Asymptote

For version 0.95

Table of Contents

1 Description 1

2 Installation 2
2.1 UNIX binary distributions 2
2.2 Microsoft Windows 2
2.3 Environment variables 2
2.4 Search paths 3
2.5 Compiling from UNIX source 4
2.6 Editing modes 4
2.7 CVS 5
2.8 Uninstall 5

3 Examples 6

4 Programming 8
4.1 Data types 8
4.2 Guides and paths 12
4.3 Pens 19
4.4 Transforms 24
4.5 Frames and pictures 24
4.6 Files 29
4.7 Structures 30
4.8 Operators 32
4.8.1 Arithmetic & logical operators 32
4.8.2 Self & prefix operators 33
4.8.3 User-defined operators 33
4.9 Implicit scaling 34
4.10 Functions 35
4.10.1 Default arguments 36
4.10.2 Named arguments 36
4.10.3 Rest arguments 37
4.10.4 Mathematical functions 39
4.11 Arrays 39
4.12 Casts 43
4.13 Import 44
4.13.1 `plain` 47
4.13.2 `simplex` 47
4.13.3 `graph` 47
4.13.4 `three` 70
4.13.5 `graph3` 74
4.13.6 `featpost3D` 79
4.13.7 `math` 79

4.13.8 geometry 80
4.13.9 stats 80
4.13.10 patterns 80
4.13.11 palette 80
4.13.12 tree 80
4.13.13 drawtree 80
4.13.14 feynman 80
4.13.15 MetaPost 80
4.13.16 unicode 80
4.13.17 latin1 81
4.13.18 babel 81
4.14 Static 81

5 Drawing commands 83
5.1 draw 83
5.2 fill 85
5.3 clip 86
5.4 label 86

6 LaTeX usage 90

7 Options 95

8 Interactive mode 97

9 Graphical User Interface 98

10 PostScript to Asymptote 99

11 Help 100

12 Acknowledgments 101

Index 102

1 Description

Asymptote is a powerful descriptive vector graphics language that provides a mathematical 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 83).

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 4.13.3 [graph], page 47). Examples of Asymptote code and output, including animations, 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

After following the instructions for your specific distribution, please see also Section 2.3 [Environment variables], page 2.

We recommend subscribing to new release announcements at

http://freshmeat.net/subscribe/50750

Users may also wish to monitor the `Asymptote` forum:

http://sourceforge.net/forum/monitor.php?forum_id=409349

and provide guidance to others by rating the `Asymptote` project:

http://freshmeat.net/projects/asy

2.1 UNIX binary distributions

Here are the commands that the root user can use to install the `Linux i386` 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` command installs a LaTeX style file):

```
tar -C / -zxf asymptote-x.xx.ARCH.tar.gz
texhash
```

Example code will be installed by default in `/usr/local/share/doc/asymptote`.

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

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

2.2 Microsoft Windows

Users of the `Microsoft Windows` operating system can install the self-extracting `Asymptote` executable `asymptote-install-x.xx.exe`. A working `TeX` implementation (such as the one available at http://www.miktex.org) will be required to typeset labels. The `Python` interpretor from http://www.python.org is only required if you wish to try out the graphical user interface (see Chapter 9 [GUI], page 98).

Example code will be installed by default in `c:\Program Files\Asymptote\examples`.

2.3 Environment variables

In interactive mode, or when given the `-V` option (the default under `MSDOS`), `Asymptote` will automatically invoke the `PostScript` viewer `gv` (under `UNIX`) or `gsview` (under `MSDOS`; available from http://www.cs.wisc.edu/~ghost/gsview/) to display graphical output. These defaults may be overridden with the optional environment variable `ASYMPTOTE_PSVIEWER`. For PDF format output, the `ASYMPTOTE_GS` environment variable specifies the location of the `PostScript`-to-PDF processor `gs` and `ASYMPTOTE_PDFVIEWER` specifies the location of the PDF viewer. The graphical user interface may also required setting the variable `ASYMPTOTE_PYTHON` if `python` is installed in a nonstandard location. Here are the default values of these environment variables:

```
UNIX:
export ASYMPTOTE_PSVIEWER=gv
export ASYMPTOTE_PDFVIEWER=acroread
export ASYMPTOTE_GS=gs
export ASYMPTOTE_PYTHON=

MSDOS:
set ASYMPTOTE_PSVIEWER=c:\Program Files\Ghostgum\gsview\gsview32.exe
set ASYMPTOTE_PDFVIEWER=
              c:\Program Files\Adobe\Acrobat 7.0\Reader\AcroRd32.exe
set ASYMPTOTE_GS=c:\Program Files\gs\gs8.51\bin\gswin32.exe
set ASYMPTOTE_PYTHON=c:\Python24\python.exe
```

To set environment variables under `Microsoft Windows XP`:

1. Click on the `Start` button.
2. Right-click on `My Computer`.
3. Choose `Properties` from the popup menu.
4. Click the `Advanced` tab.
5. Click the `Environment Variables` button.

The following environment variables normally do not require adjustment:

```
ASYMPTOTE_LATEX
ASYMPTOTE_DVIPS
ASYMPTOTE_CONVERT
ASYMPTOTE_DISPLAY
ASYMPTOTE_ANIMATE
ASYMPTOTE_XASY
```

To properly support interactive mode, the `PostScript` viewer should be capable of automatically redrawing whenever the output file is updated. The default `PostScript` viewer `gv` supports this (via a `SIGHUP` signal). Users of `ggv` will need to enable `Watch file` under `Edit/Postscript Viewer Preferences` and `gsview` users will need to enable `Options/Auto Redisplay` (however, under `MSDOS` it is still necessary to click on the `gsview` window; under `UNIX` one must manually redisplay by pressing the `r` key).

The patches supplied in the `patches` directory fix known bugs in the `UNIX` `PostScript` viewer `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.

2.4 Search paths

o In looking for `Asymptote` system files, `asy` will search the following paths, in the order listed:

1. current directory
2. `$ASYMPTOTE_DIR`

   A list of one or more directories (separated by `:` under UNIX and `;` under `MSDOS`).
3. system-wide directory (default: `/usr/share/asymptote` under `UNIX` and `c:\Program Files\Asymptote` under `MSDOS`).

