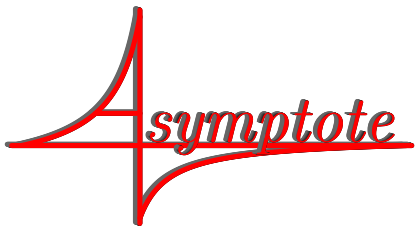
Asymptote

For version 0.83



This file documents `Asymptote`, version 0.83.

http://asymptote.sourceforge.net

Table of Contents

1 Description

Asymptote is a powerful script-based vector graphics language that provides a natural coordinate-based framework for technical drawings. Labels and equations are typeset with LaTeX, for overall document consistency, yielding the same high-quality level of typesetting that LaTeX provides for scientific text. By default it produces PostScript output, but it can also generate any format that the ImageMagick package can produce.

A major advantage of Asymptote over other graphics packages is that it is a high-level programming language, as opposed to just a graphics program: it can therefore exploit the best features of the script (command-driven) and graphical-user-interface (GUI) methods for producing figures. The rudimentary GUI xasy included with the package allows one to move script-generated objects around. To make Asymptote accessible to the average user, this GUI is currently being developed into a full-fledged interface that can generate objects directly. However, the script portion of the language is now ready for general use by users who are willing to learn a few simple Asymptote graphics commands (see Chapter 5 [Drawing commands], page 62).

Asymptote is mathematically oriented (e.g. one can use complex multiplication to rotate a vector) and uses LaTeX to do the typesetting of labels. This is an important feature for scientific applications. It was inspired by an earlier drawing program (with a weaker syntax & capabilities) called MetaPost.

Many of the features of Asymptote are written in the Asymptote language itself. While the stock version of Asymptote is designed for mathematics typesetting needs, one can write Asymptote modules that tailor it to specific applications. A scientific graphing module has already been written (see Section 4.14.3 [graph], page 38). Examples of Asymptote code and output are available at

```
http://asymptote.sourceforge.net/gallery/
```

The Asymptote vector graphics language provides:

- a natural coordinate-based framework for technical drawings, inspired by MetaPost, with a much cleaner, powerful C++-like programming syntax;
- LaTeX typesetting of labels, for overall document consistency;
- compilation of figures into virtual machine code for speed, without sacrificing portability;
- the power of a script-based language coupled to the convenience of a GUI;
- customization using its own C++-like graphics programming language;
- sensible defaults for graphical features, with the ability to override;
- a high-level mathematically oriented interface to the PostScript language for vector graphics, including affine transforms and complex variables;
- functions that can create new (anonymous) functions;
- deferred drawing that uses the simplex method to solve overall size constraint issues between fixed-sized objects (labels and arrowheads) and objects that should scale with figure size;
- a standard for typesetting mathematical figures, just as TEX/LaTeX is the de-facto standard for typesetting equations.

2 Installation

Here are the commands that the root user can use to install the binary distribution of version `x.xx` of `Asymptote` for a specific platform `ARCH` in `/usr/local`. The executable file will be `/usr/local/bin/asy` (the optional `texhash` and `install-info` commands install a LaTeX style file and man pages):

```
tar -C / -zxf asymptote-x.xx.ARCH.tar.gz
texhash
install-info --infodir=/usr/local/info /usr/local/info/asymptote.info
```

To uninstall:

```
install-info --delete --infodir=/usr/local/info \
  /usr/local/info/asymptote.info
tar -zxvf asymptote-x.xx.ARCH.tar.gz | xargs rm
texhash
```

To compile and install `Asymptote` from a source release `x.xx`, first execute the commands:

```
tar -zxf asymptote-x.xx.tar.gz
cd asymptote-x.xx
```

Then put `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_source/gc6.5.tar.gz` in the current directory and

```
./configure
make all
make install
```

For a (default) system-wide installation, the last command should be done as root. The above steps will compile an optimized single-threaded static version of the Boehm garbage collector (`http://www.hpl.hp.com/personal/Hans_Boehm/gc/`). Alternatively, one can request use of a (presumably multithreaded and therefore slower) system version of the Boehm garbage collector by configuring instead with `./configure --enable-gc=system`. One can disable use of the garbage collector by configuring with `./configure --disable-gc`. For a list of other configuration options, say `./configure --help`.

Debian users can install Hubert Chan's `Asymptote` package:

`http://www.uhoreg.ca/programming/debian.text`

If you are compiling `Asymptote` with `gcc`, you will need a relatively recent version (e.g. 3.2 or later). If you get errors compiling `interact.cc`, try installing an up-to-date version of the GNU `readline` library or else uncomment `HAVE_LIBREADLINE` in `config.h`.

The `FFTW` library is only required if you want `Asymptote` to be able to take Fourier transforms of data (say, to compute an audio power spectrum).

If you don't want to install `Asymptote` system wide, just make sure the compiled binary `asy` and GUI script `xasy` are in your path and set the environment variable `ASYMPTOTE_DIR` to point to the directory `base` (in the top level directory of the `Asymptote` source code). In looking for `Asymptote` system files, `asy` will search the following paths, in the order listed:

1. current directory
2. `$ASYMPTOTE_DIR`
3. system-wide directory (default: `/usr/local/share/asymptote`)

In interactive mode, or when given the -V option, Asymptote will attempt to open up a PostScript viewer by first trying to execute the program specified by the optional environment variable ASYMPTOTE_PSVIEWER, then gv, ggv, ghostview, and finally gsview. To support interactive mode, the PostScript viewer must be able to redraw a file whenever it changes. The default PostScript viewer gv supports this automatically (via SIGHUP); users of ggv will need to enable Watch file under Edit/Postscript Viewer Preferences. For viewing PDF format output, there is an ASYMPTOTE_PDFVIEWER environment variable and a PDF viewer list gv, acroread, and xpdf.

The patches supplied in the patches directory fix known bugs in gv-3.5.8 and gv-3.6.1, most notably the backwards-incompatible command line options of gv-3.6.1. Another bug in gv-3.6.1 requires it to be explicitly configured with ./configure --enable-signal-handle for it to work properly with Asymptote's interactive mode.

Users of emacs can edit Asymptote code with the mode asy-mode, after enabling it by putting these two lines in the ~/.emacs initialization file, replacing ASYDIR with the location of the Asymptote example files (by default, /usr/local/share/doc/asymptote):

```
(autoload 'asy-mode "ASYDIR/asy-mode.el" "Asymptote major mode." t)
(setq auto-mode-alist (cons (cons "\\.asy$" 'asy-mode) auto-mode-alist))
```

Fans of vim can

```
cp /usr/local/share/doc/asymptote/asy.vim ~/.vim/syntax/asy.vim
```

and add the following to their ~/.vimrc file:

```
augroup filetypedetect
au BufNewFile,BufRead *.asy     setf asy
augroup END
filetype plugin on
```

If any of these directories or files don't exist, just create them. To set vim up to run the current asymptote script using :make just add to ~/.vim/ftplugin/asy.vim:

```
setlocal makeprg=asy\ %
setlocal errorformat=%f:\ %l.%c:\ %m
```

To uninstall all Asymptote files, type:

```
make uninstall
```

The following commands are needed to install the latest development version of Asymptote from cvs (when prompted for the CVS password, type enter):

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/asymptote login

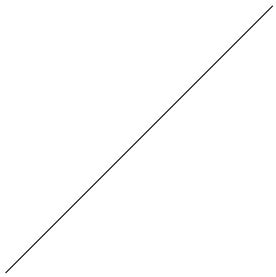
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/asymptote co asymptote
cd asymptote-x.xx
./autogen.sh
wget http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_source/gc6.4.tar.gz
./configure
make all
make install
```

To compile without optimization, use the command make CFLAGS=-g.

3 Examples

To draw a line from coordinate (0,0) to coordinate (100,100) using `Asymptote`'s interactive mode, type:

```
asy
draw((0,0)--(100,100));
```



The units here are `PostScript` "big points" (1 `bp` = 1/72 `inch`); `--` means join with a linear segment.

At this point you can type in further draw commands, which will be added to the displayed figure, or type `quit` to exit interactive mode. You can use the arrow keys in interactive mode to edit previous lines (assuming that you have support for the GNU readline library enabled). Further commands specific to interactive mode are described in Chapter 8 [Interactive mode], page 75.

In batch mode, `Asymptote` reads commands directly from a file. To try this out, type

```
draw((0,0)--(100,100));
```

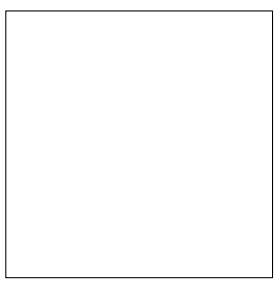
into a file, say test.asy. Then execute this file by typing

```
asy -V test
```

The `-V` option opens up a `PostScript` viewer window so you can immediately view the encapsulated `PostScript` output. By default the output will be written to the file `test.eps`; the prefix of the output file may be changed with the `-o` command line option.

One can draw a line with more than two points and create a cyclic path like this square:

```
draw((0,0)--(100,0)--(100,100)--(0,100)--cycle);
```



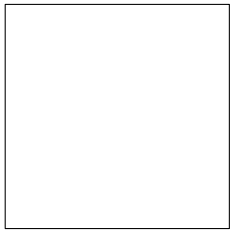
It is often inconvenient to work directly with PostScript coordinates. The next example draws a unit square scaled to width 101 bp and height 101 bp. The output is identical to that of the previous example.

```
size(101,101);
draw((0,0)--(1,0)--(1,1)--(0,1)--cycle);
```

For convenience, the path `(0,0)--(1,0)--(1,1)--(0,1)--cycle` may be replaced with the predefined variable `unitsquare`, or equivalently, `box((0,0),(1,1))`.

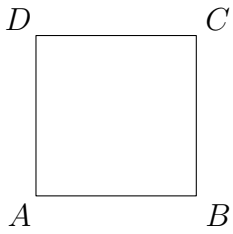
One can also specify the size in `pt` (1 `pt` = 1/72.27 `inch`), `cm`, `mm`, or `inches`. If 0 is given as a size argument, no restriction is made in that direction; the overall scaling will be determined by the other direction: See [size], page 21.

```
size(0,3cm);
draw(unitsquare);
```



Adding labels is easy in `Asymptote`; one specifies the label as a double-quoted `LaTeX` string, a coordinate, and an optional alignment direction:

```
size(0,3cm);
draw(unitsquare);
label("$A$",(0,0),SW);
label("$B$",(1,0),SE);
label("$C$",(1,1),NE);
label("$D$",(0,1),NW);
```



See section Section 4.14.3 [graph], page 38 (or the online `Asymptote` gallery at `http://asymptote.sourceforge.net`) for further examples, including two-dimensional scientific graphs.

4 Programming

Here is a short introductory example to the `Asymptote` programming language that highlights the similarity of its control structures with those of C and C++.

```
// This is a comment.

// Declaration: Declare x to be a real variable;
real x;

// Assignment: Assign the real variable x the value 1.
x=1.0;

// Conditional: Test if x equals 1 or not.
if(x == 1.0) {
  write("x equals 1.0");
} else {
  write("x is not equal to 1.0");
}

// Loop: iterate 10 times
for(int i=0; i < 10; ++i) {
  write(i);
}
```

Another example of a loop, together with user-defined functions, is provided in the file `wheelanimation.asy` in the examples directory (by default `/usr/local/share/doc/asymptote`). This example uses the `gifmerge` command to `merge` multiple images into a gif animation, using the `ImageMagick convert` program.

`Asymptote` also supports `while`, `do`, `break`, and `continue` statements just as in C/C++. In addition, it supports many features beyond the ones found in those languages.

4.1 Data types

`Asymptote` supports the following data types (in addition to user-defined types):

`void` The void type is used only by functions that take or return no arguments.

`bool` a boolean type that can only take on the values `true` and `false`. For example:

```
bool b=true;
```

defines a boolean variable `b` and initializes it to the value `true`. If no initializer is given:

```
bool b;
```

the value `false` is assumed.

`int` an integer type; if no initializer is given, the implicit value `0` is assumed.

`real` a real number; this should be set to the highest-precision native floating-point type on the architecture. The implicit initializer for type real is `0.0`.

`pair` complex number, that is, an ordered pair of real components `(x,y)`. The real and imaginary parts of a pair `z` can read as `z.x` and `z.y`. We say that `x` and `y` are virtual members of the data element pair; they cannot be directly modified, however. The implicit initializer for type pair is `(0.0,0.0)`.

There are a number of ways to take the complex conjugate of a pair:

```
pair z=(3,4);
z=(z.x,-z.y);
z=z.x-I*z.y;
z=conj(z);
```

A number of built-in functions are defined for pairs:

`pair conj(pair z)`
: returns the conjugate of `z`;

`real length(pair z)`
: returns the complex modulus `|z|` of its argument `z`. For example,

```
pair z=(3,4);
write(length(z));
```

produces the result 5. A synonym for `length(pair)` is `abs(pair)`;

`real angle(pair z)`
: returns the angle of `z` in radians;

`real degrees(pair z)`
: returns the angle of `z` in degrees in the interval [0,360);

`pair unit(pair z)`
: returns a unit vector in the direction of the pair `z`;

`pair expi(real angle)`
: returns a unit vector in the direction `angle` measured in radians;

`pair dir(real angle)`
: returns a unit vector in the direction `angle` measured in degrees;

`real xpart(pair z)`
: returns `z.x`;

`real ypart(pair z)`
: returns `z.y`;

`real dot(pair a,pair b)`
: returns the dot product `a.x*b.x+a.y*b.y`.

`triple` an ordered triple of real components `(x,y,z)` used for three-dimensional drawings. The respective components of a triple `v` can read as `v.x`, `v.y`, and `v.z`.

Here are the built-in functions for triples:

`real length(triple v)`
: returns the length `|v|` of the vector `v`.

`real polar(triple v)`
: returns the colatitude of `v` measured from the z axis in radians;

`real azimuth(triple v)`
: returns the longitude of `v` measured from the x axis in radians;

`real colatitude(triple v)`
: returns the colatitude of `v` measured from the z axis in degrees;

`real latitude(triple v)`
: returns the latitude of `v` measured from the xy plane in degrees;

`real longitude(triple v)`
: returns the longitude of `v` measured from the x axis in degrees;

`triple unit(triple v)`
: returns a unit triple in the direction of the triple `v`;

`triple dir(real colatitude, real longitude)`
: returns a unit triple in the direction `(colatitude,longitude)` measured in degrees;

`real xpart(triple v)`
: returns `v.x`;

`real ypart(triple v)`
: returns `v.y`;

`real zpart(triple v)`
: returns `v.z`;

`real dot(triple a,triple b)`
: returns the dot product `a.x*b.x+a.y*b.y+a.z*b.z`;

`real cross(triple a,triple b)`
: returns the cross product `(a.y*b.z-a.z*b.y,a.z*b.x-a.x*b.z,a.x*b.y-b.x*a.y)`.

`string` a character string, implemented using the STL `string` class.

Strings delimited by double quotes (`"`) are subject to the following mapping to allow the use of double quotes in TEX (e.g. for using the `babel` package, see Section 4.14.18 [babel], page 59):

- `\"` maps to `"`

Strings delimited by single quotes (`'`) have the same mappings as character strings in ANSI `C`:

- `\'` maps to '
- `\"` maps to "
- `\?` maps to ?
- `\\` maps to backslash
- `\a` maps to alert
- `\b` maps to backspace
- `\f` maps to form feed
- `\n` maps to newline

- \r maps to carriage return
- \t maps to tab
- \v maps to vertical tab
- \0-\377 map to corresponding octal byte
- \x0-\xFF map to corresponding hexadecimal byte

The implicit initializer for type string is the empty string `""`. In the following string functions, position 0 denotes the start of the string.

`int length(string s)`
: returns the length of the string `s`;

`int find(string s, string t, int pos=0)`
: returns the position of the first occurrence of string `t` in string `s` at or after position `pos`, or -1 if `t` is not a substring of `s`;

`int rfind(string s, string t, int pos=-1)`
: returns the position of the last occurrence of string `t` in string `s` at or before position `pos` (if `pos`=-1, at the end of the string `s`), or -1 if `t` is not a substring of `s`;

`string insert(string s, int pos, string t)`
: return the string formed by inserting string `t` at position `pos` in `s`;

`string erase(string s, int pos, int n)`
: returns the string formed by erasing the string of length `n` (if `n`=-1, to the end of the string `s`) at position `pos` in `s`;

`string substr(string s, int pos, int n=-1)`
: returns the substring of `s` starting at position `pos` and of length `n` (if `n`=-1, until the end of the string `s`);

`string reverse(string s)`
: return the string formed by reversing string `s`;

`string replace(string s, string from, string to)`
: returns a string with all occurrences of the string `from` in the string `s` changed to the string `to`;

`string replace(string s, string[][] table)`
: returns a string constructed by translating in string `s` all occurrences of the string `from` in an array `table` of string pairs `{from,to}` to the corresponding string `to`;

`string format(string s, int n)`
: returns a string containing `n` formatted according to the C-style format string `s`;

`string format(string s, real x)`
: returns a string containing `x` formatted according to the C-style format string `s` (see the documentation for the C-function `fprintf`), except that only one data field is allowed, trailing zeros are removed by default (unless `#` is specified) and TEX is used to typeset scientific notation;

`string time(string s)`

returns the current time formatted by the ANSI C routine strftime according to the string `s`. For example,

```
write(time("%a %b %d %H:%M:%S %Z %Y"));
```

outputs the time in the default format of the UNIX `date` command.

As in C/C++, complicated types may be abbreviated with `typedef` (see the example in Section 4.11 [Functions], page 28).

4.2 Guides and paths

`guide` an unresolved cubic spline (list of cubic-spline nodes and control points).

This is like a path except the computation of the cubic spline is deferred until drawing time (when it is resolved into a path); this allows two guides with free endpoint conditions to be joined together smoothly.

`path` a cubic spline resolved into a fixed path.

A path is specified as a list of pairs or paths interconnected with `--`, which denotes a straight line segment, or `..`, which denotes a cubic spline. Specifying a final node `cycle` creates a cyclic path that connects smoothly back to the initial node, as in this approximation (accurate to within 0.06%) of a unit circle:

```
guide unitcircle=E..N..W..S..cycle;
```

This example uses the standard compass directions `E=(1,0)`, `N=(0,1)`, `NE=unit(N+E)`, and `ENE=unit(E+NE)`, etc., which along with the directions `up`, `down`, `right`, and `left` are defined as pairs in the default `Asymptote` base file `plain.asy`. The routine `circle(pair c, real r)` in `plain.asy` constructs a circle of radius `r` centered on `c` by transforming unitcircle:

