface(fillregion);
draw(FilledRegion, surfacepen = gray(0.6) + opacity(0.8));

surface solidsurface = surface(p3, c=0, axis=X);
draw(solidsurface, surfacepen=surfacepen);

/*
int n = length(p3);
for (real i = 0; i <= n; i += n/10) {
  if (i >= n) i -= .01;
  draw(solidsurface.vequals(i), gray(0.3));
}
*/
draw(solidsurface.vequals(length(p3) - .001), gray(0.3));

real extra = 0.4 truecm;
path planeboundary = (xmin,ymin) -- (xmax+extra,ymin) --
    (xmax+extra,ymax+extra) -- (xmin,ymax+extra) -- cycle;
path planeoutside = planeboundary -- fillregion -- cycle;
draw(surface(planeoutside), surfacepen=planepen);

transform pushoutside = shift(0,.001);
striptop = pushoutside*striptop;
path3 dVtop = path3(striptop);
path3 openStrip = (x,0,0) -- dVtop -- (x+dx,0,0);
surface disk = surface(openStrip, c=0, axis=X);
draw(disk, diskpen);

triple cameraDirection(triple pt, projection P =
    currentprojection) {
  if (P.infinity) {
```

```
159     return unit(P.camera);
160   } else {
161     return unit(P.camera - pt);
162   }
163 }

165 triple towardCamera(triple pt, real dist = 1 truecm, projection
        P = currentprojection) {
166   return pt + dist*cameraDirection(pt, P);
167 }

169 draw(xmin*X -- xmax*X, arrow=Arrow3(TeXHead2(normal=Z)));
170 draw(ymin*Y -- ymax*Y, arrow=Arrow3(TeXHead2(normal=Z)));
171 label("$x$", position=towardCamera(xmax*X), align = E);
172 label("$y$", position=towardCamera(ymax*Y), align=N);

174 frame pic3dFrame = finish();

176 /* ------------------------------------------------- */

178 currentprojection=orthographic((3,0,10), up=Y);

180 diskpen = mediumgray;
181 draw(disk, diskpen);

183 transform3 T = rotate(10, X);
184 path3 brace = T*path3(brace((x+dx,barheight), (x+dx,0)));
185 draw(brace--cycle);
186 label("$r=f(x)$", position=midpoint(brace), align=E);

188 //Draw the bars labeling the width dx
189 path3 dxlabelpath = T * ((x, barheight, 0) -- (x+dx, barheight,
        0));
190 draw(dxlabelpath, L=dxlabel, Bars3);

192 arrow(relpoint(dxlabelpath,0), dir=W, length=myarrowlength,
        margin=DotMargin3, arrow=Arrow3(emissive(black)));
193 arrow(relpoint(dxlabelpath,1), dir=E, length=myarrowlength,
        margin=DotMargin3, arrow=Arrow3(emissive(black)));

195 draw(xmin*X -- xmax*X, arrow=Arrow3(TeXHead2(normal=Z)));
196 draw(ymin*Y -- ymax*Y, arrow=Arrow3(TeXHead2(normal=Z)));
197 label("$x$", position=towardCamera(xmax*X), align = E);
198 label("$y$", position=towardCamera(ymax*Y), align=N);

200 frame oneSlice = finish();
```

```
/* ------------------------------------------------- */

label(minipage("\raggedright Dimensions of infinitesimally thin
    sheet:
\begin{description}
\item[Area:] $\pi r^2 = \pi [f(x)]^2$
\item[Thickness:] $dx$
\item[Volume:] $dV = \text{Area}\cdot\text{thickness} = \pi
    [f(x)]^2\;dx$
\end{description}"
,6cm));

frame labelFrame = finish();

/* ------------------------------------------------- */

unit = 1;
unitsize(unit);
add(pic3dFrame);
add(labelFrame, position=(max(pic3dFrame).x, min(pic3dFrame).y
    - 1cm), align=SW);
pic3dFrame = finish();

/* ------------------------------------------------- */

//unitsize(1);    // Set the usual (postscript) coordinates.
add(pic2dFrame);
add(pic3dFrame, position=max(pic2dFrame), align=SE);
add(oneSlice, position=min(pic2dFrame)+(0,-1cm), align=SE);

// Scale up by 4 in order to increase resolution.
shipout(scale(4)*finish());
```

B Installing Asymptote

• For a Mac OS X system, Asymptote is automatically installed during a standard installation of MacTeX. The version installed will be updated only when TeX Live is rebuilt and re-installed, which is typically once a year. (Running TeX Live Utility will not update Asymptote.) If you want the cutting-edge version of Asymptote, or if the TeX Live installation does not work for whatever reason, see the following information:

– This website[16] provides pre-built binaries, albeit with some specific dependencies that many systems may not satisfy.

– Section 2.6 of the manual[17] gives information on compiling from the source code. This is the recommended method, but does not always interact well with the most up-to-date version of Mac OS X. As of this writing, version 3.31 of Asymptote works with the latest OS, but previous versions of Asymptote do not.

– There is information at `http://sourceforge.net/p/asymptote/discussion/409349/thread/4542fb8e` on installing Asymptote using Homebrew. Theoretically, this should be quite easy; however, the author of this tutorial was unable to get this to work, and the software reported that it was attempting to install Asymptote version 2.23 at a time when version 2.24 had been available for several months (but not yet on TeX Live).

• For a Windows system, the official installation instructions are fairly good. As of this writing, the most recent version of the `setup.exe` file can be downloaded from `http://sourceforge.net/projects/asymptote/files/2.35/`.

• For a Unix-like system, a version of Asymptote is included in TeX Live, but there may be additional dependencies; see, for instance, `http://tex.stackexchange.com/a/155284/484`. You should also consult the following two pages from the official documentation:

– `http://asymptote.sourceforge.net/doc/UNIX-binary-distributions.html` gives information on existing binary distributions of Asymptote.

– `http://asymptote.sourceforge.net/doc/Compiling-from-UNIX-source.html` gives information on compiling Asymptote directly from the source code.

[16] `http://asymptote.sourceforge.net/doc/MacOS-X-binary-distributions.html`
[17] `http://asymptote.sourceforge.net/doc/Compiling-from-UNIX-source.html`