2.5 Compiling from UNIX source

To compile and install a `UNIX` executable 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/gc6.6.tar.gz` in the current directory and

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

If you get errors from a broken `pdftex` installation, simply put

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

in the directory `doc` and repeat the command `make all`.

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`.

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).

2.6 Editing modes

Users of `emacs` can edit `Asymptote` code with the mode `asy-mode`, which is installed and enabled by default in the Debian package.

Fans of `vim` can customize `vim` for `Asymptote` with

```
cp @value{Datadir}/doc/asymptote/examples/asy.vim.gz ~/.vim/syntax/asy.vim.gz
gunzip ~/.vim/syntax/asy.vim.gz
```

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
```

2.7 CVS

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/gc6.6.tar.gz
./configure
make all
make install
```

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

2.8 Uninstall

To uninstall a `UNIX` binary distribution, type

```
tar -zxvf asymptote-x.xx.ARCH.tar.gz | xargs rm
texhash
```

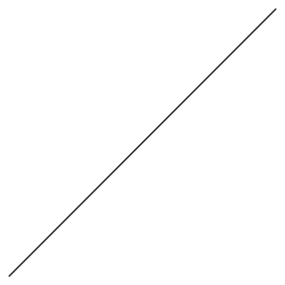
To uninstall all `Asymptote` files installed from a source distribution, use the command

```
make uninstall
```

3 Examples

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

```
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 97.

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 the command

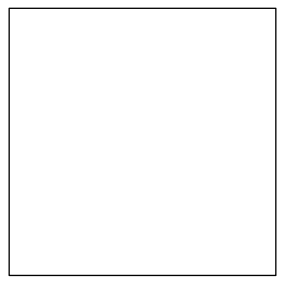
```
asy -V test
```

`MSDOS` users can drag and drop the file onto the Desktop `asy` icon or make `Asymptote` the default application for files with the extension `asy`.

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);
```



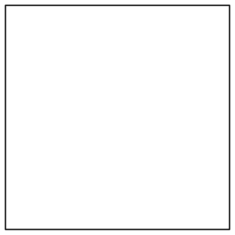
It is often inconvenient to work directly with `PostScript` coordinates. The next example draws a unit square scaled to width 101 bp and height 101 bp. The output is identical to that of the previous example.

```
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))`.

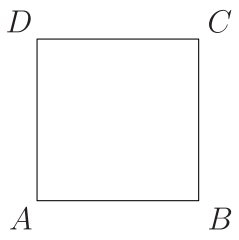
One can also specify the size in `pt` (1 `pt` = 1/72.27 `inch`), `cm`, `mm`, or `inches`. If 0 is given as a size argument, no restriction is made in that direction; the overall scaling will be determined by the other direction (see [size], page 25):

```
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 4.13.3 [graph], page 47 (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);
}
```

Loops, together with user-defined functions, are illustrated in the files `wheel.asy` and `cube.asy` in the animations subdirectory of the examples directory. These examples use 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 in the interval [`-pi`,`pi`];

`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`. A synonym for `length(triple)` is `abs(triple)`;

`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;

`real Longitude(triple v)`
: returns the longitude of `v` in degrees, or `0` if `v.x=v.y=0` rather than producing an error;

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

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

`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`;

`triple 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 4.13.18 [babel], page 81):

- \" 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 before, string after)`
: returns a string with all occurrences of the string `before` in the string `s` changed to the string `after`;

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

`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 4.10 [Functions], page 35).

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;
}
```

If high accuracy is needed, a true circle may be produced with this routine, defined in the module `graph.asy`:

```
guide Circle(pair c, real r, int ngraph=400);
```

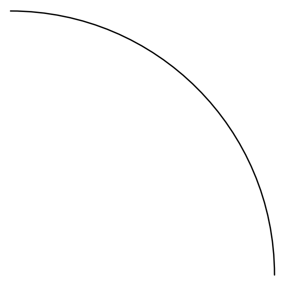
Each interior node of a cubic spline may be given a direction prefix or suffix `{dir}`: the direction of the pair `dir` specifies the direction of the incoming or outgoing tangent, respectively, to the curve at that node. Exterior nodes may be given direction specifiers only on their interior side. Cubic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

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

A good reference on Bezier curves and the algorithms that `Asymptote` uses to determine the control points is Donald Knuth's monograph, `The MetaFontbook`, chapters 3 and 14.

This example draws an approximate quarter circle:

```
size(100,0);
draw((1,0){up}..{left}(0,1));
```



A circular arc consistent with the above approximation 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);
```

If `r` < 0, the complementary arc of radius `|r|` is constructed. For convenience, an arc centered at `c` from pair `z1` to `z2` (assuming `|z2-c|=|z1-c|`) in the direction CCW (counter-clockwise) or CW (clockwise) may also be constructed with

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

If high accuracy is needed, a true arc may be produced with this routine, defined in the module `graph.asy`:

```
guide Arc(pair c, real r, real angle1, real angle2,
          int ngraph=400);
```

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

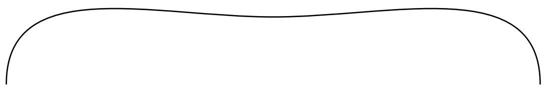
```
draw((0,0)..controls (0,100) and (100,100)..(100,0));
```

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));
```

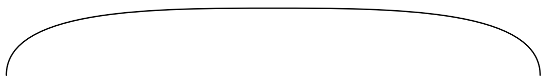
The `MetaPost ...` path connector, which requests, when possible, an inflection-free curve confined to a triangle defined by the endpoints and directions, is implemented in `Asymptote` as the convenient abbreviation `::` for `..tension atleast 1 ..` (the ellipsis `...` is used in `Asymptote` to indicate a variable number of arguments; see Section 4.10.3 [Rest arguments], page 37). For example, compare

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



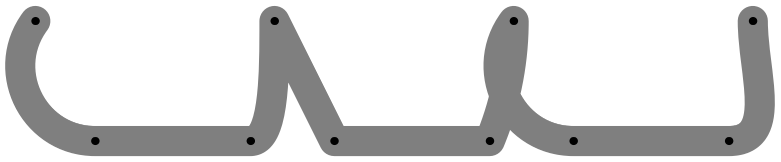
with

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



The `---` connector is an abbreviation for `..tension atleast infinity..` and the `&` connector concatenates two paths which meet at a common point. Here is an example of all five path connectors:

```
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));

dot(z);
```