```
guide circle(pair c, real r)
{
  return shift(c)*scale(r)*unitcircle;
}
```

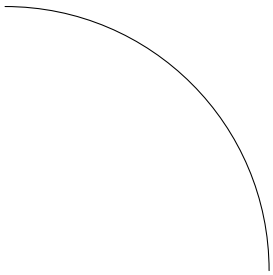
Each interior node of a cubic spline may be given a direction prefix or suffix `{dir}`: the direction of the pair `dir` specifies the direction of the incoming or outgoing tangent, respectively, to the curve at that node. Exterior nodes may be given direction specifiers only on their interior side. Cubic splines between a node `z`, with postcontrol point `Z`, and a node `w`, with precontrol point `W`, are computed as the Bezier curve

$$(1-t)^3 z + 3t(1-t)^2 Z + 3t^2(1-t)W + t^3 w \qquad 0 \le t \le 1.$$

This example draws an approximate quarter circle:

```
size(100,0);
```

```
draw((1,0){up}..{left}(0,1));
```



A general arc centered on `c` with radius `r` from `angle1` to `angle2` degrees (drawing counterclockwise if `angle2 >= angle1`) can be constructed with

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

Instead of specifying the tangent directions before and after a node, one can also specify the control points directly:

```
draw((0,0)..controls (0,100) and (100,100)..(100,0));
```

One can also change the spline tension from its default value of 1 to any real value greater than or equal to 0.75:

```
draw((100,0)..tension 2 ..(100,100)..(0,100));
draw((100,0)..tension 2 and 1 ..(100,100)..(0,100));
draw((100,0)..tension atleast 1 ..(100,100)..(0,100));
```

The `MetaPost ...` joiner, which requests, when possible, an inflection-free curve confined to a triangle defined by the endpoints and directions, is implemented in `Asymptote` as the convenient abbreviation `::` for `..tension atleast 1 ..` (the ellipsis `...` is used in `Asymptote` to indicate a variable number of arguments; see Section 4.11.3 [Rest arguments], page 30). For example, compare

```
draw((0,0){up}..(100,25){right}..(200,0){down});
```

with

```
draw((0,0){up}::(100,25){right}::(200,0){down});
```

The `---` joiner is an abbreviation for `..tension atleast infinity..` and the functionality of the `Metapost &` joiner is provided by `--`, as illustrated in the following example:

```
size(300,0);
pair[] z=new pair[10];
```

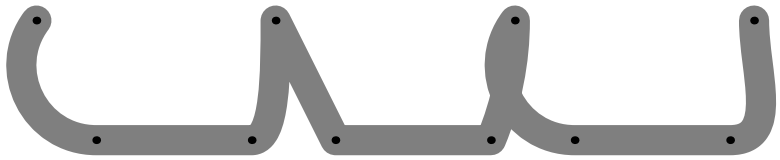
```
z[0]=(0,100); z[1]=(50,0); z[2]=(180,0);

for(int n=3; n <= 9; ++n)
  z[n]=z[n-3]+(200,0);

path p=z[0]..z[1]---z[2]::{up}z[3]
     --z[3]..z[4]--z[5]::{up}z[6]
     --z[6]::z[7]---z[8]..{up}z[9];

draw(p,grey+linewidth(4mm));

for(int n=0; n <= 9; ++n)
  dot(z[n]);
```



The curl parameter specifies the curvature at the endpoints of a path (0 means straight; the default value of 1 means approximately circular):

```
draw((100,0){curl 0}..(100,100)..{curl 0}(0,100));
```

The implicit initializer for paths and guides is `nullpath`, which is useful for building up a path within a loop. A direction specifier given to `nullpath` modifies the node on the other side: the paths

```
a..{up}nullpath..b;
c..nullpath{up}..d;
e..{up}nullpath{down}..f;
```

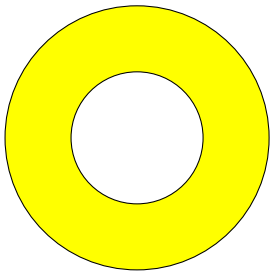
are respectively equivalent to

```
a..nullpath..{up}b;
c{up}..nullpath..d;
e{down}..nullpath..{up}f;
```

An `Asymptote` path, being connected, is equivalent to a `Postscript subpath`. The `^^` binary operator, which requests that the pen be moved (without drawing or affecting endpoint curvatures) from the final point of the left-hand path to the initial point of the right-hand path, may be used to group several `Asymptote` paths into a `path[]` array (equivalent to a `PostScript path`):

```
size(0,100);
path unitcircle=E..N..W..S..cycle;
path g=scale(2)*unitcircle;
```

```
filldraw(unitcircle^^g,evenodd+yellow,black);
```



The `PostScript` even-odd fill rule here specifies that only the region bounded between the two unit circles is filled. In this example, the same effect can be achieved by using the default zero winding number fill rule, if one is careful to alternate the orientation of the paths:

```
filldraw(unitcircle^^reverse(g),yellow,black);
```

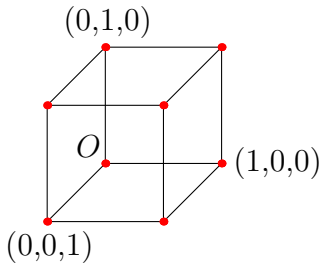
The `^^` operator is used by the `box3d` function in `three.asy` to construct a 2D projection of the edges of a 3D cube, without retracing steps:

```
size(0,100);
import math;
import three;

currentprojection=oblique;

draw(unitcube);
dot(unitcube,red);

label("$O$",(0,0,0),NW);
label("(1,0,0)",(1,0,0),E);
label("(0,1,0)",(0,1,0),N);
label("(0,0,1)",(0,0,1),S);
```



Here are some useful functions for paths:

`int length(path)`
: This is the number of (linear or cubic) segments in the path. If the path is cyclic, this is the same as the number of nodes in the path.

`int size(path)`
: This is the number of nodes in the path. If the path is cyclic, this is the same as the path length.

`pair point(path p, int n)`
: If `p` is cyclic, return the coordinates of node `n` mod `length(p)`. Otherwise, return the coordinates of node `n`, unless `n < 0` (in which case `point(0)` is returned) or `n > length(p)` (in which case `point(length(p))` is returned).

`pair point(path p, real x)`
: This returns the coordinates of the point between node `floor(x)` and `floor(x)+1` corresponding to the cubic spline parameter t=`x-floor(x)` (see [Bezier], page 10). If `x` lies outside the range [0,`length(p)`], it is first reduced modulo `length(p)` in the case where `p` is cyclic or else converted to the corresponding endpoint of `p`.

`pair dir(path, int n)`
: This returns the direction (as a pair) of the tangent to the path at node `n`. If the path contains only one point, (0,0) is returned.

`pair dir(path, real x)`
: This returns the direction of the tangent to the path at the point between node `floor(x)` and `floor(x)+1` corresponding to the cubic spline parameter t=`x-floor(x)` (see [Bezier], page 10). If the path contains only one point, (0,0) is returned.

`pair precontrol(path, int n)`
: This returns the precontrol point of node `n`.

`pair postcontrol(path, int n)`
: This returns the postcontrol point of node `n`.

`real arclength(path)`
: This returns the length of the piecewise linear or cubic curve that the path represents.

`real arctime(path, real L)`
: This returns the path "time", a real number between 0 and the length of the path in the sense of `point(path, real)`, at which the cumulative arclength (measured from the beginning of the path) equals `L`.

`real dirtime(path, pair z)`
: This returns the first "time", a real number between 0 and the length of the path in the sense of `point(path, real)`, at which the tangent to the path has direction `z`, or -1 if the path never achieves direction `z`.

`path reverse(path p)`
: returns a path running backwards along p.

`path subpath(path p, int n, int m)`
: returns the subpath running from node `n` to node `m`. If `n < m`, the direction of the subpath is reversed.

`path subpath(path p, real a, real b)`
: returns the subpath running from path time `a` to path time `b`, in the sense of `point(path, real)`. If `a < b`, the direction of the subpath is reversed.

`pair intersect(path p, path q)`
: If `p` and `q` have at least one intersection point, return a pair of times (s,t) representing the respective path times along `p` and `q`, in the sense of `point(path, real)`, for one such intersection point. If the paths do not intersect, return the pair (-1,-1).

`slice firstcut(path p, path q)`
: Return the portions of path `p` before and after the first intersection of `p` with path `q` as a structure `slice` (if no such intersection exists, the entire path is considered to be 'before' the intersection):

```
struct slice {
  public path before,after;
}
```

Note that `firstcut.after` plays the role of the `MetaPost cutbefore` command.

`slice lastcut(path p, path q)`
: Return the portions of path `p` before and after the last intersection of `p` with path `q` as a `slice` (if no such intersection exists, the entire path is considered to be 'after' the intersection).

Note that `lastcut.before` plays the role of the `MetaPost cutafter` command.

`pair max(path)`
: returns the pair(right,top) for the path bounding box.

`pair min(path)`
: returns the pair(left,bottom) for the path bounding box.

`bool cyclic(path)`
: returns `true` iff path is cyclic

`bool straight(path, int i)`
: returns `true` iff the segment between node `i` and node `i+1` is straight.

Finally, we point out an efficiency distinction in the use of guides and paths:

```
guide g;
for(int i=0; i < 10; ++i)
  g=g--(i,i);
path p=g;
```

runs in linear time, whereas

```
path p;
for(int i=0; i < 10; ++i)
  p=p--(i,i);
```

runs in quadratic time, as the entire path up to that point is copied at each step of the iteration.

4.3 Pens

In `Asymptote`, pens provide a context for the four basic drawing commands (see Chapter 5 [Drawing commands], page 62). They are used to specify the following drawing attributes: color, line type, line width, line cap, line join, fill rule, text alignment, font, font size, pattern, overwrite mode, and calligraphic transforms on the pen nib. The default pen used by the drawing routines is called `currentpen`. This provides the same functionality as the `MetaPost` command `pickup`.

Pens may be added together with the binary operator `+`. This will mix the colors of the two pens. All other non-default attributes of the right most pen will override those of the left most pen. Thus, one can obtain a yellow dashed pen by saying `dashed+red+green` or `red+green+dashed` or `red+dashed+green`. The binary operator `*` can be used to scale the color of a pen by a real number, until it saturates with one or more color components equal to 1.

- Colors are specified using one of the following colorspaces:

  `pen gray(real g)`
  : This produces a grayscale color, where the intensity `g` lies in the interval [0,1], with 0.0 denoting black and 1.0 denoting white.

  `pen rgb(real r, real g, real b)`
  : This produces an RGB color, where each of the red, green, and blue intensities `r`, `g`, `b`, lies in the interval [0,1].

  `pen cmyk(real c, real m, real y, real k)`
  : This produces a CMYK color, where each of the cyan, magenta, yellow, and black intensities `c`, `m`, `y`, `k`, lies in the interval [0,1].

  `pen invisible()`
  : This special pen writes in invisible ink, but adjusts the bounding box as if something had been drawn (like the `\phantom` command in TEX).

  The default color is `black`; this may be changed with the routine `defaultpen(pen)`. A number of named rgb colors are defined near the top of the default base file `plain.asy`:

  `black,gray,white,red,green,blue,yellow,magenta,cyan,brown,darkgreen,`
  `darkblue,orange,purple,chartreuse,fuchsia,salmon,lightblue,lavender,pink,`

  along with the primitive cmyk colors:

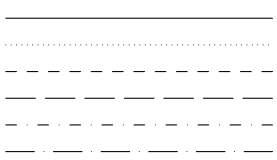
  `Cyan,Magenta,Yellow,Black.`

  The function `real[] colors(pen)` returns the color components of a pen.

- Line types are specified with the function `pen linetype(string s, bool scale=true)`, where `s` is a string of integer or real numbers separated by spaces. The first number specifies how far (if `scale` is `true`, in units of the pen linewidth;

otherwise in PostScript units) to draw with the pen on, the second number specifies how far to draw with the pen off, and so on (these spacings are automatically adjusted by Asymptote to fit the arclength of the path). Here are the predefined line types:

```
pen solid=linetype("");
pen dotted=linetype("0 4");
pen dashed=linetype("8 8");
pen longdashed=linetype("24 8");
pen dashdotted=linetype("8 8 0 8");
pen longdashdotted=linetype("24 8 0 8");
```



The default linetype is solid; this may be changed with defaultpen(pen).

- The pen line width is specified in PostScript units with pen linewidth(real). The default line width is 0.5 bp; this value may be changed with defaultpen(pen). Note that real (but not int) values are implicitly cast to pens of the specified line width:

```
pen double=1bp;
pen equivalent=1.0;
```

- A pen with a specific PostScript line cap is returned on calling linecap with an integer argument:

```
pen squarecap=linecap(0);
pen roundcap=linecap(1);
pen extendcap=linecap(2);
```

The default line cap, roundcap, may be changed with defaultpen(pen).

- A pen with a specific PostScript join style is returned on calling linejoin with an integer argument:

```
pen miterjoin=linejoin(0);
pen roundjoin=linejoin(1);
pen beveljoin=linejoin(2);
```

The default join style, roundjoin, may be changed with defaultpen(pen).

- A pen with a specific PostScript fill rule is returned on calling fillrule with an integer argument:

```
pen zerowinding=fillrule(0);
pen evenodd=fillrule(1);
```

The fill rule, which identifies the algorithm used to determine the insideness of a path or array of paths, only affects clip and fill commands. The default fill rule, zerowinding, may be changed with defaultpen(pen).

- A pen with a specific text alignment setting is returned on calling baseline with an integer argument:

```
pen nobasealign=baseline(0);
```

```
pen basealign=baseline(1);
```

The default setting, `nobasealign`, which may be changed with `defaultpen(pen)`, causes the label alignment routines to use the full label bounding box for alignment. In contrast, `basealign` requests that the TEX baseline be respected.

- The font size is specified in TEX points (1 pt = 1/72.27 inches) with the function `pen fontsize(real size, real baselineskip=1.2*size)`. The default font size, 12pt, may be changed with `defaultpen(pen)`. Nonstandard font sizes may require inserting

  ```
  texpreamble("\usepackage{type1cm}");
  ```

  at the beginning of the file.

- A pen using a specific `LaTeX NFSS` font is returned by calling the function `pen font(string encoding, string family, string series="m", string shape="n")`. The default setting, `font("OT1","cmr","m","n")`, corresponds to 12pt Computer Modern Roman; this may be changed with `defaultpen(pen)`.

  Alternatively, one may select a fixed-size `TeX` font (on which `fontsize` has no effect) like `"cmr12"` (12pt Computer Modern Roman) or `"pcrr"` (Courier) using the function `pen font(string name)`. An optional size argument can also be given to scale the font to the requested size: `pen font(string name, real size)`.

  A convenient interface to the following standard `PostScript` fonts is also provided:

  ```
  pen AvantGarde(string series="m", string shape="n");
  pen Bookman(string series="m", string shape="n");
  pen Courier(string series="m", string shape="n");
  pen Helvetica(string series="m", string shape="n");
  pen NewCenturySchoolBook(string series="m", string shape="n");
  pen Palatino(string series="m", string shape="n");
  pen TimesRoman(string series="m", string shape="n");
  pen ZapfChancery(string series="m", string shape="n");
  pen Symbol(string series="m", string shape="n");
  pen ZapfDingbats(string series="m", string shape="n");
  ```

- `PostScript` commands within a `picture` may be used to create a tiling pattern, identified by the string `name`, for `fill` and `draw` operations by adding it to the default `PostScript` preamble frame `patterns`, with optional left-bottom margin `lb` and right-top margin `rt`.

  ```
  void add(frame preamble=patterns, string name, picture pic, pair lb=0,
           pair rt=0)
  ```

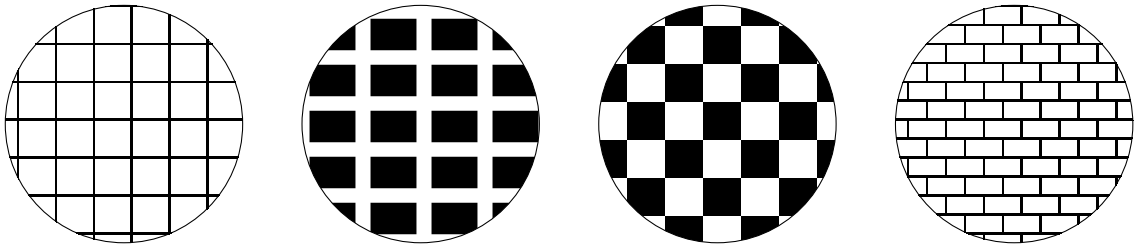
  To `fill` or `draw` using pattern `name`, use the pen `pattern("name")`. For example, rectangular tilings can be constructed using the routines `picture tile(real Hx=5mm, real Hy=0, pen p=currentpen, Filltype filltype=NoFill)`, `picture checker(real Hx=5mm, real Hy=0, pen p=currentpen)`, and `picture brick(real Hx=5mm, real Hy=0, pen p=currentpen)` defined in `patterns.asy`:

  ```
  size(0,90);
  import patterns;

  add("tile",tile());
  ```

```
add("filledtilewithmargin",tile(6mm,4mm,red,Fill),(1mm,1mm),(1mm,1mm));
add("checker",checker());
add("brick",brick());

real s=2.5;
filldraw(unitcircle,pattern("tile"));
filldraw(shift(s,0)*unitcircle,pattern("filledtilewithmargin"));
filldraw(shift(2s,0)*unitcircle,pattern("checker"));
filldraw(shift(3s,0)*unitcircle,pattern("brick"));
```

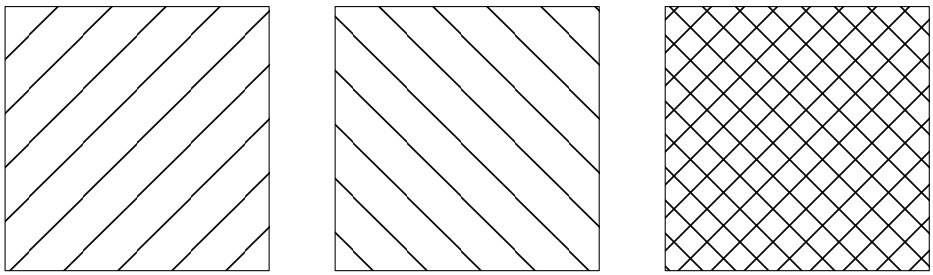


Hatch patterns can be generated with the routines `picture hatch(real H=5mm, pair dir=NE, pen p=currentpen)`, `picture crosshatch(real H=5mm, pen p=currentpen)`:

```
size(0,100);
import patterns;

add("hatch",hatch());
add("hatchback",hatch(NW));
add("crosshatch",crosshatch(3mm));

real s=1.25;
filldraw(unitsquare,pattern("hatch"));
filldraw(shift(s,0)*unitsquare,pattern("hatchback"));
filldraw(shift(2s,0)*unitsquare,pattern("crosshatch"));
```



You may need to turn off aliasing in your `PostScript` viewer for patterns to appear correctly. Custom patterns can easily be constructed, following the examples

in `pattern.asy`. The tiled pattern can even incorporate shading (see [gradient shading], page 64), as illustrated in this example (not included in the manual because not all printers support `PostScript` 3):

```
size(0,100);
import patterns;

real d=4mm;
picture tiling;
guide square=scale(d)*unitsquare;
fill(tiling,square,white,(0,0),black,(d,d));
fill(tiling,shift(d,d)*square,blue);
add("shadedtiling",tiling);

filldraw(unitcircle,pattern("shadedtiling"));
```

- One can prevent labels from overwriting one another by using the pen attribute `overwrite`, which takes a single argument:

  `Allow` Allow labels to overwrite one another. This is the default behaviour (unless overridden with `defaultpen(pen)`.

  `Suppress` Suppress, with a warning, each label that would overwrite another label.

  `SuppressQuiet`
  Suppress, without warning, each label that would overwrite another label.

  `Move` Move a label that would overwrite another out of the way and issue a warning. As this adjustment is during the final output phase (in `PostScript` coordinates) it could result in a larger figure than requested.

  `MoveQuiet`
  Move a label that would overwrite another out of the way, without warning. As this adjustment is during the final output phase (in `PostScript` coordinates) it could result in a larger figure than requested.

Calling the routine `defaultpen()` without arguments resets all pen default attributes to their initial values.

4.4 Transforms

`Asymptote` makes extensive use of affine transforms of the form $a + Mx$, where a is a pair and M is a $2x2$ matrix. Transforms can be applied to pairs, guides, paths, pens, transforms, frames, and pictures by multiplication (via the binary operator `*`) on the left (see [circle], page 10 for an example). Transforms can be composed with one another and inverted with the function `transform inverse(transform)`; they can also be raised to any integer power with the `^` operator.

The built-in transforms are:

`identity()`
the identity transform;

`shift(pair z)`
: translates by the pair `z`;

`xscale(real x)`
: scales by `x` in the x direction;

`yscale(real y)`
: scales by `y` in the y direction;

`scale(pair z)`
: equivalent to multiplication by `z`;

`slant(real s)`
: maps `(x,y) -> (x+s*y,y)`;

`rotate(real angle, pair z=(0,0))`
: rotates by `angle` in degrees about `z`;

`reflect(pair z, pair w)`
: reflects about the line `z--w`.

The implicit initializer for transforms is `identity()`.

4.5 Frames and pictures

`frame`
: Frames are canvases for drawing in `PostScript` coordinates. While working with frames directly is occasionally necessary for constructing deferred drawing routines, pictures are usually more convenient to work with. The implicit initializer for frames is `newframe`. The function `bool empty(frame f)` returns `true` only if the frame `f` is empty.

`picture`
: Pictures are high-level structures (see Section 4.7 [Structures], page 25) defined in `plain.asy` that provide canvases for drawing in user coordinates. The default picture is called `currentpicture`. A new picture can be created like this:

```
picture pic;
```

Anonymous pictures can be made by the expression `new picture`.

The `size` routine specifies the dimensions of the desired picture:

```
void size(picture pic=currentpicture,
          real xsize, real ysize=0, bool keepAspect=Aspect);
```

If `xsize` and `ysize` are both 0, user coordinates will be interpreted as `PostScript` coordinates. In this case, the transform mapping `pic` to the final output frame is `identity`.

If exactly one of `xsize` or `ysize` is 0, no size restriction is imposed in that direction; it will be scaled the same as the other direction.

If `keepAspect` is set to `Aspect` or `true`, the picture will be scaled with its aspect ratio preserved such that the final width is no more than `xsize` and the final height is no more than `ysize`.

If `keepAspect` is set to `IgnoreAspect` or `false`, the picture will be scaled in both directions so that the final width is `xsize` and the height is `ysize`.

A picture can be fit to a frame and converted into a `PostScript` image by calling the function `shipout`:

```
void shipout(string prefix=defaultfilename, picture pic,
             frame preamble=patterns,
             orientation orientation=Portrait,
             string format="", bool wait=NoWait);
void shipout(string prefix=defaultfilename,
             orientation orientation=Portrait,
             string format="", bool wait=NoWait);
```

A `shipout()` command is added implicitly at file exit if no previous `shipout` commands have been executed.

A picture `pic` can be explicitly fit to a frame by calling one of these member functions:

```
frame pic.fit();
frame pic.fit(pair dir);
frame pic.fit(real xsize, real ysize, bool keepAspect=true);
frame pic.fit(real xsize, real ysize, bool keepAspect=true,
              pair dir);
```

The first two forms use the `xsize`, `ysize`, and `keepAspect` values set by the `size` command (or the default values of `0`, `0`, and `false`, respectively). The optional `dir` argument specifies a direction to use for aligning the frame, in a manner completely analogous to the `align` argument of `label` (see Section 5.4 [label], page 65). Frame alignment is illustrated in the examples [errorbars], page 45 and [image], page 52.

```
frame f=pic.fit(xsize,ysize,true);
```

The default page orientation is `Portrait`. To output in landscape mode, simply replace the call to `shipout()` with:

```
shipout(Landscape);
```

To rotate in the other direction, replace `Landscape` with `Seascape`.

To draw a bounding box with a 0.25 cm margin around a picture, fit the picture into a frame using the `bbox` function (defined in `plain.asy`):

```
shipout(bbox(0.25cm));
```

A `picture` may be fit to a frame with the background color of pen `p` with the function `bbox(p,Fill)`.

To draw or fill a box or ellipse around a frame or label, use one of the routines (which for convenience also return the boundary as a guide):

```
guide box(frame f, real xmargin=0, real ymargin=infinity,
          pen pbox=currentpen, filltype filltype=NoFill);
guide ellipse(frame f, real xmargin=0, real ymargin=infinity,
              pen pbox=currentpen, filltype filltype=NoFill);
guide labelbox(frame f, real xmargin=0, real ymargin=infinity,
               string s, real angle=0, pair position,
               pair align=0, pen p=currentpen,
               pen pbox=currentpen, filltype filltype=NoFill);
guide labelellipse(frame f, real xmargin=0, real ymargin=infinity,
                   string s, real angle=0, pair position,
```

```
                    pair align=0, pen p=currentpen,
                    pen pbox=currentpen, filltype filltype=NoFill);
```

Sometimes it is useful to draw objects on separate pictures and add one picture to another using the `add` function:

```
void add(picture src, bool group=true);
void add(picture dest, picture src, bool group=true);
```

The first example adds `src` to `currentpicture`; the second one adds `src` to `dest`. The `group` option specifies whether or not the graphical user interface `xasy` should treat all of the elements of `src` as a single entity (see Chapter 9 [GUI], page 76).

There are also routines to add a picture or frame `src` specified in postscript coordinates to another picture about the user coordinate `origin`:

```
void add(pair origin, picture dest, picture src, bool group=true);
void add(pair origin, picture src, bool group=true);
void add(pair origin=(0,0), picture dest=currentpicture,
         frame src, bool group=true);
```

Alternatively, one can use `attach` to automatically increase the size of picture `dest` to accommodate adding a frame `src` about the user coordinate `origin`:

```
void attach(pair origin=(0,0), picture dest=currentpicture,
            frame src, bool group=true);
```

To erase the contents of a picture (but not the size specification), use the function

```
void erase(picture pic=currentpicture);
```

To save a snapshot of `currentpicture` and `currentpen` use the function `save()`.

To restore a snapshot of `currentpicture` and `currentpen` use the function `restore()`.

Many further examples of picture and frame operations are provided in the base file `plain.asy`.

It is possible to insert verbatim `PostScript` commands in a picture with the routine

```
void postscript(picture pic=currentpicture, string s);
```

Verbatim TEX commands can be inserted in the intermediate `LaTeX` output file with the function

```
void tex(picture pic=currentpicture, string s);
```

To issue a global TEX command (such as a TEX macro definition) in the TEX preamble (valid for the remainder of the top-level module) use:

```
void texpreamble(string s);
```

4.6 Files

`Asymptote` can read and write text files (including comma-separated value) files and portable XDR (External Data Representation) binary files.

An input file must first be opened with `input(string, bool check=true)`; reading is then done by assignment:

```
file fin=input("test.txt");
real a=fin;
```

If the optional boolean argument `check` is `false`, no check will be made that the file exists. If the file does not exist or is not readable, the function `bool error(file)` will return `true`. One can change the current working directory with the `string cd(string)` function, which returns the new working directory.

When reading pairs, the enclosing parenthesis are optional. Strings are also read by assignment, by reading characters up to but not including a newline. In addition, `Asymptote` provides the function `string getc(file)` to read the next character only, returning it as a string.

Writing is implemented using the function call `file output(string name,bool append=false)`, which opens a file `name` for output, appending to an existing file only if `append` is `true`:

```
real a=1, b=2, c=3;
file fout=output("test.txt");
write(fout,a);
```

There are two special files: `stdin`, which reads from the keyboard, and `stdout`, which writes to the terminal.

A final optional `endl` argument to `write` causes a newline to be written after writing the data. Alternatively, a final `tab` argument writes a tab instead. The file argument to `write` is also optional; when omitted the write is done to `stdout` and terminated by a newline. In this case, the write may include up to 3 data arguments, which will be output to `stdout`, separated by tabs. That is,

```
write();
write(a);
write(a,b);
write(a,b,c);
```

is equivalent to

```
write(stdout,endl);
write(stdout,a,endl);
write(stdout,a,tab); write(stdout,b,endl);
write(stdout,a,tab); write(stdout,b,tab); write(stdout,c,endl);
```

A file may also be opened with `xinput` or `xoutput` instead of `input` or `output`, in which case it will read or write double precision values written in Sun Microsystem's XDR (External Data Representation) portable binary format (available on all UNIX platforms). The function `file single(file)` sets the file to read single precision XDR values; calling `file single(file,false)` sets it back to read doubles again. The default initializer for file is `stdout`.

One can test a file for end-of-file with the boolean function `eof(file)`, end-of-line with `eol(file)`, and for I/O errors with `error(file)`. One can flush the output buffers with `flush(file)`, clear a previous I/O error with `clear(file)`, and close the file with `close(file)`. To set the number of digits of output precision, use `precision(file,int)`.

4.7 Structures

Users may also define their own data types as structures, along with user-defined operators, much as in C++. By default, structure members are read-only when referenced outside the structure, but may be optionally declared `public` (read-write) or `private` (read and write allowed only inside the structure). The virtual structure `this` refers to the enclosing structure. A default initializer for a structure `S` can be defined by creating a function `S operator init()`. This can be used to initialize each instance of `S` with `new S` (which creates a new anonymous instance of `S`).

```
struct S {
  public real a=1;
  real f(real a) {return a+this.a;}
}

S operator init() {return new S;}

S s;                            // Initializes s with S operator init();

write(s.f(2));                  // Outputs 3

S operator + (S s1, S s2)
{
  S result;
  result.a=s1.a+s2.a;
  return result;
}

write((s+s).f(0));              // Outputs 2
```

Any code at the top-level scope within the structure is executed on initialization. In the following example, the member function `void init(real x=0)` plays the role of a default constructor. Currently, any arguments to the constructor must be passed with an explicit call after initialization:

```
struct S {
  real x;
  void init(real x=0) {this.x=x;}
  init();
}

S operator init() {return new S;}

S a;
S b;
b.init(1);

write(a.x);             // Outputs 0
write(b.x);             // Outputs 1
```

For more complicated examples, see the structure `triangle` in `geometry.asy` or the structures `Legend` and `picture` in the default `Asymptote` base file `plain.asy`.

4.8 Arithmetic & logical operators

`Asymptote` uses the standard binary arithmetic operators. However, when one integer is divided by another, both arguments are converted to real values before dividing and a real quotient is returned (since this is usually what is intended). The function `int quotient(int x, int y)` returns the integer part of `x/y`, rounded toward 0. In all other cases both operands are promoted to the same type, which will also be the type of the result:

`+` addition

`-` subtraction

`*` multiplication

`/` division

`%` modulo; the result always has the same sign as the divisor.

`^` power; if the exponent (second argument) is an int, recursive multiplication is used; otherwise, logarithms and exponentials are used. `**` is a synonym for `^`.

The usual boolean operators are also defined:

`==` equals

`!=` not equals

`<` less than

`<=` less than or equals

`>=` greater than or equals

`>` greater than

`&&` and

`||` or

`^` xor

`!` not

`Asymptote` also supports the C-like conditional syntax:

```
bool positive=(pi >= 0) ? true : false;
```

4.9 Self & prefix operators

As in C, each of the arithmetic operators `+`, `-`, `*`, `/`, `%`, and `^` can be used as a self-operator. The prefix operators `++` (increment by one) and `--` (decrement by one) are also defined. For example,

```
int i=1;
i += 2;
int j=++i;
```

is equivalent to the code

```
int i=1;
i=i+2;
int j=i=i+1;
```

However, postfix operators like `i++` and `i--` are not defined (because of the inherent ambiguities that would arise with the `--` path-joining operator). In the rare instances where `i++` and `i--` are really needed, one can substitute the expressions `(++i-1)` and `(--i+1)`, respectively.

Keep in mind that these self- and prefix operators can cause side effects due to multiple evaluations, as in the following example (which writes "hi!" twice rather than just once):

```
struct T {
  public int x=0;
}

T sayhi() {
  write("hi!");
  return new T;
}

sayhi().x += 1;
```

4.10 Implicit scaling

If a numeric literal is in front of certain types of expressions, then the two are multiplied:

```
int x=2;
real y=2.0;
real cm=72/2.540005;

write(3x);
write(2.5x);
write(3y);
write(-1.602e-19 y);
write(0.5(x,y));
write(2x^2);
write(3x+2y);
write(3(x+2y));
write(3sin(x));
write(3(sin(x))^2);
write(10cm);
```

This produces the output

```
6
5
6
-3.204e-19
(1,1)
16
10
```

```
18
2.72789228047704
7.44139629388625
283.464008929116
```

4.11 Functions

`Asymptote` functions are treated as variables with a signature (non-function variables have null signatures). Variables with the same name are allowed, so long as they have distinct signatures.

Functions arguments are passed by value. To pass an argument by reference, simply enclose it in a structure (see Section 4.7 [Structures], page 25).

Here are some examples of `Asymptote` functions:

1. Two distinct variables:

```
int x, x();
x=5;
x=new int() {return 17;};
x=x();              // calls x() and puts the result, 17, in the scalar x
```

2. Traditional function definitions are allowed:

```
int sqr(int x)
{
  return x*x;
}
sqr=null;           // but the function is still just a variable.
```

3. Casting can be used to resolve ambiguities:

```
int a, a(), b, b(); // Valid: creates four variables.
a=b;                // Invalid: assignment is ambiguous.
a=(int) b;          // Valid: resolves ambiguity.
(int) (a=b);        // Valid: resolves ambiguity.
(int) a=b;          // Invalid: cast expressions cannot be L-values.

int c();
c=a;                // Valid: only one possible assignment.
```

4. Anonymous (so-called "high-order") functions are also allowed:

```
typedef int intop(int);
intop adder(int m)
{
  return new int(int n) {return m+n;};
}
intop addby7=adder(7);
write(addby7(1));   // Writes 8.
```

5. Anonymous functions can be used to redefine a function variable that has been declared (and implicitly initialized to the null function) but not yet explicitly defined:

```
void f(bool b);
```

```
void g(bool b) {
  if(b) f(b);
  else write(b);
}

f=new void(bool b) {
  write(b);
  g(false);
};

g(true);
```

`Asymptote` is the only language we know of that treats functions as variables, but allows overloading by distinguishing variables by their signatures.

Functions are allowed to call themselves recursively. As in C++, infinite nested recursion will generate a stack overflow (reported as a segmentation fault, unless the GNU library `libsigsegv` is installed at configuration time).

4.11.1 Default arguments

`Asymptote` supports a more flexible mechanism for default function arguments than C++: they may appear anywhere in the function prototype. Because certain data types are implicitly cast to more sophisticated types (see Section 4.13 [Casts], page 36) one can often avoid ambiguities by ordering function arguments from the simplest to the most complicated. For example, given

```
real f(int a=1, real b=0) {return a+b;}
```

then `f(1)` returns 1.0, but `f(1.0)` returns 2.0.

The value of a default argument is determined simply by evaluating the given `Asymptote` expression (effectively using a "cut and paste" operation to substitute the default value).

4.11.2 Named arguments

It is sometimes difficult to remember the order in which arguments appear in a function declaration. Named (keyword) arguments make calling functions with multiple arguments easier. Unlike in the C and C++ languages, an assignment in a function argument is interpreted as an assignment to a parameter of the same name in the function signature, *not within the local scope*. The command-line option `-d` may be used to check `Asymptote` code for cases where a named argument may be mistaken for a local assignment.

When matching arguments to signatures, first all of the keywords are matched, then the arguments without names are matched against the unmatched formals as usual. For example,

```
int f(int x, int y) {
  return 10x+y;
}
write(f(4,x=3));
```

output 34, as `x` is already matched when we try to match the unnamed argument `4`, so it gets matched to the next item, `y`.

For the rare occasions where it is desirable to assign a value to local variable within a function argument (generally *not* a good programming practice), simply enclose the assignment in parentheses. For example, given the definition of `f` in the previous example,

```
int x;
write(f(4,(x=3)));
```

is equivalent to the statements

```
int x;
x=3;
write(f(4,3));
```

and outputs 43.

As a technical detail, we point out that, since variables of the same name but different signatures are allowed in the same scope, the code

```
int f(int x, int x()) {
  return x+x();
}
int seven() {return 7;}
```

is legal in `Asymptote`, with `f(2,seven)` returning 9. A named argument matches the first unmatched formal of the same name, so `f(x=2,x=seven)` is an equivalent call, but `f(x=seven,2)` is not, as the first argument is matched to the first formal, and `int ()` cannot be implicitly cast to `int`. Default arguments do not affect which formal a named argument is matched to, so if `f` were defined as

```
int f(int x=3, int x()) {
  return x+x();
}
```

then `f(x=seven)` would be illegal, even though `f(seven)` obviously would be allowed.

4.11.3 Rest arguments

Rest arguments allow one to write functions that take a variable number of arguments:

```
// This function sums its arguments.
int sum(... int[] nums) {
  int total=0;
  for (int i=0; i < nums.length; ++i)
    total += nums[i];
  return total;
}

sum(1,2,3,4);                       // returns 10
sum();                              // returns 0

// This function subtracts subsequent arguments from the first.
int subtract(int start ... int[] subs) {
  for (int i=0; i < subs.length; ++i)
    start -= subs[i];
  return start;
```

```
}

subtract(10,1,2);                    // returns 7
subtract(10);                        // returns 10
subtract();                          // illegal
```

Putting an argument into a rest array is called *packing*. One can give an explicit list of arguments for the rest argument, so `subtract` could alternatively be implemented as

```
int subtract(int start ... int[] subs) {
  return start - sum(... subs);
}
```

One can even combine normal arguments with rest arguments:

```
sum(1,2,3 ... new int[] {4,5,6});   // returns 21
```

This builds a new six-element array that is passed to `sum` as `nums`. The opposite operation, *unpacking*, is not allowed:

```
subtract(... new int[] {10, 1, 2});
```

is illegal, as the start formal is not matched.

If no arguments are packed, then a zero-length array (as opposed to `null`) is bound to the rest parameter. Note that default arguments are ignored for rest formals and the rest argument is not bound to a keyword.

The overloading resolution in `Asymptote` is similar to the function matching rules used in C++. Every argument match is given a score. Exact matches score better than matches with casting, and matches with formals score better than packing an argument into the rest array. A candidate is maximal if all of the arguments score as well in it as with any other candidate. If there is one unique maximal candidate, it is chosen; otherwise, there is an ambiguity error.