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

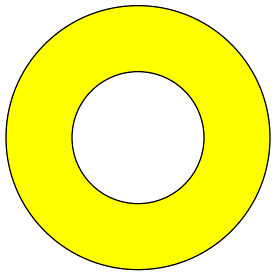
```
a..{up}nullpath..b;
c..nullpath{up}..d;
e..{up}nullpath{down}..f;
```

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);
```



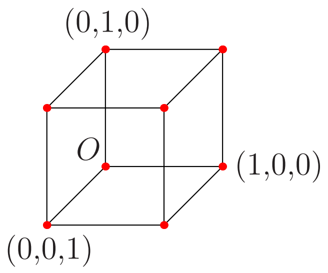
The `PostScript` even-odd fill rule here specifies that only the region bounded between the two unit circles is filled (see [fillrule], page 20). In this example, the same effect can be achieved by using the default zero winding number fill rule, if one is careful to alternate the orientation of the paths:

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

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

```
import three;

size(0,100);

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 t);`
: This returns the coordinates of the point between node `floor(t)` and `floor(t)+1` corresponding to the cubic spline parameter t =`t-floor(t)` (see [Bezier], page 12). If `t` 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 t);`
: This returns the direction of the tangent to the path at the point between node `floor(t)` and `floor(t)+1` corresponding to the cubic spline parameter t =`t-floor(t)` (see [Bezier], page 12). 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 precontrol(path, real t);`
: This returns the effective precontrol point at parameter `t`.

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

`pair postcontrol(path, real t);`
: This returns the effective postcontrol point at parameter `t`.

`real arclength(path);`
: This returns the length (in user coordinates) 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, real fuzz=0);`
: If `p` and `q` have at least one intersection point, return a pair of times representing the respective path times along `p` and `q`, in the sense of `point(path, real)`, for one such intersection point (as chosen by the algorithm described on page 137 of `The MetaFontbook`). Perform the computations to the absolute error specified by `fuzz`, or, if `fuzz` is 0, to machine precision. If the paths do not intersect, return the pair `(-1,-1)`.

`pair intersectionpoint(path p, path q, real fuzz=0);`
: This returns `point(p,intersect(p,q,fuzz).x)`, the actual point of intersection.

`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 min(path);`

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

`pair max(path);`

returns the pair(right,top) 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.

`bool inside(path g, pair z, pen p=currentpen);`

returns `true` iff the point `z` is inside the region bounded by the cyclic path `g` according to the fillrule of pen `p` (see [fillrule], page 20).

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 83). 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 rightmost pen will override those of the leftmost 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:

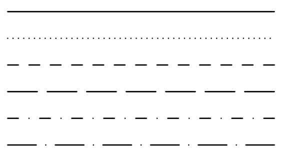
  ```
  Cyan,Magenta,Yellow,Black.
  ```

  The function `real[] colors(pen)` returns the color components of a pen. The functions `pen gray(pen)`, `pen rgb(pen)`, and `pen cmyk(pen)` return new pens obtained by converting their arguments to the respective color spaces.

- 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)`. For convenience, in `plain.asy` we define

  ```
  static void defaultpen(real w) {defaultpen(linewidth(w));}
  static pen operator +(pen p, real w) {return p+linewidth(w);}
  static pen operator +(real w, pen p) {return linewidth(w)+p;}
  ```

  so that one may set the linewidth like this:

  ```
  defaultpen(2);
  pen p=red+0.5;
  ```

- 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);
  pen zerowindingoverlap=fillrule(2);
  pen evenoddoverlap=fillrule(3);
  ```

  The fill rule, which identifies the algorithm used to determine the insideness of a path or array of paths, only affects the `clip`, `fill`, and `inside` functions. For the `zerowinding` fill rule, a point `z` is outside the region bounded by a path if the number of upward intersections of the path with the horizontal line `z--z+infinity` minus the number of downward intersections is zero. For the `evenodd` fill rule, `z` is considered to be outside the region if the total number of such intersections is even. A label is considered to be inside the region only if all four corners of its (possibly rotated) bounding box are within the region. The fill rules `zerowindingoverlap` and `evenoddoverlap` are respectively identical to `zerowinding` and `evenodd`, except that a label is considered to be inside the region whenever its center is within the region. While this allows labels to extend beyond the clipping region, any actual overlap is ignored when determining picture bounds. The default fill rule, `zerowinding`, may be changed with `defaultpen(pen)`.

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

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

  ```
  import fontsize;
  ```

  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:

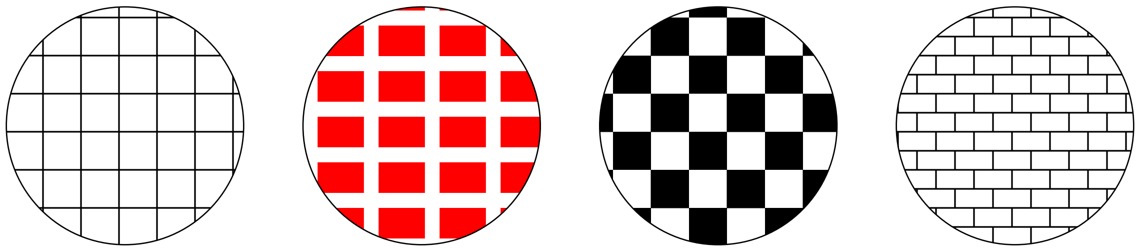
  ```
  pen AvantGarde(string series="m", string shape="n");
  pen Bookman(string series="m", string shape="n");
  pen Courier(string series="m", string shape="n");
  pen Helvetica(string series="m", string shape="n");
  pen NewCenturySchoolBook(string series="m", string shape="n");
  pen Palatino(string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

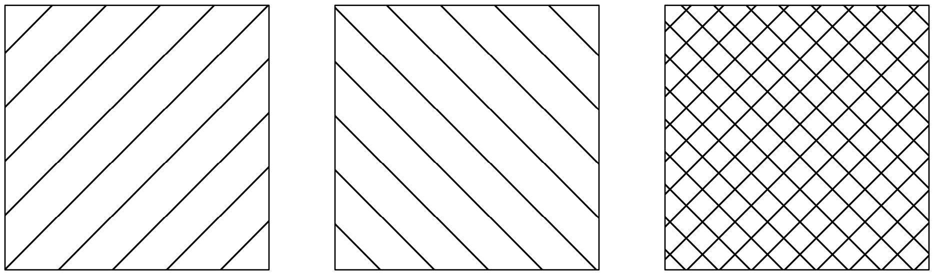
To `fill` or `draw` using pattern `name`, use the pen `pattern("name")`. For example, rectangular tilings can be constructed using the routines `picture tile(real Hx=5mm, real Hy=0, pen p=currentpen, filltype filltype=NoFill)`, `picture checker(real Hx=5mm, real Hy=0, pen p=currentpen)`, and `picture 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 85), 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;
axialshade(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.