```
int f(path g);
int f(guide g);
f((0,0)--(100,100)); // matches the second; the argument is a guide

int g(int x, real y);
int g(real x, int x);

g(3,4); // ambiguous; the first candidate is better for the first argument,
        // but the second candidate is better for the second argument

int h(... int[] rest);
int h(real x ... int[] rest);

h(1,2); // the second definition matches, even though there is a cast,
        // because casting is preferred over packing

int i(int x ... int[] rest);
int i(real x, real y ... int[] rest);

i(3,4); // ambiguous; the first candidate is better for the first argument,
```

```
// but the second candidate is better for the second one
```

4.11.4 Mathematical functions

`Asymptote` has built-in versions of the standard `libm` mathematical real(real) functions `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log`, `pow10`, `log10`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`, `sqrt`, `cbrt`, `fabs`, as well as the identity function `identity`. `Asymptote` also defines the Bessel function of order `n` of the first kind `J(int n, real)` and the second kind `Y(int n, real)`. The standard real(real, real) functions `atan2`, `hypot`, `fmod`, `remainder` are also included.

The functions `ceil`, `floor`, and `round` differ from their usual definitions in that they all return an `int` value rather than a real (since that is normally what one wants). We also define a function `sgn`, which returns the sign of its real argument as an integer (-1, 0, or 1).

There is an `abs(int)` function, as well as an `abs(real)` function (equivalent to `fabs(real)`) and an `abs(pair)` function (equivalent to `length(pair)`).

Random numbers can be seeded with `srand(int)` and generated with the `int rand()` function, which returns a random integer between 0 and the integer `randMax()`. A Gaussian random number generator `Gaussrand` and a collection of statistics routines, including `histogram`, are provided in the base file `stats.asy`.

4.12 Arrays

Appending `[]` to any type (either built-in or user defined) produces an array (technically a vector) indexed by integers in the interval [0,`length`-1], where `length` represents the array length. For convenience, integer indices in [-`length`,-1] are mapped to [0,`length`-1]. Reading an array element with an index outside of these bounds generates an error. Assigning an array element with an index less than -`length` generates an error; however, if the index is non-negative, the array is resized, as required, to accommodate the new element. One can also index an array `A` with an integer array `B`, to obtain the array formed by indexing array `A` with successive elements of array `B`.

The declaration

```
real[] A;
```

initializes `A` to be an empty (zero-length) array. Empty arrays should be distinguished from null arrays. If we say

```
real[] A=null;
```

then `A` cannot be dereferenced at all (null arrays have no length and cannot be read from or assigned to).

Arrays can be explicitly initialized like this:

```
real[] A={0,1,2};
```

Array assignment in `Asymptote` does a shallow copy: only the pointer is copied (if one copy if modified, the other will be too). The `copy` function listed below provides a deep copy of an array.

Every array `A` has three virtual members, `length`, `push`, and `pop`. `A.length` evaluates to the length of the array `A`, `A.push()` is a function that pushes its argument onto the end of the array, and `A.pop()` is a function that pops and returns the last element of the array.

Like all functions in Asymptote, push and pop can be "pulled off" of the array and used on their own:

```
int[] A={1,2,3};
void f(int)=A.push;
f(7);               // A now contains {1,2,3,7}.
int g()=A.pop;
write(g());         // Outputs 7.
```

The [] suffix can also appear after the variable name; this is sometimes convenient for declaring a list of variables and arrays of the same type:

```
real a,A[];
```

This declares a to be real and implicitly declares A to be of type real[]. But beware that this alternative syntax does not construct certain internal type-dependent functions that take real[] as an argument: alias, copy, concat, sequence, map, and transpose for type real[] won't be defined until the type real[] is used explicitly somewhere.

In the following list of built-in array functions, T represents a generic type.

`new T[]` returns a new empty array of type T[];

`new T[] {list}`
: returns a new array of type T[] initialized with list (a comma delimited list of elements).

`new T[n]` returns a new array of n elements of type T[]. Unless they are arrays themselves, these n array elements are not initialized.

`int[] sequence(int n)`
: if n >= 1 returns the array {0,1,...,n-1} (otherwise returns a null array);

`int[] sequence(int n, int m)`
: if m >= n returns an array {n,n+1,...,m} (otherwise returns a null array);

`T[] sequence(T f(int),n)`
: if n >= 1 returns the sequence {f_i :i=0,1,...n-1} given a function T f(int) and integer int n (otherwise returns a null array);

`int[] reverse(int n)`
: if n >= 1 returns the array {n-1,n-2,...,0} (otherwise returns a null array);

`int find(bool[], int n=1)`
: returns the index of the nth true value or -1 if not found. If n is negative, search backwards from the end of the array for the -nth value;

`int search(T[], T key)`
: For ordered types T, searches a sorted ordered array of n elements to find an interval containing key, returning -1 if key is less than the first element, n-1 if key is greater than or equal to the last element, and otherwise the index corresponding to the left-hand (smaller) endpoint.

`T[] copy(T[] A)`
: returns a deep copy of the array A;

`T[] concat(T[] A, T[] B)`
: returns a new array formed by concatenating arrays A and B;

`bool alias(T[] A, T[] B)`
: returns `true` if the arrays `A` and `B` are identical;

`T[] sort(T[] A)`
: For ordered types `T`, returns a copy of `A` sorted in ascending order;

`T[][] sort(T[][] A)`
: For ordered types `T`, returns a copy of `A` with the rows sorted by the first column, breaking ties with successively higher columns. For example:

```
string[][] a={{"bob","9"},{"alice","5"},{"pete","7"},
              {"alice","4"}};
write("Row sort (by column 0, using column 1 to break ties):");
write(stdout,sort(a));
```

produces

```
alice   4
alice   5
bob     9
pete    7
```

`T[][] transpose(T[][] A)`
: returns the transpose of `A`.

`T sum(T[] A)`
: For arithmetic types `T`, returns the sum of `A`.

`T min(T[] A)`
: For ordered types `T`, returns the minimum element of `A`.

`T max(T[] A)`
: For ordered types `T`, returns the maximum element of `A`.

`map(f(T), T[] A)`
: returns the array obtained by applying the function `f` to each element of the array `A`.

`T[] min(T[] A, T[] B)`
: For ordered types `T`, and arrays `A` and `B` of the same length, returns an array composed of the minimum of the corresponding elements of `A` and `B`.

`T[] max(T[] A, T[] B)`
: For ordered types `T`, and arrays `A` and `B` of the same length, returns an array composed of the maximum of the corresponding elements of `A` and `B`.

`pair[] fft(pair[] A, int sign)`
: returns the Fast Fourier Transform of `A` (if the optional `FFTW` package is installed), using the given `sign`. Here is a simple example:

```
int n=4;
pair[] f=sequence(n);
write(f);
pair[] g=fft(f,-1);
write();
write(g);
```

```
f=fft(g,1);
write();
write(f/n);
```

`Asymptote` includes a full set of vectorized array instructions for arithmetic (including self) and logical operations. These element-by-element instructions are implemented in C++ code for speed. Given

```
real[] a={1,2};
real[] b={3,2};
```

then `a == b` and `a >= 2` both evaluate to the vector `{false, true}`. To test whether all components of `a` and `b` agree, use the boolean function `all(a == b)`. One can also use conditionals like `(a >= 2) ? a : b`, which returns the array `{3,2}`, or `write((a >= 2) ? a : null`, which returns the array `{2}`.

All of the standard built-in `libm` functions of signature `real(real)` also take a real array as an argument, effectively like an implicit call to `map`.

As with other built-in types, arrays of the basic data types can be read in by assignment. In this example, the code

```
file fin=input("test.txt");
real[] A=fin;
```

reads real values into `A` until the end of file is reached (or an I/O error occurs). If line mode is set with `line(file)`, then reading will stop once the end of the line is reached instead (line mode may be cleared with `line(file,false)`):

```
file fin=input("test.txt");
real[] A=line(fin);
```

Another useful mode is comma-separated-value mode, set with `csv(file)` and cleared with `csv(file,false)`, which skips over any comma delimiters:

```
file fin=input("test.txt");
real[] A=csv(fin);
```

To restrict the number of values read, use the `dimension(file,int)` function:

```
file fin=input("test.txt");
real[] A=dimension(fin,10);
```

This reads 10 values into A, unless end-of-file (or end-of-line in line mode) occurs first. Attempting to read beyond the end of the file will produce a runtime error message. Specifying a value of 0 for the integer limit is equivalent to the previous example of reading until end-of-file (or end-of-line in line mode) is encountered.

Two- and three-dimensional arrays of the basic data types can be read in like this:

```
file fin=input("test.txt");
real[][] A=dimension(fin,2,3);
real[][][] B=dimension(fin,2,3,4);
```

Again, an integer limit of zero means no restriction.

Sometimes the array dimensions are stored with the data as integer fields at the beginning of an array. Such arrays can be read in with the functions `read1`, `read2`, and `read3`, respectively:

```
file fin=input("test.txt");
real[] A=read1(fin);
real[][] B=read2(fin);
real[][][] C=read3(fin);
```

One, two, and three-dimensional arrays of the basic data types can be output with the functions `write(file,T[])`, `write(file,T[][])`, `write(file,T[][][])`, respectively. The command `scroll(int n)` is useful for pausing the output after every n output lines (press `Enter` to continue).

4.13 Casts

`Asymptote` implicitly casts `int` to `real`, `int` to `pair`, `real` to `pair`, `pair` to `path`, `pair` to `guide`, `path` to `guide`, `guide` to `path`, and `real` to `pen`. Implicit casts are also automatically attempted when trying to match function calls with possible function signatures. Implicit casting can be inhibited by declaring individual arguments `explicit` in the function signature, say to avoid an ambiguous function call in the following example, which outputs 0:

```
int f(pair a) {return 0;}
int f(explicit real x) {return 1;}

write(f(0));
```

Other conversions, say `real` to `int` or `real` to `string`, require an explicit cast:

```
int i=(int) 2.5;
string s=(string) 2.5;

real[] a={2.5,-3.5};
int[] b=(int []) a;
write(stdout,b);     // Outputs 2,-3
```

Casting to user-defined types is also possible using `operator cast`:

```
struct rpair {
  public real radius;
  public real angle;
}

rpair operator init() {return new rpair;}

pair operator cast(rpair x) {
  return (x.radius*cos(x.angle),x.radius*sin(x.angle));
}

rpair x;
x.radius=1;
x.angle=pi/6;

write(x);            // Outputs (0.866025403784439,0.5)
```

One must use care when defining new cast operators. Suppose that in some code one wants all integers to represent multiples of 100. To convert them to reals, one would first want to multiply them by 100. However, the straightforward implementation

```
real operator cast(int x) {return x*100;}
```

is equivalent to an infinite recursion, since the result `x*100` needs itself to be cast from an integer to a real. Instead, we want to use the standard conversion of int to real:

```
real convert(int x) {return x*100;}
real operator cast(int x)=convert;
```

Explicit casts are implemented similarly, with `operator ecast`.

4.14 Import

One can import other `Asymptote` files with the `import` command:

```
import graph;
```

Such `Asymptote` files are often called modules.

`Asymptote` searches the locations listed in [search paths], page 2 for the matching file. If the file name contains nonalphanumeric characters, enclose it with quotation marks:

```
import "/usr/local/share/asymptote/graph.asy";
```

A variable `var` from a module named `module` can be distinguished from a local variable of the same name by prefixing it with the module name and a period: `module.var`.

Imports within an imported module are not seen outside of that module: even though `plain` imports `simplex`, the `simplex` module will not be accessible to the file-level module, unless it is explicitly imported. An error will be issued if one attempts to import a module multiple times.

Modules are allowed to import themselves recursively. Infinite nested importing will generate a stack overflow (reported as a segmentation fault, unless the GNU library `libsigsegv` is installed at configuration time).

Currently, the module name to be imported must be known at compile time. However, you can execute an `Asymptote` file determined at runtime in an independent process with the function

```
void execute(string file);
```

You can also evaluate an asymptote expression (without any return value, however) contained in the string `s` with:

```
void eval(string s);
```

`Asymptote` currently ships with the following base modules:

4.14.1 `plain`

This is the default `Asymptote` base file, which defines key parts of the drawing language (such as the `picture` structure).

By default, the top-level module automatically imports `plain`; use the `-noplain` command-line option to disable this feature. In any case, no error message will be issued if one tries to explicitly `import plain` (if other imported packages depend implicitly on the presence of `plain`, it may be necessary to allocate it in a higher enclosing scope by declaring the import `static`; see Section 4.15 [Static], page 59).

4.14.2 simplex

This package solves the two-variable linear programming problem using the simplex method. It is used by `plain` for automatic sizing of pictures.

4.14.3 graph

This package implements two-dimensional linear and logarithmic graphs, including automatic scale and tick selection (with the ability to override manually). A graph is a `guide` (that can be drawn with the draw command, with an optional legend) constructed with one of the following routines:

- ```
  guide graph(picture pic=currentpicture, real f(real), real a, real b,
              int n=ngraph, interpolate interpolatetype=Straight);
  ```
  Returns a graph using the scaling information for picture `pic` (see [automatic scaling], page 47) of the function `f` on the interval [`a`,`b`], sampling at `n` evenly spaced points, with one of these interpolation types:
  - `Straight` (linear interpolation)
  - `Spline` (piecewise Bezier cubic spline interpolation)
- ```
  guide graph(picture pic=currentpicture, real x(real), real y(real),
              real a, real b, int n=ngraph,
              interpolate interpolatetype=Straight);
  ```
  Returns a graph using the scaling information for picture `pic` of the parametrized function (`x`(t),`y`(t)) for t in [`a`,`b`], sampling at `n` evenly spaced points, with the given interpolation type.
- ```
  guide graph(picture pic=currentpicture, pair z(real), real a, real b,
              int n=ngraph, interpolate interpolatetype=Straight);
  ```
  Returns a graph using the scaling information for picture `pic` of the parametrized function `z`(t) for t in [`a`,`b`], sampling at `n` evenly spaced points, with the given interpolation type.
- ```
  guide graph(picture pic=currentpicture, pair[] z, bool[] cond={},
              interpolate interpolatetype=Straight);
  ```
  Returns a graph using the scaling information for picture `pic` of those elements of the array `z` for which the corresponding elements of the boolean array `cond` are `true`, with the given interpolation type.
- ```
  guide graph(picture pic=currentpicture, real[] x, real[] y,
              bool[] cond={}, interpolate interpolatetype=Straight);
  ```
  Returns a graph using the scaling information for picture `pic` of those elements of the arrays (`x`,`y`) for which the corresponding elements of the boolean array `cond` are `true`, with the given interpolation type.
-
  ```

```
guide graph(real f(real), real a, real b, int n=ngraph, real T(real),
          interpolate interpolatetype=Straight);
```

Returns a graph using the scaling information for picture `pic` of the function `f` on the interval [`T(a)`,`T(b)`], sampling at `n` points evenly spaced in [`a`,`b`], with the given interpolation type.

- ```
  guide polargraph(real f(real), real a, real b, int n=ngraph,
                   interpolate interpolatetype=Straight);
  ```

  Returns a polar-coordinate graph using the scaling information for picture `pic` of the function `f` on the interval [`a`,`b`], sampling at `n` evenly spaced points, with the given interpolation type.

An axis can be drawn on a picture with one of the following commands:

- ```
  void xaxis(picture pic=currentpicture,
             real xmin=-infinity, real xmax=infinity, string s="",
             real position=infinity, real angle=0, pair align=0,
             pair shift=0, pair side=0,
             pen plabel=currentpen, pen p=plabel, pen pticklabel=plabel,
             axis axis=YZero, ticks ticks=NoTicks, arrowbar arrow=None,
             bool put=Below);
  ```

  Draw an x axis on picture `pic` from x=`xmin` to x=`xmax` using pen `p`, optionally labelling it with string `s` at relative location `position` (a real number from [0,1]) using angle `angle`, alignment `align` (user coordinates), true shift `shift` (postscript coordinates) on side `side` of the axis, and pens `plabel` and `pticklabel`. An infinite value of `xmin` or `xmax` specifies that the corresponding axis limit will be automatically determined from the picture limits. The axis placement is determined by one of the following `axis` types:

  `YZero(bool extend=true)`
  : Request an x axis at y=0 extending to the full dimensions of the picture, unless `extend`=false.

  `YEquals(real Y, bool extend=true)`
  : Request an x axis at y=`Y` extending to the full dimensions of the picture, unless `extend`=false.

  `Bottom(bool extend=false)`
  : Request a bottom axis.

  `Top(bool extend=false)`
  : Request a top axis.

  `BottomTop(bool extend=false)`
  : Request a bottom and top axis.

The optional `arrow` argument takes the same values as in the `draw` command (see [arrows], page 62). If `put=Below` and the `extend` flag for `axis` is `false`, the axis is drawn before any existing objects in the current layer.

The default tick option is `NoTicks`. The option `LeftTicks` (`RightTicks`) can be used to draw ticks on the left (right) of the path, relative to the direction in which the path is drawn. These tick routines accept a number of optional arguments:

```
ticks LeftTicks(bool begin=true, int N=0, int n=0, real Step=0,
                real step=0, real Size=Ticksize, real size=ticksize,
                string format, bool end=true);
```

If any of these parameters are omitted (or 0), reasonable defaults will be chosen:

`begin` requests a big tick at the beginning of the axis;

`N` when automatic scaling is enabled (the default; see [automatic scaling], page 47), divide the arclength evenly into this many intervals, separated by big ticks;

`n` divide each arclength interval into this many subintervals, separated by small ticks;

`Step` represents the spacing (in user arclength coordinates) between big ticks (if `N=0`);

`step` represents the spacing (in user arclength coordinates) between small ticks (if `n=0`);

`format` is a string used for formatting the tick values (see [format], page 9), normally beginning and ending with `$`) to enable the `LaTeX` math mode fonts;

`end` requests a big tick at the end of the axis.