The routine `defaultpen()` returns the current default pen attributes. Calling the routine `resetdefaultpen()` resets all pen default attributes to their initial values.

4.4 Transforms

`Asymptote` makes extensive use of affine transforms. A pair `(x,y)` is transformed by the transform `t=(t.x,t.y,t.xx,t.xy,t.yx,t.yy)` to `(x',y')`, where

```
x' = t.x + t.xx * x + t.xy * y
y' = t.y + t.yx * x + t.yy * y
```

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 12 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(real s);`
: scale by `s` in both x and y directions;

`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 a, pair b);`
: reflects about the line `a--b`.

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. The functions `min(frame f)` and `max(frame f)` return the (left,bottom) and (right,top) coordinates of the frame bounding box, respectively. The contents of frame `src` may be appended to frame `dest` with the command

```
void add(frame dest, frame src);
```

or prepended with

```
void prepend(frame dest, frame src);
```

A frame obtained by aligning frame `f` in the direction `dir`, in a manner analogous to the `align` argument of `label` (see Section 5.4 [label], page 86), is returned by

```
frame align(frame f, pair dir);
```

`picture` Pictures are high-level structures (see Section 4.7 [Structures], page 30) 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 x, real y,
          bool keepAspect=Aspect);
```

If the `x` and `y` sizes 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 `x` or `y` 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 `x` and the final height is no more than `y`.

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

To ensure that each dimension is no more than `size`, use the routine

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

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, bool quiet=false);
void shipout(string prefix=defaultfilename,
             orientation orientation=Portrait,
             string format="", bool wait=NoWait, bool quiet=false);
```

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

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

The default size and aspect ratio settings are those given to the `size` command (which default to `0`, `0`, and `true`, respectively).

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 margins around a picture, fit the picture to a frame using the function

```
frame bbox(picture pic=currentpicture, real xmargin=0,
           real ymargin=xmargin, pen p=currentpen,
           filltype filltype=NoFill);
```

Here `filltype` specifies one of the following fill types:

`Fill` Fill with the pen used to draw the boundary.

`Fill(pen p=nullpen)`
: If `p` is `nullpen`, fill with the pen used to draw the boundary; otherwise fill with pen `p`.

`NoFill` Do not fill; draw only the boundary.

`UnFill` Clip the region.

`UnFill` Clip the region and surrounding margins `xmargin` and `ymargin`.

For example, to draw a bounding box around a picture with a 0.25 cm margin and output the resulting frame, use the command:

```
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)`.

The function

```
pair point(picture pic=currentpicture, pair dir);
```

is a convenient way of determining the point on the boundary of the user-coordinate bounding box of `pic` in the direction `dir` relative to its center.

The member functions `pic.min()` and `pic.max()` calculate the `PostScript` bounds that picture `pic` would have if it were currently fit to a frame using its default size specification.

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,
         filltype filltype=NoFill, bool put=Above);
void add(picture dest, picture src, bool group=true,
         filltype filltype=NoFill, bool put=Above);
```

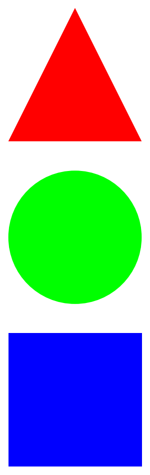
The first example adds `src` to `currentpicture`; the second one adds `src` to `dest`. The `group` option specifies whether or not the graphical user interface `xasy` should treat all of the elements of `src` as a single entity (see Chapter 9 [GUI], page 98), `filltype` requests optional background filling or clipping, and `put` specifies whether to add `src` above or below existing objects.

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,
         filltype filltype=NoFill, bool put=Above);
void add(pair origin, picture src, bool group=true,
         filltype filltype=NoFill, bool put=Above);
void add(pair origin=0, picture dest=currentpicture, frame src,
         bool group=true, filltype filltype=NoFill,
         bool put=Above);
void add(pair origin=0, picture dest=currentpicture, frame src,
         pair dir, bool group=true, filltype filltype=NoFill,
         bool put=Above);
```

The `dir` argument in the last form specifies a direction to use for aligning the frame, in a manner analogous to the `align` argument of `label` (see Section 5.4 [label], page 86). Illustrations of frame alignment can be found in the examples [errorbars], page 55 and [image], page 67. If you want to align 3 or more subpictures, group them two at a time:

```
picture pic1;
real size=50;
size(pic1,size);
fill(pic1,(0,0)--(50,100)--(100,0)--cycle,red);

picture pic2;
size(pic2,size);
fill(pic2,unitcircle,green);

picture pic3;
size(pic3,size);
fill(pic3,unitsquare,blue);

picture pic;
add(pic,pic1.fit(),N);
add(pic,pic2.fit(),10S);

add(pic.fit(),N);
add(pic3.fit(),10S);
```



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, picture dest=currentpicture,
           frame src, bool group=true,
           filltype filltype=NoFill, bool put=Above);
void attach(pair origin=0, picture dest=currentpicture, frame src,
            pair dir, bool group=true, filltype filltype=NoFill,
            bool put=Above);
```

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

```
guide box(frame f, Label L="", real xmargin=0,
          real ymargin=xmargin, pen p=currentpen,
          filltype filltype=NoFill, bool put=Above);
guide ellipse(frame f, Label L="", real xmargin=0,
              real ymargin=xmargin, pen p=currentpen,
              filltype filltype=NoFill, bool put=Above);
void box(picture pic=currentpicture, Label L,
         real xmargin=0, real ymargin=xmargin, pen p=currentpen,
         filltype filltype=NoFill, bool put=Above);
```

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`, `currentpen`, and `currentprojection`, use the function `save()`.

To restore a snapshot of `currentpicture`, `currentpen`, and `currentprojection`, 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, string commentchar="#")`; 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`. The string `commentchar` specifies a comment character (the default comment character `#` is actually determined by the variable `commentchar` in `plain.asy`). If this character is encountered in a data file, the remainder of the line is ignored. When reading strings, the comment character must be in the first column (otherwise it will treated as an ordinary character).

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.

A file named `name` can be open for output with

```
file output(string name, bool append=false);
```

data will be appended to an existing file only if the file is opened with `append=true`. Data of type `T` can be written to an output file by calling one of the following functions

```
write(string s="", T x, suffix e=endl);
write(file fout, string s="", T x, suffix e=none);
write(string s="" ... T[] x);
write(file fout, string s="" ... T[] x);
write(file fout=stdout, suffix e=endl);
```

If the `fout` is not specified, `stdout` is used and terminated with a newline. If specified, the optional identifying string `s` is written before the data `x`. Except when writing objects that may generate multiple lines of output (like arrays or guides), an arbitrary number of data values may be given. The suffix `e` may be one of the following: `none` (do nothing), `endl` (terminate with a newline), or `tab` (terminate with a tab). Here is a simple example of data output:

```
file fout=output("test.txt");
write(fout,1);                // Writes "1"
write(fout);                  // Writes a new line
write(fout,"List: ",1,2,3);   // Writes "List: 1     2     3"
```

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

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)`.

The routines

```
string getstring(string name, string default="", string prompt="",
                 string prefix=getstringprefix, bool save=true);
real getreal(string name, real default=0, string prompt="",
             string prefix=getstringprefix, bool save=true);
```

defined in `plain.asy` may be used to read a value from `stdin`. If `save=true`, the value is saved to the file named `prefix+name`, to provide the default value for subsequent runs. The default value (initially `default`) is displayed after `prompt`. The initial value of `getstringprefix` is ".asy_".

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. Any code at the top-level scope within the structure is executed on initialization.

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
```

In the following example, the static function `T.T(real x)` is a constructor that initializes and returns a new instance of `T`:

```
struct T {
  real x;
  static T T(real x) {T t=new T; t.x=x; return t;}
}

T operator init() {return new T;}

T a;
T b=T.T(1);

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

The name of the constructor need not be identical to the name of the structure; for example, see `triangle.SAS` in `geometry.asy`.

Structure assignment does a shallow copy; a deep copy requires writing an explicit `copy()` member. The function `bool alias(T,T)` checks to see if two instances of the structure `T` are identical. The boolean operators `==` and `!=` are by default equivalent to `alias` and `!alias` respectively, but may be overwritten for a particular type do a deep comparison.

When `a` is defined both as a variable and a type, the qualified name `a.b` refers to the variable instead of the type.

Much like in C++, casting (see Section 4.12 [Casts], page 43) provides for an elegant implementation of structure inheritance, including virtual functions:

```
struct parent {
  real x=1;
  public void virtual(int) {write (0);}
  void f() {virtual(1);}
}

parent operator init() {return new parent;}
```

```
void write(parent p) {write(p.x);}

struct child {
  parent parent;
  real y=2;
  void virtual(int x) {write (x);}
  parent.virtual=virtual;
  void f()=parent.f;
}

parent operator cast(child child) {return child.parent;}

child operator init() {return new child;}

parent p;
child c;

write(c);                       // Outputs 1;

p.f();                          // Outputs 0;
c.f();                          // Outputs 1;

write(c.parent.x);              // Outputs 1;
write(c.y);                     // Outputs 2;
```

Further examples of structures are `Legend` and `picture` in the default `Asymptote` base file `plain.asy`.

4.8 Operators

4.8.1 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 greatest integer less than or equal to `x/y`. 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. In particular, this makes `q*quotient(p,q)+p%q == p` for all integers `p` and nonzero integers `q`.
`^`	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;
```

4.8.2 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.

4.8.3 User-defined operators

The following symbols may be used with `operator` to define or redefine operators on structures and built-in types:

```
- + * / % ^ ! < > == != <= >= & && || ^^ .. :: -- --- ++
<< >> $ $$ @ @@
```

The operators on the second line have precedence one higher than the boolean operators `<`, `>`, `<=`, and `>=`.

Guide operators like `..` may be overloaded, say, to write a user function that produces a new guide from a given guide:

```
guide dots(...guide[] g)=operator ..;

guide operator ..(...guide[] g) {
  guide G;
  if(g.length > 0) {
    write(g[0]);
    G=g[0];
  }
  for(int i=1; i < g.length; ++i) {
    write(g[i]);
    write();
    G=dots(G,g[i]);
  }
  return G;
}

guide g=(0,0){up}..{SW}(100,100){NE}..{curl 3}(50,50)..(10,10);
write("g=",none);
write(g);
```

4.9 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
```

4.10 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 30).

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 based on 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).

4.10.1 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 4.12 [Casts], page 43) 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 by evaluating the given `Asymptote` expression in the scope where the called function is defined.

4.10.2 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.

4.10.3 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 (regardless of casting) 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
```

4.10.4 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 order `n` Bessel functions of the first kind `J(int n, real)` and second kind `Y(int n, real)`, as well as the gamma function `gamma`, the error function `erf`, and the complementary error function `erfc`. The standard real(real, real) functions `atan2`, `hypot`, `fmod`, `remainder` are also included.

The functions `floor`, `ceil`, 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). The functions `Floor`, `Ceil`, and `Round` are respectively similar, except that if the result cannot be converted to a valid int, they return `intMax()` for positive arguments and `-intMax()` for negative arguments, rather than generating an integer overflow. 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`.

4.11 Arrays

Appending `[]` to a built-in or user-defined type yields an array. By default, attempts to access or assign to an array element using a negative index generates an error. While reading an array element with an index beyond the length of the array also generates an error, assignment to such an element causes the array to be resized 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` of type `T[]` has the virtual members `int length`, `void cyclic(bool)`, `bool cyclicflag`, `T push(T)`, `void append(T[])`, and `T pop()`. The member `A.length` evaluates to the length of the array. Setting `A.cyclic(true)` signifies that array indices should be reduced modulo the current array length. Reading from or writing to a nonempty cyclic array never leads to out-of-bounds errors or array resizing. The member `A.cyclicflag` returns the current setting of the `cyclic` flag. The functions `A.push` and `A.append` append their arguments onto the end of the array (for convenience `A.push` also returns its argument), while `A.pop()` pops and returns the last element. Like all `Asymptote` functions, `cyclic`, `push`, `pop`, and `append` can be "pulled off" of the array and used on their own. For example,

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

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 `n`th `true` value or -1 if not found. If `n` is negative, search backwards from the end of the array for the `-n`th 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);
```