It is also possible to specify custom tick locations with `LeftTicks` and `RightTicks` by passing explicit real arrays `Ticks` and (optionally) `ticks` containing the locations of the big and small ticks, respectively:

```
ticks LeftTicks(real[] Ticks, real[] ticks=new real[],
                real Size=Ticksize, real size=ticksize, string format);
```

-

```
void yaxis(picture pic=currentpicture,
           real ymin=-infinity, real ymax=infinity, string s="",
           real position=infinity, real angle=infinity, pair align=0,
           pair shift=0, pair side=0,
           pen plabel=currentpen, pen p=plabel, pen pticklabel=plabel,
           axis axis=XZero, ticks ticks=NoTicks, arrowbar arrow=None,
           bool put=Below);
```

Draw a y axis on picture `pic` from y=`ymin` to y=`ymax` using pen `p`, optionally labelling it with string `s` at relative location `position` (a real number from [0,1]) using angle `angle`, alignment `align`, true shift `shift` on side `side` of the axis, and pens `plabel` and `pticklabel`. The tick type is specified by `ticks` and the axis placement is determined by one of the following `axis` types:

`XZero(bool extend=true)`
: Request a y axis at x=0 extending to the full dimensions of the picture, unless `extend`=false.

`XEquals(real X, bool extend=true)`
: Request a y axis at x=`X` extending to the full dimensions of the picture, unless `extend`=false.

`Left(bool extend=false)`
: Request a left axis.

`Right(bool extend=false)`
: Request a right axis.

`LeftRight(bool extend=false)`
: Request a left and right axis.

The optional `arrow` argument takes the same values as in the `draw` command (see [arrows], page 62). If `put=Below` and the `extend` flag for `axis` is `false`, the axis is drawn before any existing objects in the current layer.

- For convenience, the functions

```
void xequals(picture pic=currentpicture, real x, bool extend=false,
             real ymin=-infinity, real ymax=infinity, string s="",
             real position=infinity, real angle=infinity, pair align=0,
             pair shift=0, pair side=0,
             pen plabel=currentpen, pen p=plabel, pen pticklabel=plabel,
             ticks ticks=NoTicks, bool put=Above, arrowbar arrow=None);
```

and

```
void yequals(picture pic=currentpicture, real y, bool extend=false,
             real xmin=-infinity, real xmax=infinity, string s="",
             real position=infinity, real angle=0, pair align=0,
             pair shift=0, pair side=0,
             pen plabel=currentpen, pen p=plabel, pen pticklabel=plabel,
             ticks ticks=NoTicks, bool put=Above, arrowbar arrow=None);
```

can be respectively used to call `yaxis` and `xaxis` with the appropriate axis types `XEquals(x,extend)` and `YEquals(y,extend)`. This is the recommended way of drawing vertical or horizontal lines and axes at arbitrary locations.

-

```
void axis(picture pic=currentpicture, guide g,
          real tickmin=-infinity, real tickmax=infinity,
          string s="", real position=1, real angle=0, pair align=S,
          pair shift=0, pair side=right, pen plabel=currentpen,
          pen p=plabel, pen pticklabel=plabel, ticks ticks=NoTicks,
          arrowbar arrow=None, int[] divisor=new int[],
          bool logarithmic=false, scaleT scale=Linear, part part,
          bool put=Above, bool opposite=false);
```

This routine can be used to draw a general axis on picture `pic` using pen `p`, based on an arbitrary path `g`, with ticks placed along its arclength (either evenly spaced or at custom locations). One may specify optional tick minimum and maximum values `tickmin` and `tickmax`, a tick label pen `pticklabel`, and optionally label the axis with string `s` at relative location `position` (an `arctime` value from [0,1]), angle `angle`, alignment `align`, true shift `shift` on side `side` of the axis, and pen `plabel`. The tick type is specified by `ticks` and the integer array `divisor` specifies what tick divisors to try in the attempt to produce uncrowded tick labels. A `true` value for the flag `opposite` identifies an unlabelled secondary axis (typically drawn opposite to a primary axis). The axis is drawn on top of any existing objects in the current layer only if `put` is `Above`.

- These routines are useful for manually putting ticks and labels on axes:

```
void xtick(picture pic=currentpicture, string s="", pair z,
           pair dir=N, pair align=0, pair shift=infinity,
           real size=Ticksize, pen p=currentpen);
void ytick(picture pic=currentpicture, string s="", explicit pair z,
           pair dir=E, pair align=0, pair shift=infinity,
           real size=Ticksize, pen p=currentpen);
void ytick(picture pic=currentpicture, string s="", real y,
           pair dir=E, pair align=0, pair shift=infinity,
           real size=Ticksize, pen p=currentpen);
void tick(picture pic=currentpicture, pair z, pair align,
          real size=Ticksize, pen p=currentpen);
void labelx(picture pic=currentpicture, string s="", pair z,
            pair align=S, pair shift=infinity,
            string format=defaultformat, pen p=currentpen);
void labely(picture pic=currentpicture, string s="", explicit pair z,
            pair align=W, pair shift=infinity,
            string format=defaultformat, pen p=currentpen);
void labely(picture pic=currentpicture, string s="", real y,
            pair align=W, pair shift=infinity,
            string format=defaultformat, pen p=currentpen);
void labelxtick(picture pic=currentpicture, pair z,
                pair align=S, pair shift=infinity,
                string format=defaultformat, real size=Ticksize,
                pen p=currentpen);
void labelytick(picture pic=currentpicture, explicit pair z,
                pair align=W, pair shift=infinity,
                string format=defaultformat, real size=Ticksize,
                pen p=currentpen);
void labelytick(picture pic=currentpicture, real y,
                pair align=W, pair shift=infinity,
                string format=defaultformat, real size=Ticksize,
                pen p=currentpen);
```

Here are some simple examples of two-dimensional graphs:

1. This example draws a textbook-style graph of $y = \exp(x)$, with the y axis starting at

$y = 0$:

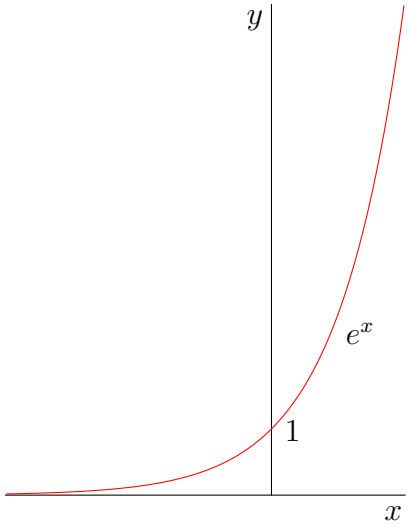
```
import graph;
size(150,0);

real f(real x) {return exp(x);}
pair F(real x) {return (x,f(x));}

xaxis("$x$");
yaxis(0,"$y$");

draw(graph(f,-4,2,Spline),red);

labely(1,E);
label("$e^x$",F(1),SE);
```



2. The next example draws a scientific-style graph with a legend. The position of the legend can either be adjusted explicitly or by using the graphical user interface `xasy` (see Chapter 9 [GUI], page 76).

```
import graph;

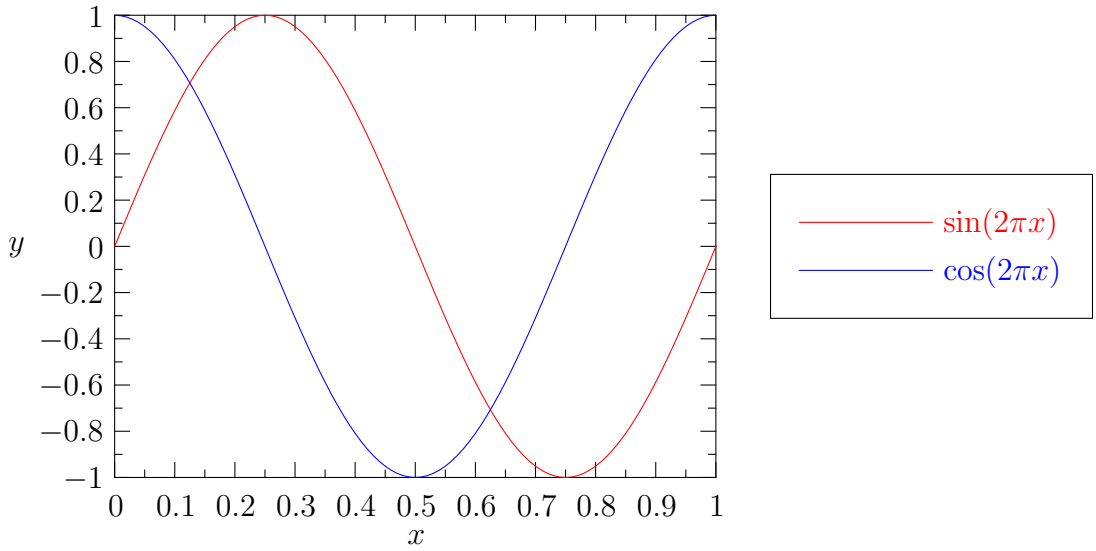
size(400,200,IgnoreAspect);

real Sin(real t) {return sin(2pi*t);}
real Cos(real t) {return cos(2pi*t);}

draw(graph(Sin,0,1),red,"$\sin(2\pi x)$");
draw(graph(Cos,0,1),blue,"$\cos(2\pi x)$");
```

```
xaxis("$x$",BottomTop,LeftTicks);
yaxis("$y$",LeftRight,RightTicks);

add(point(E),legend(20E));
```



To specify a fixed size for the graph proper, use `attach`:

```
import graph;

size(250,200,IgnoreAspect);

real Sin(real t) {return sin(2pi*t);}
real Cos(real t) {return cos(2pi*t);}

draw(graph(Sin,0,1),red,"$\sin(2\pi x)$");
draw(graph(Cos,0,1),blue,"$\cos(2\pi x)$");

xaxis("$x$",BottomTop,LeftTicks);
yaxis("$y$",LeftRight,RightTicks);

attach(point(E),legend(20E));
```

3. This example draws a graph of one array versus another (both of the same size) using custom tick locations and a smaller font size for the tick labels on the y axis.

```
import graph;

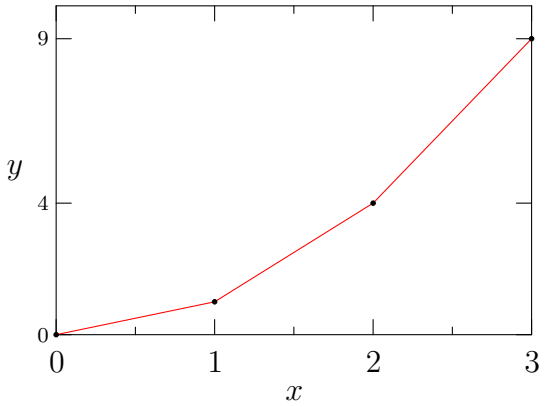
size(200,150,IgnoreAspect);

real[] x={0,1,2,3};
```

```
real[] y=x^2;

draw(graph(x,y),red,MarkFill[0]);

xaxis("$x$",BottomTop,LeftTicks);
yaxis("$y$",pticklabel=fontsize(8),LeftRight,
      RightTicks(new real[]{0,4,9}));
```



4. The next example draws two graphs of an array of coordinate pairs, using frame alignment and data markers. In the left-hand graph, the marker is constructed with the function `frame marker(path g, pen p=currentpen, filltype filltype=NoFill)` using the path `unitcircle`. In the right-hand graph, the unit n-sided regular polygon `polygon(int n)` and the unit n-point cross `cross(int n)` are used to build a custom marker frame. This example also illustrates the `errorbar` routines:

```
void errorbars(picture pic=currentpicture, pair[] z, pair[] dp,
               pair[] dm={}, bool[] cond={}, pen p=currentpen,
               real size=0);

void errorbars(picture pic=currentpicture, real[] x, real[] y,
               real[] dpx, real[] dpy, real[] dmx={}, real[] dmy={},
               bool[] cond={}, pen p=currentpen, real size=0);
```

Here, the positive and negative extents of the error are given by the absolute values of the elements of the pair array `dp` and the optional pair array `dm`. If `dm` is not specified, the positive and negative extents of the error are assumed to be equal.

```
import graph;

picture pic;
real xsize=200, ysize=140;
size(pic,xsize,ysize,IgnoreAspect);

pair[] f={(5,5),(50,20),(90,90)};
pair[] df={(0,0),(5,7),(0,5)};
```

```
errorbars(pic,f,df,red);
draw(pic,graph(pic,f),marker(scale(0.8mm)*unitcircle,blue,Fill),Below);

xaxis(pic,"$x$",BottomTop,LeftTicks);
yaxis(pic,"$y$",LeftRight,RightTicks);

picture pic2;
size(pic2,xsize,ysize,IgnoreAspect);

frame mark;
filldraw(mark,scale(0.8mm)*polygon(6),green);
draw(mark,scale(0.8mm)*cross(6),blue);

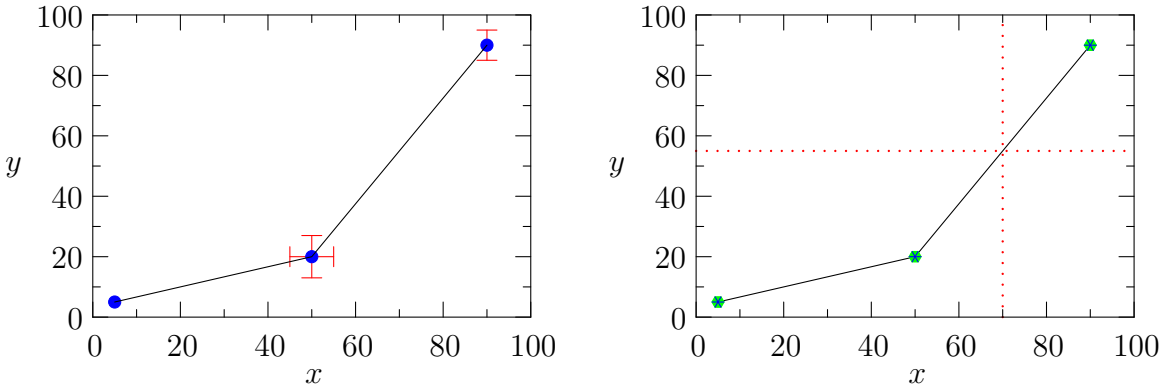
draw(pic2,graph(f),mark);

xaxis(pic2,"$x$",BottomTop,LeftTicks);
yaxis(pic2,"$y$",LeftRight,RightTicks);

xaxis(pic2,red+Dotted,YEquals(55.0,false));
yaxis(pic2,red+Dotted,XEquals(70.0,false));

// Fit pic to W of origin:
add(pic.fit(W));

// Fit pic2 to E of (5mm,0):
add((5mm,0),pic2.fit(E));
```



5. This example draws a graph of a parametrized curve.

   The calls to

   ```
   xlimits(picture pic=currentpicture, real min=-infinity,
           real max=infinity, bool crop=Crop);
   ```

   and the analogous function `ylimits` can be uncommented to restrict the respective axes limits for picture `pic` to the specified `min` and `max` values (alternatively, the function

`limits(pair, pair)` can be used to limit the axes to the box having opposite vertices at the given pairs). Existing objects in picture `pic` will be cropped to lie within the given limits unless `crop=NoCrop`. For example, if `xlimits` or `ylimits` are called with no arguments, existing objects in `currentpicture` will be cropped to the current graph limits. The function `crop(picture pic)` is equivalent to calling both `xlimits()` and `ylimits()`.

```
import graph;

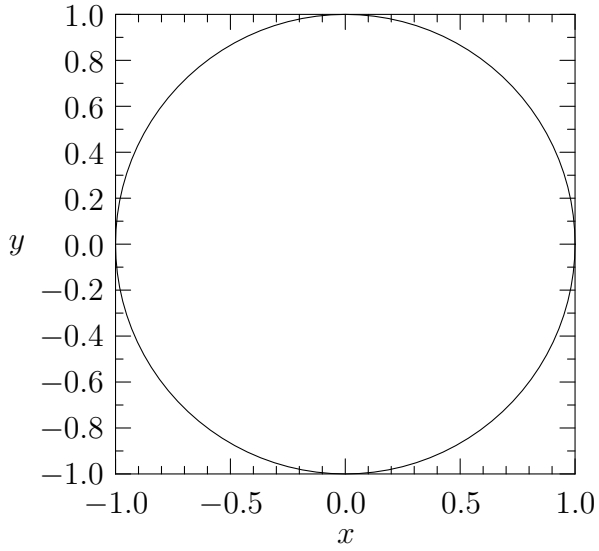
size(0,200);

real f(real t) {return cos(2pi*t);}
real g(real t) {return sin(2pi*t);}

draw(graph(f,g,0,1));

//xlimits(0,1);
//ylimits(-1,0);

xaxis("$x$",BottomTop,LeftTicks("$%#.1f$"));
yaxis("$y$",LeftRight,RightTicks("$%#.1f$"));
```



Axis scaling can be requested and/or automatic selection of the axis limits can be inhibited with the `scale` routine:

```
void scale(picture pic=currentpicture, scaleT x, scaleT y);
```

This sets the scalings for picture `pic`. The `graph` routines accept an optional `picture` argument for determining the appropriate scalings to use; if none is given, it uses those

set for `currentpicture`. All path coordinates (and any call to `limits`, etc.) refer to scaled data. Two frequently used scaling routines `Linear` and `Log` are predefined in `graph`.

Scaling routines can be given two optional boolean arguments: `automin` and `automax`. These default to `true`, but can be respectively set to `false` to disable automatic selection of "nice" axis minimum and maximum values. `Linear` can also take as an optional final argument a multiplicative scaling factor (e.g. for a depth axis, `Linear(-1)` requests axis reversal).

For example, to draw a log graph of a function, use `scale(Log,Log)`:

```
import graph;

size(200,200,IgnoreAspect);

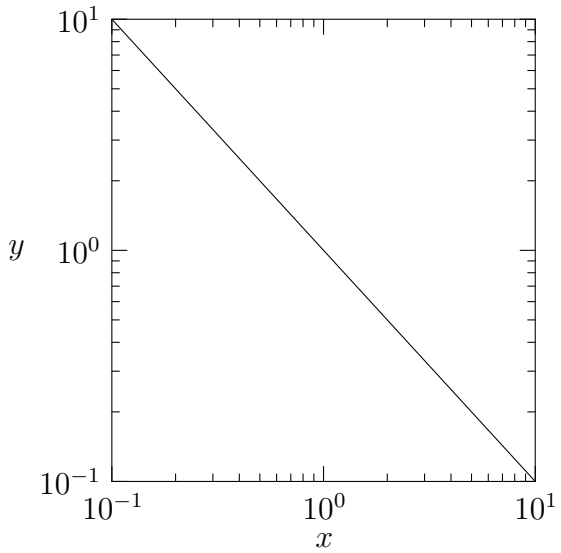
real f(real t) {return 1/t;}

scale(Log,Log);

draw(graph(f,0.1,10));

//xlimits(1,10);
//ylimits(0.1,1);

xaxis("$x$",BottomTop,LeftTicks);
yaxis("$y$",LeftRight,RightTicks);
```



6. `Asymptote` can draw secondary axes with the routines

```
picture secondaryX(picture primary=currentpicture, void f(picture));
picture secondaryY(picture primary=currentpicture, void f(picture));
```

In this example, `secondaryY` is used to draw a secondary linear y axis against a primary logarithmic y axis:

```
import graph;
texpreamble("\def\Arg{\mathop {\rm Arg}\nolimits}");

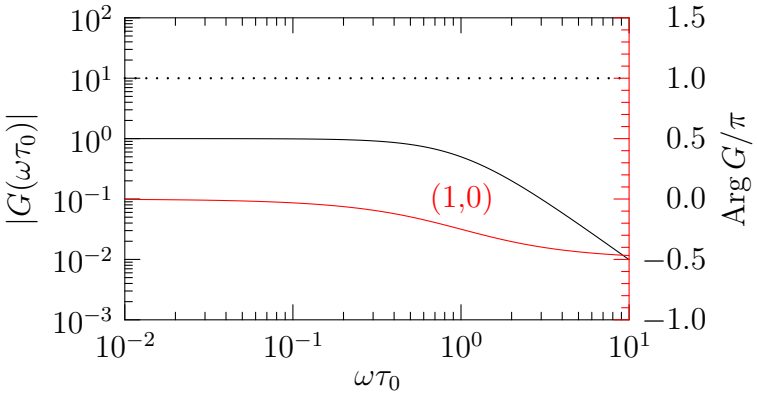
size(10cm,5cm,IgnoreAspect);

real ampl(real x) {return 1/(1+x^2);}
real phas(real x) {return -atan(x)/pi;}

scale(Log,Log);
draw(graph(ampl,0.01,10));
ylimits(.001,100);

xaxis("$\omega\tau_0$",BottomTop,LeftTicks);
yaxis("$|G(\omega\tau_0)|$",Left,RightTicks);

picture q=secondaryY(new void(picture p) {
                       scale(p,Log,Linear);
                       draw(p,graph(p,phas,0.01,10),red);
                       ylimits(p,-1,1.5);
                       yaxis(p,"$\Arg G/\pi$",black,red,Courier(),Right,
                             LeftTicks("$% #.1f$"));
                       xaxis(p,Dotted,YEquals(1,false));
                     });
label(q,"(1,0)",Scale(q,(1,0)),red);
add(q);
```



A secondary logarithmic y axis can be drawn like this:

```
import graph;

size(9cm,6cm,IgnoreAspect);
string data="secondaryaxis.csv";
```

```
file in=line(csv(input(data)));

string[] titlelabel=in;
string[] columnlabel=in;

real[][] a=dimension(in,0,0);
a=transpose(a);
real[] t=a[0], susceptible=a[1], infectious=a[2], dead=a[3], larvae=a[4];
real[] susceptibleM=a[5], exposed=a[6],infectiousM=a[7];

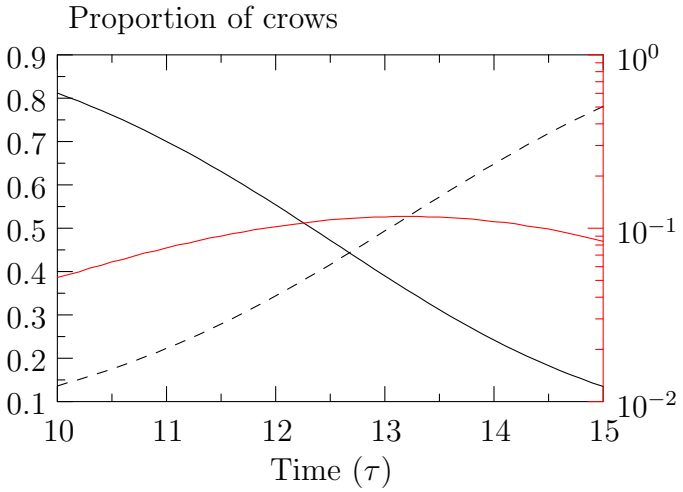
draw(graph(t,susceptible,t >= 10 && t <= 15));
draw(graph(t,dead,t >= 10 && t <= 15),dashed);

xaxis("Time ($\tau$)",BottomTop,LeftTicks);
yaxis(Left,RightTicks);

picture secondary=secondaryY(new void(picture pic) {
  scale(pic,Linear,Log);
  draw(pic,graph(pic,t,infectious,t >= 10 && t <= 15),red);
  yaxis(pic,black,red,Right,LeftTicks);
});

add(secondary);

label("Proportion of crows",point(NW),E,5mm*N);
```



7. Here is a histogram example, which uses the `stats` module.

```
import graph;
import stats;
```

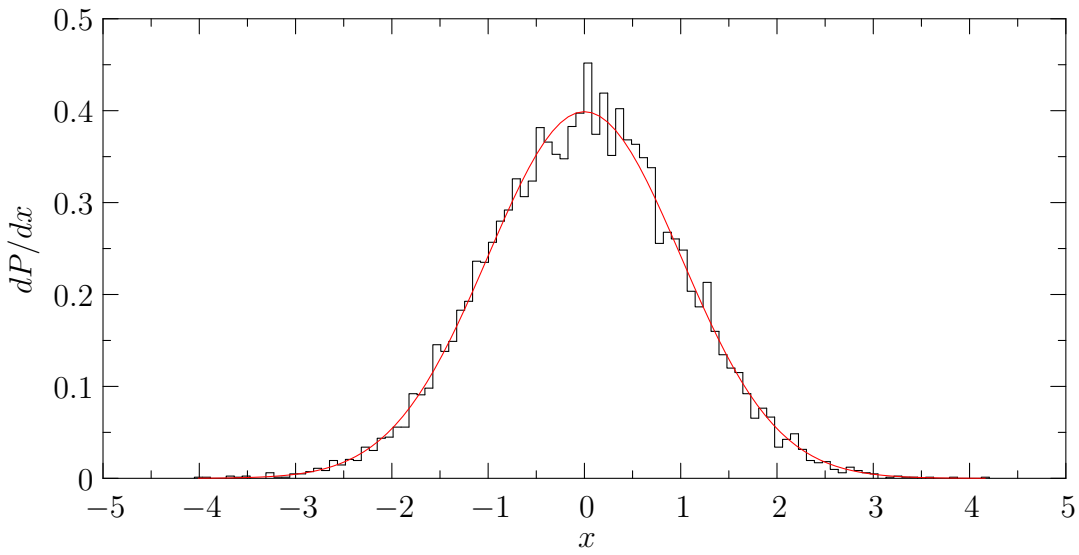
```
size(400,200,IgnoreAspect);

int n=10000;
real[] a=new real[n];
for(int i=0; i < n; ++i) a[i]=Gaussrand();

int nbins=100;
real dx=(max(a)-min(a))/(nbins-1);
real[] x=min(a)-dx/2+sequence(nbins+1)*dx;
real[] freq=frequency(x,a);
freq /= (dx*sum(freq));
histogram(x,freq);

draw(graph(Gaussian,min(a),max(a)),red);

xaxis("$x$",BottomTop,LeftTicks);
yaxis("$dP/dx$",LeftRight,RightTicks);
```

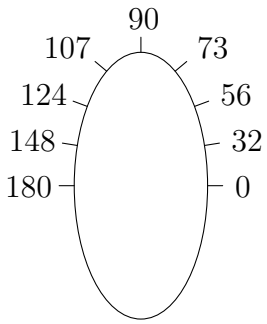


8. Here is an example that illustrates the general `axis` routine.

```
import graph;
size(0,100);

guide g=ellipse((0,0),100,200);
```

```
axis(g,0,0.5*arclength(g),RightTicks(8,"$%.0f$"),degrees);
```



9. `Asymptote` can also generate color density images and palettes. The following palettes are predefined in `palette.asy`:

   `pen[] Grayscale(int NColors=256)`
   : a grayscale palette;

   `pen[] Rainbow(int NColors=65501)`
   : a rainbow spectrum;

   `pen[] BWRainbow(int NColors=65485)`
   : a rainbow spectrum tapering off to black/white at the ends;

   `pen[] BWRainbow2(int NColors=65485)`
   : a double rainbow palette tapering off to black/white at the ends, with a linearly scaled intensity.

   The function `cmyk(pen[] Palette)` may be used to convert any of these palettes to the CMYK colorspace. A color density plot can added to a picture `pic` by generating from a real[][] array `data`, using palette `palette`, an image spanning the rectangular region with opposite corners at coordinates `initial` and `final`:

   ```
   void image(picture pic=currentpicture, real[][] data, pen[] palette,
              pair initial, pair final);
   ```

   An optionally labelled palette bar may be generated with the routine

   ```
   picture palette(real[][] data, real width=Ticksize,
                   pen[] palette, string s="", real position=0.5,
                   real angle=infinity, pair align=E, pair shift=0,
                   pair side=right, pen plabel=currentpen, pen p=plabel,
                   pen pticklabel=plabel, paletteticks ticks=PaletteTicks)
   ```

   The argument `paletteticks` is a special tick type (see [ticks], page 40) that takes the following arguments:

   ```
   paletteticks PaletteTicks(bool begin=true, int N=0, real Step=0,
                             string format=defaultformat, bool end=true)
   ```

   The image and palette bar can be fit (and optionally aligned) to a frame and added to picture `dest` at the location `origin` using `add(pair origin=(0,0), picture dest=currentpicture, frame)`:

```
import graph;
import palette;

int n=256;
real ninv=2pi/n;
real[][] v=new real[n][n];

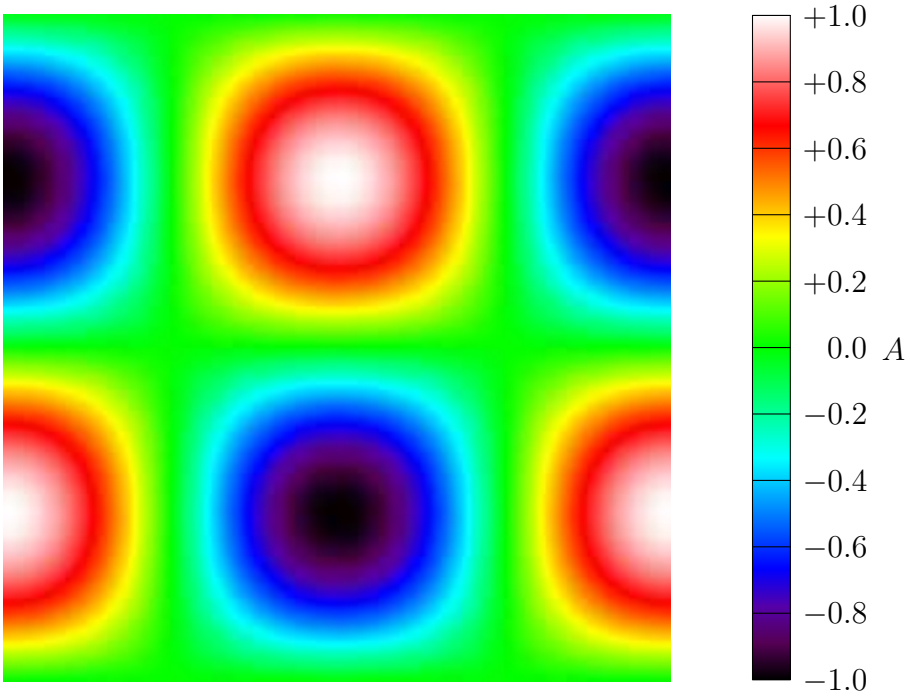
for(int i=0; i < n; ++i)
  for(int j=0; j < n; ++j)
    v[i][j]=sin(i*ninv)*cos(j*ninv);

pen[] Palette=BWRainbow();

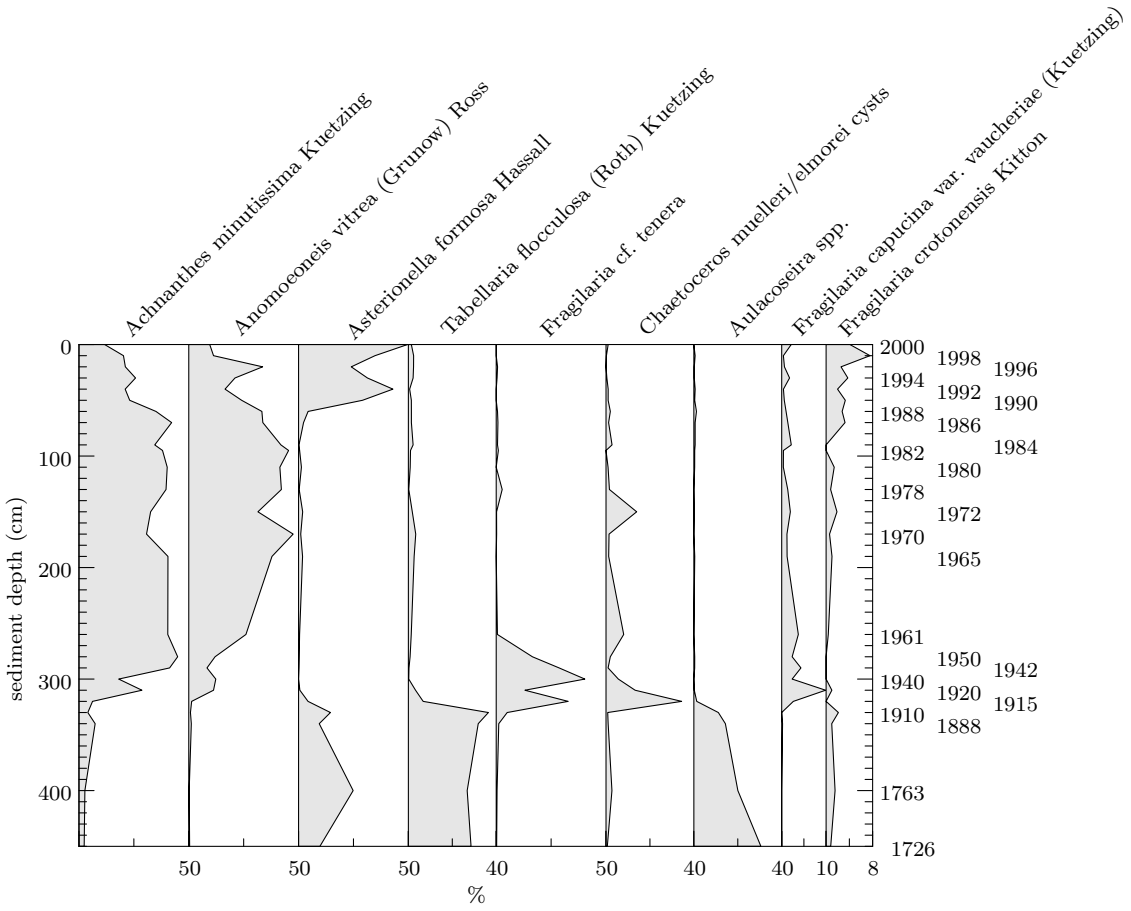
picture plot;

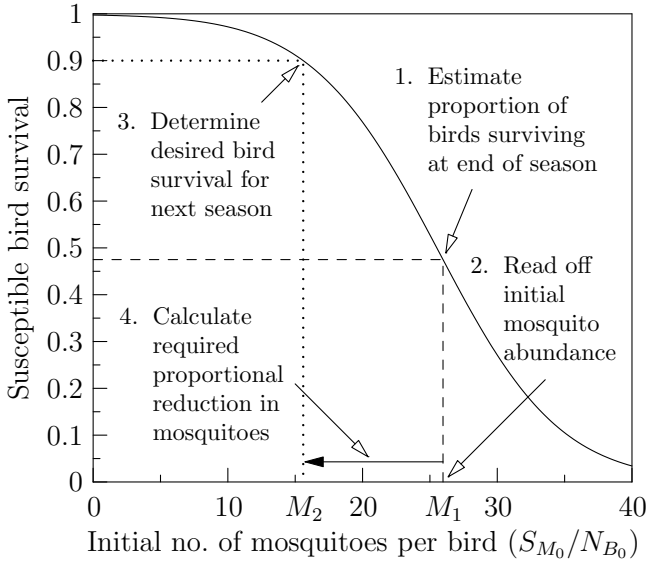
image(plot,v,Palette,(0,0),(1,1));
picture bar=palette(v,5mm,Palette,"$A$",PaletteTicks("$%+#.1f$"));

add(plot.fit(250,250,W));
add((1cm,0),bar.fit(0,250,E));
```



10. The following scientific graphs, which illustrate many features of `Asymptote`'s graphics routines, were generated from the examples `diatom.asy` and `westnile.asy`, using the comma-separated data in `diatom.csv` and `westnile.csv`.





4.14.4 `three`

This module extends the notion of a guide in `Asymptote` to three dimensions, introducing the new type `guide3`, along with the cycle terminator `cycle3` and tension operator `tension3`. A `guide3` will accept triple nodes, direction specifiers, and control points (using the usual `controls` operator). For the resulting spline to be invariant under 3D rotation, it is important to specify at least one direction specifier per node (or explicit control points).

These projections to two dimensions are predefined:

`oblique` The point `(x,y,z)` is projected to `(x-0.5z,y-0.5z)`. If an optional real argument is given to `oblique`, the negative z axis is drawn at this angle in degrees measured counterclockwise from the positive x axis.

`orthographic(triple camera)`
This projects three dimensions onto two using the view seen at the location `camera`. Parallel lines are projected to parallel lines.

`perspective(triple camera)`
This projects three dimensions onto two taking account of perspective, as seen from the location `camera`.