`real[] tridiagonal(real[] a, real[] b, real[] c, real[] f);`
: Solve the periodic tridiagonal problem L^{-1} `f`, where `f` is an n vector and L is the $n \times n$ matrix

```
[ b[0] c[0]           a[0]   ]
[ a[1] b[1] c[1]             ]
[      a[2] b[2] c[2]        ]
[                ...         ]
[       c[n-1] a[n-1] b[n-1] ]
```

For Dirichlet boundary conditions (denoted here by `u[-1]` and `u[n]`), replace `f[0]` by `f[0]-a[0]u[-1]` and `f[n-1]-c[n-1]u[n]`; then set `a[0]=c[n-1]=0`.

`real[] quadraticroots(real a, real b, real c);`
: This numerically robust solver returns the real roots of the quadratic equation `ax^2+bx+c=0`.

`real[] cubicroots(real a, real b, real c, real d);`
: This numerically robust solver returns the real roots of the cubic equation `ax^3+bx^2+cx+d=0`.

`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).

4.12 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`.

4.13 Import

While `Asymptote` provides many features by default, some applications require specialized features contained in external `Asymptote` modules. For instance, the lines

```
access graph;
graph.axes();
```

draw x and y axes on a two-dimensional graph. Here, the command looks up the module under the name `graph` in a global dictionary of modules and puts it in a new variable named `graph`. The module is a structure, and we can refer to its fields as we usually would with a structure.

Often, one wants to use module functions without having to specify the module name. The code

```
from graph access axes;
```

adds the `axes` field of `graph` into the local name space, so that subsequently, one can just write `axes()`. If the given name is overloaded, all types and variables of that name are added. To add more than one name, just use a comma-separated list:

```
from graph access axes, xaxis, yaxis;
```

Wild card notation can be used to add all non-private fields and types of a module to the local name space:

```
from graph access *;
```

Similarly, one can add the non-private fields and types of a structure to the local environment with the `unravel` keyword:

```
struct matrix {
  real a,b,c,d;
}

real det(matrix m) {
  unravel m;
  return a*d-b*c;
}
```

Alternatively, one can unravel selective fields:

```
real det(matrix m) {
  from m unravel a,b,c as C,d;
  return a*d-b*C;
}
```

The command

```
import graph;
```

is a convenient abbreviation for the commands

```
access graph;
unravel graph;
```

That is, `import graph` first loads a module into a structure called `graph` and then adds its non-private fields and types to the local environment. This way, if a member variable (or function) is overwritten with a local variable (or function of the same signature), the original one can still be accessed by qualifying it with the module name.

Wild card importing will work fine in most cases, but one does not usually know all of the internal types and variables of a module, which can also change as the module writer adds or changes features of the module. As such, it is prudent to add `import` commands at the start of an `Asymptote` file, so that imported names won't shadow locally defined functions. Still, imported names may shadow other imported names, depending on the order in which they were imported, and imported functions may cause overloading resolution problems if they have the same name as local functions defined later.

To rename modules or fields when adding them to the local environment, use `as`:

```
access graph as graph2d;
from graph access xaxis as xline, yaxis as yline;
```

The command

```
import graph as graph2d;
```

is a convenient abbreviation for the commands

```
access graph as graph2d;
unravel graph2d;
```

Currently, all modules are implemented as `Asymptote` files. When looking up a module that has not yet been loaded, `Asymptote` searches the standard search paths (see Section 2.4 [Search paths], page 3) for the matching file. The file corresponding to that name is read and the code within it is interpreted as the body of a structure defining the module.

If the file name contains nonalphanumeric characters, enclose it with quotation marks:

```
access "/usr/share/asymptote/graph.asy" as graph;
```

```
from "/usr/share/asymptote/graph.asy" access axes;
```

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

It is an error if modules import themselves (or each other in a cycle).

The module name to be imported must be known at compile time. However, you can execute an `Asymptote` file determined at runtime in a new `Asymptote` environment with the function

```
void execute(string s, bool embedded=false);
```

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

```
void eval(string s, bool embedded=false);
```

If `embedded` is `true`, the string `s` will be evaluated at the top level of the current environment instead of in an independent environment.

One can evaluate arbitrary `Asymptote` code (which may contain unescaped quotation marks) with the command

```
void eval(code s, bool embedded=false);
```

Here `code` is a special type used with `quote {}` to enclose `Asymptote code` like this:

```
real a=1;
code s=quote {
  write(a);
};
eval(s,true);        // Outputs 1
```

To include the contents of a file `graph` verbatim (as if the contents of the file were inserted at that point), use one of the forms:

```
include graph;
```

```
include "/usr/share/asymptote/graph.asy";
```

`Asymptote` currently ships with the following base modules:

4.13.1 plain

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

By default, an implicit private import plain; occurs before translating a file and before the first command given in interactive mode. This also applies when translating files for module definitions (except when translating plain, of course). This means that the types and functions defined in plain are accessible in almost all Asymptote code. Use the -noplain command-line option to disable this feature.

4.13.2 simplex

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

4.13.3 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 join=operator --);
  ```
  Returns a graph using the scaling information for picture pic (see [automatic scaling], page 57) of the function f on the interval [a,b], sampling at n evenly spaced points, with one of these interpolation types:
  - operator -- (linear interpolation; the abbreviation Straight is also accepted)
  - operator .. (piecewise Bezier cubic spline interpolation; the abbreviation Spline is also accepted)

-
  ```
  guide graph(picture pic=currentpicture, real x(real), real y(real),
              real a, real b, int n=ngraph,
              interpolate join=operator --);
  ```
  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 join=operator --);
  ```
  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 join=operator --);
  ```

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 join=operator --);
```

  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 join=operator --);
```

  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 join=operator --);
```

  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, Label L="", axis axis=YZero,
           real xmin=-infinity, real xmax=infinity, pen p=currentpen,
           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 Label `L`. The relative label location along the axis (a real number from [0,1]) defaults to 1 (see [Label], page 86), so that the label is drawn at the end of the axis. 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 (or y=1 on a logarithmic axis) 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 83). If `put=Below` and the `extend` flag for `axis` is `false`, the axis is drawn before any existing objects in the current picture.

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(Label format="", ticklabel ticklabel=null,
                bool beginlabel=true, bool endlabel=true,
                int N=0, int n=0, real Step=0, real step=0,
                bool begin=true, bool end=true,
                real Size=0, real size=0, bool extend=false,
                pen pTick=nullpen, pen ptick=nullpen);
```

If any of these parameters are omitted, reasonable defaults will be chosen:

`Label format`
: override the default tick label format (`defaultformat`, initially `"$%.4g$"`), rotation, pen, and alignment (for example, `LeftSide`, `Center`, or `RightSide`) relative to the axis. To enable `LaTeX` math mode fonts, the format string should begin and end with `$` see [format], page 11; if the format string is `"%"`, the tick label will be suppressed;

`ticklabel`
: is a function `string(real x)` returning the label (by default, format(format.s,x)) for each tick value `x`;

`bool beginlabel`
: include the first label;

`bool endlabel`
: include the last label;

`int N`
: when automatic scaling is enabled (the default; see [automatic scaling], page 57), divide the values evenly into this many intervals, separated by big ticks;

`int n`
: divide each value interval into this many subintervals, separated by small ticks;

`real Step`
: the tick value spacing between big ticks (if `N=0`);

`real step`
: the tick value spacing between small ticks (if `n=0`);

`bool begin`
: include the first big tick;

`bool end`
: include the last big tick;

`real Size`
: the size of the big ticks (in `PostScript` coordinates);

`real size` the size of the small ticks (in `PostScript` coordinates);

`bool extend;`
extend the big ticks across the graph (useful for drawing a grid on the graph);

`pen pTick` an optional pen used to draw the big ticks;

`pen ptick` an optional pen used to draw the small ticks.

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(Label format="", ticklabel ticklabel=null,
                bool beginlabel=true, bool endlabel=true,
                real[] Ticks, real[] ticks=new real[],
                real Size=0, real size=0, bool extend=false,
                pen pTick=nullpen, pen ptick=nullpen)
```

-

```
void yaxis(picture pic=currentpicture, Label L="", axis axis=XZero,
           real ymin=-infinity, real ymax=infinity, pen p=currentpen,
           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 Label `L`. The relative location of the label (a real number from [0,1]) defaults to 1 (see [Label], page 86). An infinite value of `ymin` or `ymax` specifies that the corresponding axis limit will be automatically determined from the picture limits. 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 (or x=1 on a logarithmic axis) 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 83). If `put`=`Below` and the `extend` flag for `axis` is `false`, the axis is drawn before any existing objects in the current picture.

-

For convenience, the functions

```
void xequals(picture pic=currentpicture, Label L="", real x,
             bool extend=false, real ymin=-infinity, real ymax=infinity,
             pen p=currentpen, ticks ticks=NoTicks, bool put=Above,
             arrowbar arrow=None);
```

and

```
void yequals(picture pic=currentpicture, Label L="", real y,
             bool extend=false, real xmin=-infinity, real xmax=infinity,
             pen p=currentpen, 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, Label L="", guide g,
          pen p=currentpen, ticks ticks, ticklocate locate,
          arrowbar arrow=None, int[] divisor=new int[],
          bool put=Above, bool opposite=false);
```

This routine can be used to draw on picture `pic` a general axis based on an arbitrary path `g`, using pen `p`. One can optionally label the axis with Label `L` and add an arrow `arrow`. The tick type is given by `ticks`. The optional 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 a primary axis). The axis is drawn on top of any existing objects in the current picture only if `put` is `Above`. The tick locater `ticklocate` is constructed by the routine

```
ticklocate ticklocate(real a, real b, autoscaleT S=defaultS,
                      real tickmin=-infinity, real tickmax=infinity,
                      real time(real)=null, pair dir(real)=zero);
```

where `a` and `b` specify the respective tick values at `point(g,0)` and `point(g,length(g))`, `S` specifies the autoscaling transformation, the function `real time(real v)` returns the time corresponding to the value `v`, and `pair dir(real t)` returns the absolute tick direction as a function of `t` (zero means draw the tick perpendicular to the axis).

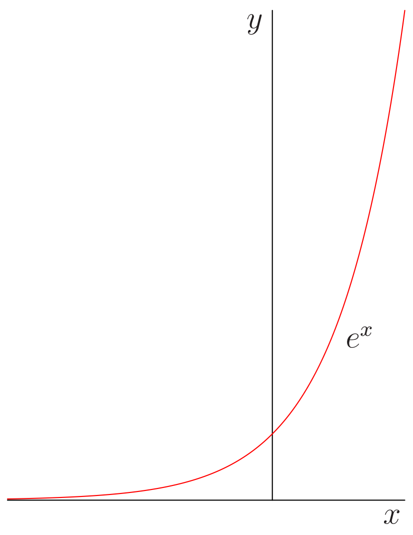
- These routines are useful for manually putting ticks and labels on axes (if the special variable `Label` is given as the `Label` argument, the `format` argument will be used to format a string based on the tick location):