Three-dimensional objects may be transformed with one of the following built-in `transform3` types:

`shift(triple v)`
translates by the triple `v`;

`xscale3(real x)`
scales by `x` in the x direction;

`yscale3(real y)`
scales by `y` in the y direction;

`zscale3(real z)`
scales by `z` in the z direction;

`scale(real s)`
scales by `s` in the x, y, and z directions;

`rotate(real angle, triple v, triple u=(0,0,0))`
rotates by `angle` in degrees about the axis `u--v`;

Here is an example of a helix using perspective drawing:

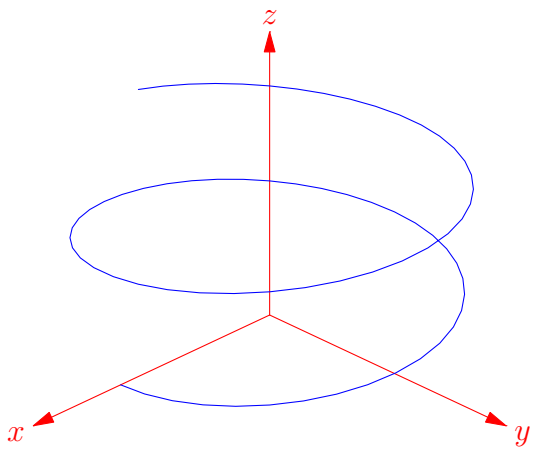
```
import three;
size(200,0);

currentprojection=perspective((4,4,3));

triple f(path p, real position) {
  pair z=point(p,position);
  return (z.x,z.y,position/length(p));
}

real r=1.5;
draw("$x$",(0,0,0)--(r,0,0),1,red,Arrow);
draw("$y$",(0,0,0)--(0,r,0),1,red,Arrow);
draw("$z$",(0,0,0)--(0,0,r),1,red,Arrow);

draw(graph(f,E..N..W..S..E..N..W..S),blue);
```



Here is an example of a surface plot:

```
import three;
size(200,0);

currentprojection=perspective((5,4,2));

real f(pair z) {return 0.5+exp(-abs(z)^2);}

draw((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle3);

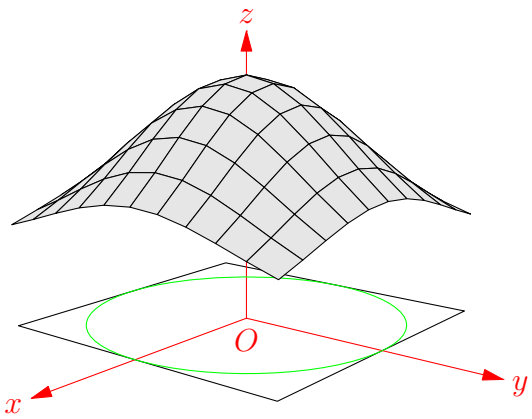
real r=1.75;
draw("$x$",(0,0,0)--(r,0,0),1,red,Arrow);
```

```
draw("$y$",(0,0,0)--(0,r,0),1,red,Arrow);
draw("$z$",(0,0,0)--(0,0,r),1,red,Arrow);

draw((1,0,0){Y}..(0,1,0){-X}..(-1,0,0){-Y}..(0,-1,0){X}..cycle3,green);

label("$O$",(0,0,0),S,red);

add(surface(f,(-1,-1),(1,1),n=10));
```



General hidden surface removal will be implemented by using a binary space partition and picture clipping in an upcoming release.

4.14.5 graph3d

This module will be developed into a full three-dimensional graphing package in the near future.

4.14.6 featpost3D

Until a complete 3D graphics package (see Section 4.14.5 [graph3d], page 57) is written, a preliminary port of the `MetaPost` 3D package `featpost3D` of L. Nobre G., C. Barbarosie and J. Schwaiger to `Asymptote` makes some 3D functionality already available, as illustrated by the examples `near_earth` and `conicurv`.

4.14.7 math

This package extends `Asymptote`'s mathematical capabilities with radian/degree conversion routines, point-in-polygon and intersection algorithms, 3D vectors, matrix arithmetic and inversion, and a linear equation solver (via Gauss-Jordan elimination).

Unlike `MetaPost`, `Asymptote` does not implicitly solve linear equations and therefore does not have the notion of a `whatever` unknown. The following routine provides a useful replacement for a common use of `whatever`: finding the intersection point of the lines through `P`, `Q` and `p`, `q`, respectively:

`pair extension(pair P, pair Q, pair p, pair q);`

Return the intersection point of the extensions of the line segments `PQ` and `pq`.

Here are some additional routines provided in the `math` package:

`void drawline(picture pic=currentpicture, pair P, pair Q, pen p=currentpen);`
: draw the visible portion of the (infinite) line going through `P` and `Q`, without altering the size of picture `pic`, using pen `p`.

`real intersection(triple P, triple Q, triple n, triple Z);`
: Return the intersection time of the extension of the line segment `PQ` with the plane perpendicular to `n` and passing through `Z`.

`triple intersectionpoint(triple n0, triple P0, triple n1, triple P1);`
: Return any point on the intersection of the two planes with normals `n0` and `n1` passing through points `P0` and `P1`, respectively. If the planes are parallel, return `(infinity,infinity,infinity)`.

`real[] solve(real[][] a, real[] b)`
: Solve the linear equation $ax = b$ by Gauss-Jordan elimination, returning the solution x, where `a` is an $n \times n$ matrix and `b` is an array of length n. For example:

```
import math;
real[][] a={{1,-2,3,0},{4,-5,6,2},{-7,-8,10,5},{1,50,1,-2}};
real[] b={7,19,33,3};
real[] x=solve(a,b);
write(a); write();
write(b); write();
write(x); write();
write(a*x);
```

`real[][] solve(real[][] a, real[][] b, bool overwrite=false)`
: Solve the linear equation $ax = b$ returning the solution x, where `a` is an $n \times n$ matrix and `b` is an $n \times m$ matrix. If overwrite=true, b is replaced by `x`.

`bool straight(path p)`
: returns `true` iff the path `p` is straight.

4.14.8 geometry

This module provides the beginnings of a geometry package. It currently includes a triangle structure and functions to draw interior arcs of triangles and perpendicular symbols.

4.14.9 stats

This package implements a Gaussian random number generator and a collection of statistics routines, including `histogram` and `leastsquares`.

4.14.10 patterns

This package implements `Postscript` tiling patterns and includes several convenient pattern generation routines.

4.14.11 palette

This package implements color density images and palette bars, along with several predefined palettes (see [images], page 52).

4.14.12 tree

This package implements an example of a dynamic binary search tree.

4.14.13 drawtree

This is a simple tree drawing module used by the example `treetest.asy`.

4.14.14 feynman

This package, contributed by Martin Wiebusch, is useful for drawing Feynman diagrams, as illustrated by the examples `eetomumu.asy` and `fermi.asy`.

4.14.15 MetaPost

This package provides some useful routines to help `MetaPost` users migrate old `MetaPost` code to `Asymptote`.

4.14.16 unicode

Import this package at the beginning of the file to instruct `LaTeX` to accept `unicode` (UTF-8) standardized international characters. You will also need to set up `LaTeX` support for `unicode` by unpacking in your `LaTeX` source directory (e.g. `/usr/share/texmf/tex/latex`) the file

`http://www.unruh.de/DniQ/latex/unicode/unicode.tgz`

and then running the command

`texhash`

4.14.17 latin1

If you don't have `LaTeX` support for `unicode` installed, you can enable support for Western European languages (ISO 8859-1) by importing the module `latin1`. This module can be used as a template for providing support for other ISO 8859 alphabets.

4.14.18 babel

This module implements the `LaTeX babel` package in `Asymptote`. For example:

```
import babel;
babel("german");
```

4.15 Static

Static qualifiers allocate the memory address of a variable in a higher enclosing scope.

For a function body, the variable is allocated in the block where the function is defined; so in the code

```
struct s {
  int count() {
    static int c=0;
    ++c;
    return c;
  }
}
```

there is one instance of the variable `c` for each object `s` (as opposed for each call of `count`).

Similarly, in

```
int factorial(int n) {
  int helper(int k) {
    static int x=1;
    x *= k;
    return k == 1 ? x : helper(k-1);
  }
  return helper(n);
}
```

there is one instance of `x` for every call to `factorial` (and not for every call to `helper`), so this is a correct, but ugly, implementation of factorial.

Similarly, a static variable declared within a structure is allocated in the block where the structure is defined. Thus,

```
struct A {
  struct B {
    static pair z;
  }
}
```

creates one object `z` for each object of type `A` created.

In this example,

```
int pow(int n, int k) {
  struct A {
    static int x=1;
    void helper() {
      x *= n;
    }
  }
  A operator init() {return new A;}
  for (int i=0; i < k; ++i) {
    A a;
    a.helper();
  }
  return A.x;
}
```

there is one instance of `x` for each call to `pow`, so this is an ugly implementation of exponentiation.

A file-level module is really just a structure, so a file `stuff.asy` containing

```
import other;
public int x=5;
int y=x*4;
int sqr(int x) {return x*x;}
```

is in some way equivalent to

```
struct stuff {
  import other;
  public int x=5;
  int y=x*4;
  int sqr(int x) { return x*x; }
}
```

When you `import` a module, it is like creating a new instance of the struct. That is,

```
int cube(int x) {
  import stuff;
  return x*sqr(x);
}
```

is essentially equivalent to

```
int cube(int x) {
  stuff s=new stuff;
  return x*s.sqr(x);
}
```

If you use static functions and variables from a module, then every import is referring to the same data. But every import of a module gets a different instance of its dynamic variables.

Declarations are dynamic by default. Suppose `X.asy` contains

```
public int x=0;
```

and `inc.asy` contains

```
import X;
void inc() {++x;}
```

Then the file `test.asy`:

```
import X;
import inc;
write(x);
inc();
write(x);
```

should write 0 twice, because the variable `x` imported in `inc.asy` is distinct from the variable `x` imported directly from `X.asy`.

If `inc.asy` were to contain only

```
void inc() {++x;}
```

then, in order for it to be able to access the variable `x` in `X.asy`, one needs to allocate `X.asy` in a higher enclosing block by making its import `static` in `test.asy`:

```
static import x;
import inc;
write(x);
inc();
write(x);
```

This writes first 0 and then 1.

5 Drawing commands

All of Asymptote's graphical capabilities are based on four primitive commands. The three PostScript drawing commands draw, fill, and clip add objects to a picture in the order in which they are executed, with the most recently drawn object appearing on top. The labeling command label can be used to add text labels and external EPS images, which will appear on top of the PostScript objects (since this is normally what one wants), but again in the relative order in which they were executed. After drawing objects on a picture, the picture can be output with the shipout function (see [shipout], page 21).

If you wish to draw PostScript objects on top of labels (or verbatim tex commands; see [tex], page 23), the layer command may be used to start a new PostScript/LaTeX layer:

```
void layer(picture pic=currentpicture);
```

The layer function gives one full control over the order in which objects are drawn. Layers are drawn sequentially, with the most recent layer appearing on top. Within each layer, labels, images, and verbatim tex commands are always drawn after the PostScript objects in that layer.

While some of these drawing commands take many options, they all have sensible default values (for example, the picture argument defaults to currentpicture).

5.1 draw

```
void draw(picture pic=currentpicture, string s="", real angle=0,
          path g, real position=0.5*length(g), pair align=0, pair shift=0,
          side side=RightSide, pen plabel=currentpen, pen p=plabel,
          arrowbar arrow=None, arrowbar bar=None, margin margin=NoMargin,
          string legend="", frame mark=nullframe, bool putmark=Above);
```

Draw the path g on the picture pic, optionally labeled by string s writing at angle angle (in degrees) at position position and aligned using the pair align (or, if zero, aligned on side side) and true shift shift, using pen plabel for labelling and pen p for drawing, with optional drawing attributes (arrows, bars, margins, legend). Only one parameter, the path, is required. For convenience, the arguments arrow and bar may be specified in either order. The argument legend is a string to use in constructing an optional legend entry. An optional frame mark may be drawn at every node, on on top of (underneath) the path if putmark=Above (putmark=Below).

The possible values of side are LeftSide, Center, and RightSide. This parameter specifies the location of the path label relative to the direction in which the path is drawn. The value of side is ignored if align is nonzero or if position corresponds to an endpoint of the path g.

Bars are useful for indicating dimensions. The possible values of bar are None, BeginBar, EndBar (or equivalently Bar), and Bars (which draws a bar at both ends of the path). Each of these bar specifiers (except for None) will accept an optional real argument that denotes the length of the bar in PostScript coordinates. The default bar length is barsize(p).

The possible values of arrow are None, Blank (which draws no arrows or path), BeginArrow, EndArrow (or equivalently Arrow), and Arrows (which draws an arrow at

both ends of the path). Each of these arrow specifiers (except for `None` and `Blank`) may be given the optional arguments real `size` (arrowhead size in `PostScript` coordinates), real `angle` (arrowhead angle in degrees), `Fill` or `NoFill`, and a relative real `position` along the path (an `arctime`) where the tip of the arrow should be placed. The default arrowhead size is `arrowheadsize(p)`. There are also arrow versions with slightly modified default values of `size` and `angle` suitable for curved arrows: `BeginArcArrow`, `EndArcArrow` (or equivalently `ArcArrow`), and `ArcArrows`.

Margins can be used to shrink the visible portion of a path by `labelmargin(p)` to avoid overlap with other drawn objects. Typical values of `margin` are `NoMargin`, `BeginMargin`, `EndMargin` (or equivalently `Margin`), and `Margins` (which leaves a margin at both ends of the path). One may use `Margin(real begin, real end)` to specify the size of the beginning and ending margin, respectively, in multiples of the units `labelmargin(p)` used for aligning labels. Alternatively, `BeginPenMargin`, `EndPenMargin` (or equivalently `PenMargin`), `PenMargins`, `PenMargin(real begin, real end)` specify a margin in units of the pen linewidth, taking account of the pen linewidth when drawing the path or arrow. For example, use `DotMargin`, an abbreviation for `PenMargin(-0.5,0.5*dotfactor)`, to draw from the usual beginning point just up to the boundary of an end dot of width `dotfactor*linewidth(p)`. The qualifiers `BeginDotMargin`, `EndDotMargin`, and `DotMargins` work similarly. The qualifier `TrueMargin(real begin, real end)` allows one to specify a margin directly in `PostScript` units, independent of the pen linewidth.

The use of arrows, bars, and margins is illustrated by the examples `Pythagoras.asy`, `sqrtx01.asy`, and `triads.asy` (installed in `/usr/local/share/doc/asymptote` by default).

The legend for a picture `pic` can be fit and aligned to a frame with the routine

```
frame legend(picture pic=currentpicture, pair dir=0);
```

this legend frame can then be added about the point `origin` to a picture `dest`:

```
add(pair origin=(0,0), picture dest=currentpicture, frame);
```

To draw a dot, simply draw a path containing a single point. The `dot` command defined in `plain.asy` draws a dot with a diameter given by either the default linewidth magnified by `dotfactor` (6 by default) or an explicit pen linewidth. If no pen argument is given, `currentpen`, with its linewidth magnified by `dotfactor`, is used:

```
void dot(picture pic=currentpicture, pair c);
void dot(picture pic=currentpicture, pair c, pen p);
void dot(picture pic=currentpicture, pair[] c, pen p=currentpen);
```

The third routine draws a dot at every point of an pair array `c`. One can also draw a dot at every node of a path:

```
void dot(picture pic=currentpicture, guide g, pen p=currentpen);
void dot(picture pic=currentpicture, path[] g, pen p=currentpen);
```

See section Section 5.1 [draw], page 62 for a more general way of marking path nodes.

To draw a fixed-sized object (in `PostScript` coordinates) about the user coordinate `origin`, use the routine

```
void draw(pair origin, picture pic=currentpicture, string s="", real angle=0,
          path g, real position=0.5*length(g), pair align=0, pair shift=0,
          side side=RightSide, pen plabel=currentpen, pen p=plabel,
```

```
          arrowbar arrow=None, arrowbar bar=None, margin margin=NoMargin,
          string legend="", frame mark=nullframe, bool putmark=Above);
```

5.2 fill

```
void fill(picture pic=currentpicture, path g,
          pen pena=currentpen, pair a=0, real ra=0,
          pen penb=currentpen, pair b=0, real rb=0);
```

Fill the interior region bounded by the cyclic path `g` on the picture `pic`, using the pen `pena`.

Gradient shading requires that `pena` and `penb` contain distinct colors (not patterns). If `a` != `b` and `ra`=`rb`=0, fill with a color gradient varying smoothly from `pena` to `penb` in the direction of the line segment `a--b`. If `ra` != 0 or `rb` != 0, fill with a color gradient varying smoothly from `pena` on the circle with center `a` and radius `ra` to `penb` on the circle with center `b` and radius `rb`:

```
size(100,0);
fill(unitsquare,yellow,(0,0),0,red,(0,0),1);
```

There is also a convenient `filldraw` command, which fills the path and then draws in the boundary. One can specify separate pens for each operation:

```
void filldraw(picture pic=currentpicture, path g, pen pena=currentpen,
              pen drawpen=currentpen, pair a=0, real ra=0,
              pen penb=currentpen, pair b=0, real rb=0);
```

There are also fixed-size versions of `fill` and `filldraw` that allow one to fill an object described in `PostScript` coordinates about the user coordinate `origin`:

```
void fill(pair origin, picture pic=currentpicture, path g,
          pen pena=currentpen, pair a=0, real ra=0,
          pen penb=currentpen, pair b=0, real rb=0);

void filldraw(pair origin, picture pic=currentpicture, path g,
              pen pena=currentpen,
              pen drawpen=currentpen, pair a=0, real ra=0,
              pen penb=currentpen, pair b=0, real rb=0);
```

The following routines use `evenodd` clipping together with the `^^` operator to unfill a region:

```
void unfill(picture pic=currentpicture, path g);
void unfill(picture pic=currentpicture, path[] g);
```

5.3 clip

```
void clip(picture pic=currentpicture, path g, pen p=currentpen);
```

Clip the current contents of picture `pic` to the region bounded by the path `g`, using fillrule `p` (see [fillrule], page 17). For an illustration of picture clipping, see the first example in Chapter 6 [LaTeX usage], page 69.

5.4 label

```
void label(picture pic=currentpicture, string s, real angle=0,
           pair position, pair align=0, pair shift=0, pen p=currentpen);
```

Draw label `s` on `pic`, writing at angle `angle` (in degrees), at user coordinate `position`, aligned with the vector (compass direction) `align` and `PostScript` coordinate translation `shift`, using pen `p`. If `align` is 0, the label will be centered at `position`; otherwise it will be aligned in the direction of `align` and displaced from the user coordinate `position` by the `PostScript` offset `align*labelmargin(p)`.

A label with an arrow can be produced with one of the following routines:

```
void arrow(picture pic=currentpicture, string s="", real angle=0,
           pair b, pair align, real length=arrowlength, pair shift=0,
           pen p=currentpen, real size=arrowsize(p), real Angle=arrowangle,
           arrowhead arrowhead=Fill,margin margin=EndMargin);

void arrow(picture pic=currentpicture, string s, real angle=0,
           pair shift=0, path g, pen p=currentpen, real size=arrowsize(p),
           real Angle=arrowangle, arrowhead arrowhead=Fill,
           margin margin=NoMargin);
```

The function `string include(string name, string options="")` returns a string that can be used to include an encapsulated `PostScript` (EPS) file. Here, `name` is the name of the file to include and `options` is a string containing a comma-separated list of optional bounding box (`bb=llx lly urx ury`), width (`width=value`), height (`height=value`), rotation (`angle=value`), scaling (`scale=factor`), clipping (`clip=bool`), and draft mode (`draftx=bool`) parameters. The `layer()` function can be used to force future objects to be drawn on top of the included image:

```
label(include("file.eps","width=1cm"),(0,0),NE);
layer();
```

The `string baseline(string s, pair align=S, string template="M")` function can be used to enlarge the bounding box of letters aligned below a horizontal line to match a given template, so that their baselines lie on a horizontal line. See `Pythagoras.asy` for an example.

The `string minipage(string s, width=100pt)` function can be used to format string `s` into a paragraph of width `width`, as illustrated in the following example:

```
size(9cm,10cm,IgnoreAspect);

pair d=(1,0.25);
real s=1.6d.x;
```

```
real y=0.6;
defaultpen(fontsize(8));

picture box(string s, pair z=(0,0)) {
  picture pic;
  draw(pic,box(-d/2,d/2));
  label(pic,s,(0,0));
  return shift(z)*pic;
}

label("Birds",(0,y));
picture removed=box("Removed ($R_B$)");
picture infectious=box("Infectious ($I_B$)",(0,-1.5));
picture susceptible=box("Susceptible ($S_B$)",(0,-3));

add(removed);
add(infectious);
add(susceptible);

label("Mosquitoes",(s,y));
picture larval=box("Larval ($L_M$)",(s,0));
picture susceptibleM=box("Susceptible ($S_M$)",(s,-1));
picture exposed=box("Exposed ($E_M$)",(s,-2));
picture infectiousM=box("Infectious ($I_M$)",(s,-3));

add(larval);
add(susceptibleM);
add(exposed);
add(infectiousM);

path ls=point(larval,S)--point(susceptibleM,N);
path se=point(susceptibleM,S)--point(exposed,N);
path ei=point(exposed,S)--point(infectiousM,N);
path si=point(susceptible,N)--point(infectious,S);

draw(minipage("\flushright{recovery rate ($g$) \& death rate from virus
($\mu_V$)}",40pt),point(infectious,N)--point(removed,S),LeftSide,Arrow,
PenMargin);

draw(si,LeftSide,Arrow,PenMargin);

draw(minipage("\flushright{maturation rate ($m$)}",50pt),ls,RightSide,
Arrow,PenMargin);
draw(minipage("\flushright{viral incubation rate ($k$)}",40pt),ei,
RightSide,Arrow,PenMargin);

path ise=point(infectious,E)--point(se,0.5);
```

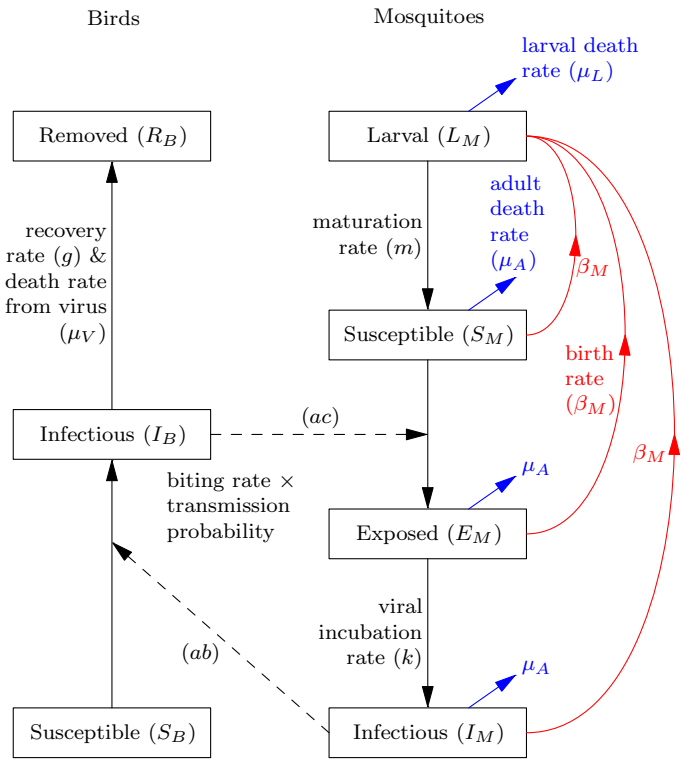
```
draw("$(ac)$",ise,LeftSide,dashed,Arrow,PenMargin);
label(minipage("\flushleft{biting rate $\times$ transmission
probability}",50pt),point(infectious,SE),dir(-60)+S);

path isi=point(infectiousM,W)--point(si,2.0/3);

draw("$(ab)$",isi,LeftSide,dashed,Arrow,PenMargin);
draw(se,LeftSide,Arrow,PenMargin);

real t=2.0;
draw("$\beta_M$",
     point(susceptibleM,E){right}..tension t..{left}point(larval,E),
     2*(S+SE),red,Arrow(Fill,0.5));
draw(minipage("\flushleft{birth rate ($\beta_M$)}",20pt),
     point(exposed,E){right}..tension t..{left}point(larval,E),2SW,red,
     Arrow(Fill,0.5));
draw("$\beta_M$",
     point(infectiousM,E){right}..tension t..{left}point(larval,E),2SW,
     red,Arrow(Fill,0.5));

path arrow=(0,0)--0.75cm*dir(35);
draw(point(larval,NNE),
          minipage("\flushleft{larval death rate ($\mu_L$)}",45pt),
          arrow,1,blue,Arrow);
draw(point(susceptibleM,NNE),
          minipage("\flushleft{adult death rate ($\mu_A$)}",20pt),
          arrow,1,N,blue,Arrow);
draw(point(exposed,NNE),"$\mu_A$",arrow,1,blue,Arrow);
draw(point(infectiousM,NNE),"$\mu_A$",arrow,1,blue,Arrow);
```



One can prevent labels from overwriting one another with the `overwrite` pen attribute (see [overwrite], page 20).

6 LaTeX usage

Asymptote comes with a convenient LaTeX style file asymptote.sty that makes LaTeX Asymptote-aware. Entering Asymptote code directly into the LaTeX source file, at the point where it is needed, keeps figures organized and avoids the need to invent new file names for each figure.

For example, the sample LaTeX file below, named latexusage.tex, can be run as follows:

```
latex latexusage
asy latexusage
latex latexusage
```

Here now is latexusage.tex:

```
\documentclass[12pt]{article}
\usepackage{asymptote}
\begin{document}
\begin{asydef}
// Global definitions can be put here.
\end{asydef}

Here is a figure produced with Asymptote, drawn to width 5cm:

\begin{center}
\begin{asy}
size(5cm,0);
pen colour1=red;
pen colour2=green;

pair z0=(0,0);
pair z1=(-1,0);
pair z2=(1,0);
real r=1.5;
guide c1=circle(z1,r);
guide c2=circle(z2,r);
fill(c1,colour1);
fill(c2,colour2);

picture intersection=new picture;
fill(intersection,c1,colour1+colour2);
clip(intersection,c2);

add(intersection);

draw(c1);
draw(c2);

label("$A$",z1);
```

```
label("$B$",z2);

pair z=(0,-2);
real m=3;
margin BigMargin=Margin(0,m*dot(unit(z1-z),unit(z0-z)));

draw("$A\cap B$",conj(z)--z0,0,Arrow,BigMargin);
draw("$A\cup B$",z--z0,0,Arrow,BigMargin);
draw(z--z1,Arrow,Margin(0,m));
draw(z--z2,Arrow,Margin(0,m));

shipout(bbox(0.25cm));
\end{asy}
\end{center}

Each graph is drawn in its own environment. One can specify the width
and height to \LaTeX\ explicitly:

\begin{center}
\begin{asy}[3cm,0]
guide center = (0,1){W}..tension 0.8..(0,0){(1,-.5)}..tension 0.8..{W}(0,-1);

draw((0,1)..(-1,0)..(0,-1));
filldraw(center{E}..{N}(1,0)..{W}cycle);
fill(circle((0,0.5),0.125),white);
fill(circle((0,-0.5),0.125));
\end{asy}
\end{center}

The default width is the full line width:

\begin{center}
\begin{asy}
import graph;

real f(real x) {return sqrt(x);}
pair F(real x) {return (x,f(x));}

real g(real x) {return -sqrt(x);}
pair G(real x) {return (x,g(x));}

guide p=(0,0)--graph(f,0,1,Spline)--(1,0);
fill(p--cycle,lightgray);
draw(p);
draw((0,0)--graph(g,0,1,Spline)--(1,0),dotted);

real x=0.5;
```

```
pair c=(4,0);

transform T=xscale(0.5);
draw((2.695,0),T*arc(0,0.30cm,20,340),ArcArrow);
fill(shift(c)*T*circle(0,-f(x)),red+white);
draw(F(x)--c+(0,f(x)),dashed+red);
draw(G(x)--c+(0,g(x)),dashed+red);

labeldot((1,1));
arrow("$y=\sqrt{x}$",F(0.7),N);

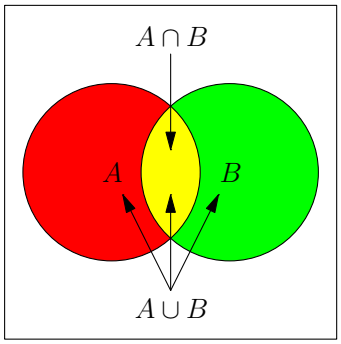
arrow((3,0.5*f(x)),W,1cm,red);
arrow((3,-0.5*f(x)),W,1cm,red);

xaxis(0,c.x,"$x$",dashed);
yaxis("$y$");

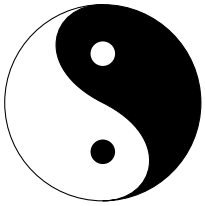
draw("$r$",(x,0)--F(x),E,red,Arrows,BeginBar,PenMargins);
draw("$r$",(x,0)--G(x),E,red,Arrows,PenMargins);
draw("$r$",c--c+(0,f(x)),Arrow,PenMargin);
dot(c);
\end{asy}
\end{center}

\end{document}
```

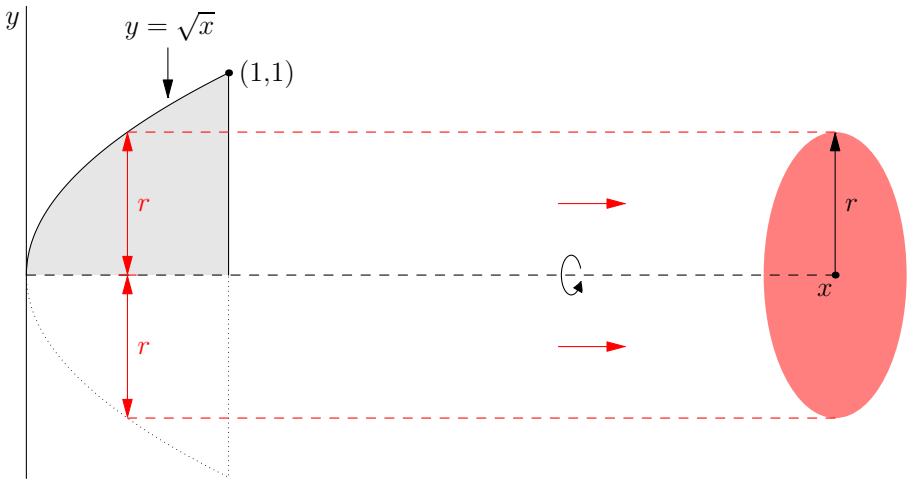
Here is a figure produced with Asymptote, drawn to width 5cm:



Each graph is drawn in its own environment. One can specify the width and height to LaTeX explicitly:



The default width is the full line width:



7 Options

Type `asy -h` to see the full list of command line options supported by `Asymptote`:

```
Usage: asy [options] [file ...]

Options:

-V, -View        View output file
-x magnification Deconstruct into transparent GIF objects
-c               Clear GUI operations
-i               Ignore GUI operations
-f format        Convert each output file to specified format
-o name          (First) output file name
-h, -help        Show summary of options
-O pair          PostScript offset
-C               Center on page (default)
-B               Align to bottom-left corner of page
-T               Align to top-left corner of page
-Z               Position origin at (0,0) (implies -L)
-d               Enable debugging messages
-v, -verbose     Increase verbosity level
-k               Keep intermediate files
-L               Disable LaTeX label postprocessing
-p               Parse test
-s               Translate test
-l               List available global functions and variables.
-m               Mask fpu exceptions (default for interactive mode)
-nomask          Don't mask fpu exceptions (default for batch mode)
-bw              Convert colors to black and white
-gray            Convert colors to grayscale
-rgb             Convert cmyk colors to rgb
-cmyk            Convert rgb colors to cmyk
-safe            Disable system call (default)
-unsafe          Enable system call
-noplain         Disable automatic importing of plain
```

If no arguments are given, `Asymptote` runs in interactive mode (see Chapter 8 [Interactive mode], page 75).

If `-` is given as the file argument, `Asymptote` reads from standard input.

If multiple files are specified, they are treated as separate `Asymptote` runs.

An alternative output format may be produced by using the `-f format` option. This supports any format supported by the `ImageMagick convert` program (a relatively recent version of which must be installed). `ImageMagick` should be configured with `--enable-lzw` to produce LZW-compressed GIFs (the Unisys LZW patent expired on June 20, 2003).

If the option `-unsafe` is given, `Asymptote` runs in unsafe mode. This enables the `int system(string)` call, allowing one to execute arbitrary shell commands. The default mode, `-safe`, disables this call.

By default, `Asymptote` attempts to center the figure on the page, assuming that the paper type is `letter`. The default paper type may be changed to `a4` with the environment variable `ASYMPTOTE_PAPERTYPE`. Currently only these two paper types are supported. Note that adding a new type, say `poster`, will also require defining `posterSize` in the dvips configuration file.

A `PostScript` offset may be specified as a pair (in bp units) with the `-O` option:

```
asy -O 0,0 file
```

The default offset is zero. The offset is adjusted if it would result in a negative vertical bounding box coordinate.

Additional debugging output is produced with each additional `-v` option:

`-v` Display top-level module and final output file names.

`-vv` Also display imported module names and final `LaTeX` and `dvips` processing information.

`-vvv` Also output `LaTeX` bidirectional pipe diagnostics.

`-vvvv` Also output knot solver diagnostics.

`-vvvvv` Also output `Asymptote` traceback diagnostics.

8 Interactive mode

Because `Asymptote` as currently designed does not natively support a line-at-a-time mode, interactive mode is currently simulated by rerunning all previous commands in a new module as each line is entered. This emulation will eventually become unnecessary, once `Asymptote`'s module handling is redesigned to support incremental input.

In interactive mode, it is not necessary to type a semicolon at the end of each line; one will automatically be appended.

The following special commands are supported only in interactive mode and must be entered immediately after the prompt:

`quit` exits interactive mode (abbreviated as `q`).

`redraw` re-executes the previous commands

`reset` resets `Asymptote` to its initial state, except that a prior call to `scroll` is respected (see [scroll], page 36).

`input FILE`
: directly inputs code from `FILE` into `Asymptote`. Any existing `input` commands will automatically be cleared by first doing a `reset`, if necessary. For this reason, after a previous `input` command, first issue an explicit `reset` before entering any code that you want to execute before a new `input` command. For convenience, a trailing semi-colon followed by optional `Asymptote` commands may be entered on the same line.

Typing `ctrl-C` interrupts the execution of Asymptote code and returns control to the interactive prompt.

9 Graphical User Interface

In the event that adjustments to the final figure are required, the Graphical User Interface (GUI) `xasy` included with `Asymptote` allows you to move graphical objects around with mouse `Button-1`. The modified layout can be written to disk with the `w` key in a form readable to `Asymptote`. A wheel mouse is convenient for raising and lowering objects, to expose the object to be moved. If a wheel mouse is not available, mouse `Button-2` (lower) can be used repeatedly instead. Here are the currently defined key mappings:

`z` undo

`r` redo

`<Delete>` delete

`w` write

`q` quit

One can also draw connected line segments by holding down the shift key and pressing mouse `Button-1` at each desired node. Releasing the shift key ends the definition of the path. More features will be added to this preliminary GUI soon.

As `xasy` is written in the interactive scripting language Python/TK, it requires that both Python and the `tkinter` package be installed. To use `xasy`, one must first deconstruct `Asymptote` pictures into transparent GIF images with the command `asy -xN`, where `N` denotes the magnification (say 2). The command `asy -VxN` automatically invokes `xasy` once deconstruction is complete.

Deconstruction of compound objects (such as arrows) can be prevented by enclosing them within the commands

```
void begingroup(picture pic=currentpicture);
void endgroup(picture pic=currentpicture);
```

By default, the elements of a picture or frame will be grouped together on adding them to a picture. However, the elements of a frame added to another frame are not grouped together by default: their elements will individually deconstructed (see [add], page 23).

10 PostScript to Asymptote

An Asymptote backend to pstoedit (http://pstoedit.net) is provided in the file pstoedit-3.40asy.patch in the patches directory of the Asymptote source. Unlike virtually all other pstoedit backends, this driver includes native clipping, even-odd fill rule, and PostScript subpath support. To install pstoedit with this backend, do the following (denoting the location of the Asymptote source directory by ASYMPTOTE_SOURCE):

```
tar -zxf pstoedit-3.40.tar.gz
cd pstoedit-3.40
patch -p1 < ASYMPTOTE_SOURCE/patches/pstoedit-3.40asy.patch
./configure --prefix=/usr
make install
```

For example, try:

```
asy -V /usr/local/share/doc/asymptote/venn.asy
pstoedit venn.eps test.asy
asy -V test
```

If the line widths aren't quite correct, try giving pstoedit the -dis option. If the fonts aren't typeset correctly, try giving pstoedit the -dt option.

11 Help

Questions on installing and using Asymptote should be sent to the `Asymptote` mailing list, after first subscribing (currently only members can post):

`https://lists.sourceforge.net/lists/listinfo/asymptote-discuss`

If you find a bug in Asymptote, please check (if possible) whether the bug is still present in the latest CVS version before submitting a bug report. New bugs can be submitted using the Bug Tracking System at

`http://sourceforge.net/projects/asymptote`

`Asymptote` can be configured with the optional GNU library `libsigsegv`, available from `http://libsigsegv.sourceforge.net`, which allows one to distinguish user-generated `Asymptote` stack overflows (see [stack overflow], page 29) from true segmentation faults (due to internal C++ programming errors; please submit the `Asymptote` code that generates such segmentation faults along with your bug report).

12 Acknowledgments

Financial support for the development of `Asymptote` was generously provided by the Natural Sciences and Engineering Research Council of Canada, the Pacific Institute for Mathematical Sciences, and the University of Alberta Faculty of Science.

We also would like to acknowledge the previous work of John D. Hobby, author of the program `MetaPost` that inspired the development of `Asymptote`, and Donald E. Knuth, author of TEX and `MetaFont` (on which `MetaPost` is based).

The authors of `Asymptote` are Andy Hammerlindl, John Bowman, and Tom Prince. Sean Healy designed the `Asymptote` logo.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z