```
void xtick(picture pic=currentpicture, Label L="", pair z,
           pair dir=N, string format="",
           real size=Ticksize, pen p=currentpen);
void ytick(picture pic=currentpicture, Label L="", explicit pair z,
           pair dir=E, string format="",
           real size=Ticksize, pen p=currentpen);
void ytick(picture pic=currentpicture, Label L="", real y,
           pair dir=E, string format="",
           real size=Ticksize, pen p=currentpen);
void tick(picture pic=currentpicture, pair z,
          pair dir, real size=Ticksize, pen p=currentpen);
void labelx(picture pic=currentpicture, Label L="", pair z,
            align align=S, string format="", pen p=nullpen);
void labelx(picture pic=currentpicture, Label L,
            string format="", explicit pen p=currentpen);
void labely(picture pic=currentpicture, Label L="", explicit pair z,
            align align=W, string format="", pen p=nullpen);
void labely(picture pic=currentpicture, Label L="", real y,
            align align=W, string format="", pen p=nullpen);
void labely(picture pic=currentpicture, Label L,
            string format="", explicit pen p=nullpen);
```

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 $y = 0$:

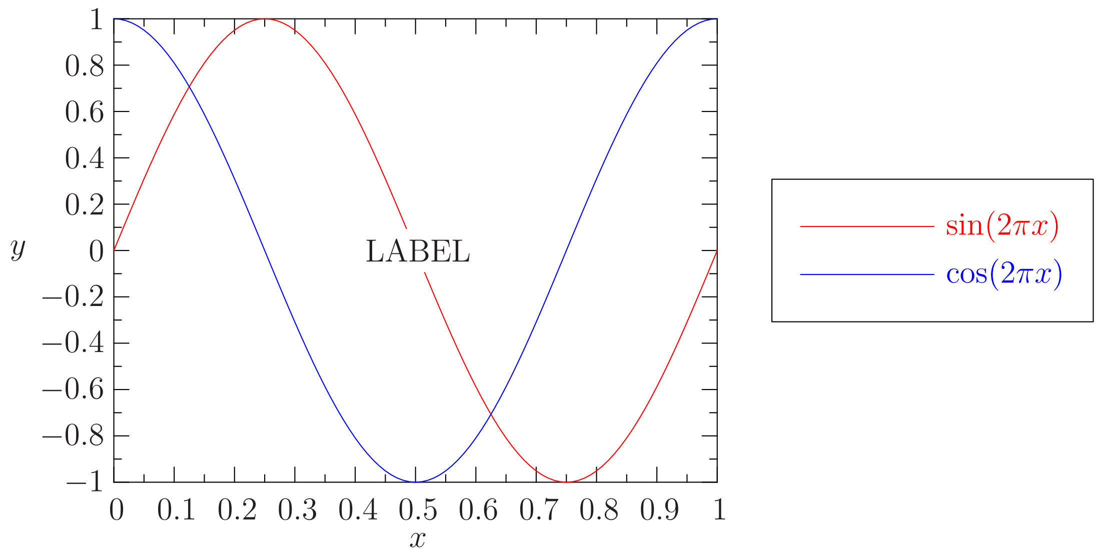
```
import graph;
size(150,0);

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

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

draw(graph(f,-4,2,operator ..),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 be adjusted either explicitly or by using the graphical user interface `xasy` (see

Chapter 9 [GUI], page 98). If an `UnFill(real xmargin=0, real ymargin=xmargin)` or `Fill(pen)` option is specified to `add`, the legend will obscure any underlying objects. Here we illustrate how to clip the portion of the picture covered by a label:

```
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);

label("LABEL",point(0),UnFill(1mm));

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

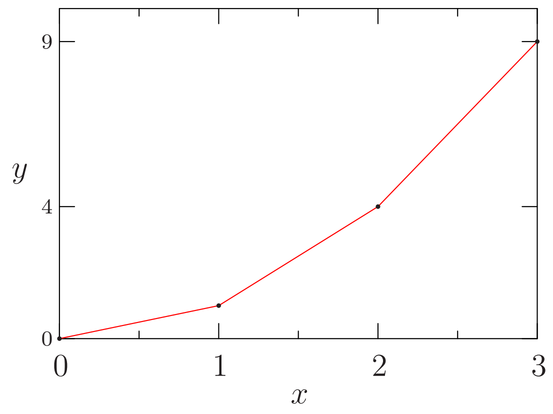


To specify a fixed size for the graph proper, use `attach`.

```
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);

label("LABEL",point(0),UnFill(1mm));

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

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("$y$",LeftRight,RightTicks(Label(fontsize(8)),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 markers, constructed with

```
marker marker(path g, markroutine markroutine=marknodes,
              pen p=currentpen, filltype filltype=NoFill,
              bool put=Above);
```

using the path `unitcircle` (see [filltype], page 26), are drawn below each node. Any frame can be converted to a marker, using

```
marker marker(frame f, markroutine markroutine=marknodes,
              bool put=Above);
```

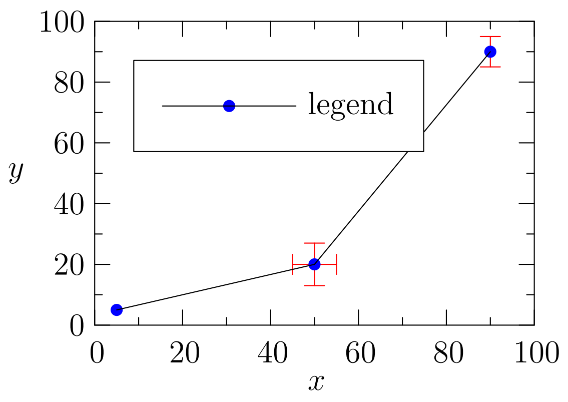
In the right-hand graph, the unit n-sided regular polygon `polygon(int n)` and the unit n-point cross `cross(int n)` are used to build a custom marker frame. Here `markuniform(int n)` adds this frame at `n` uniformly spaced points along the arclength of the path. This example also illustrates the `errorbar` routines:

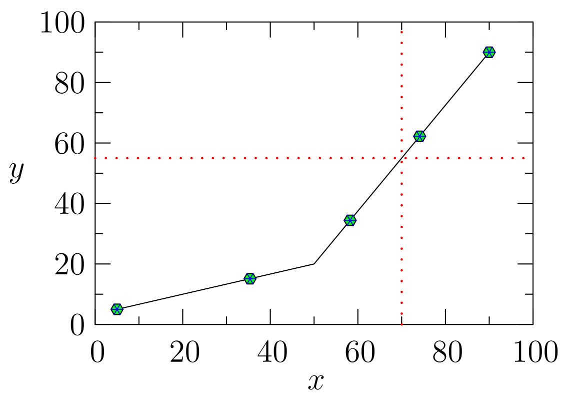
```
void errorbars(picture pic=currentpicture, pair[] z, pair[] dp,
               pair[] dm={}, bool[] cond={}, pen p=currentpen,
               real size=0);

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),"legend",
     marker(scale(0.8mm)*unitcircle,blue,Fill,Below));

xaxis(pic,"$x$",BottomTop,LeftTicks);
yaxis(pic,"$y$",LeftRight,RightTicks);
add(point(pic,NW),pic,legend(pic,20SE),UnFill);

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(pic2,f),marker(mark,markuniform(5)));

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

yequals(pic2,55.0,red+Dotted);
xequals(pic2,70.0,red+Dotted);

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

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





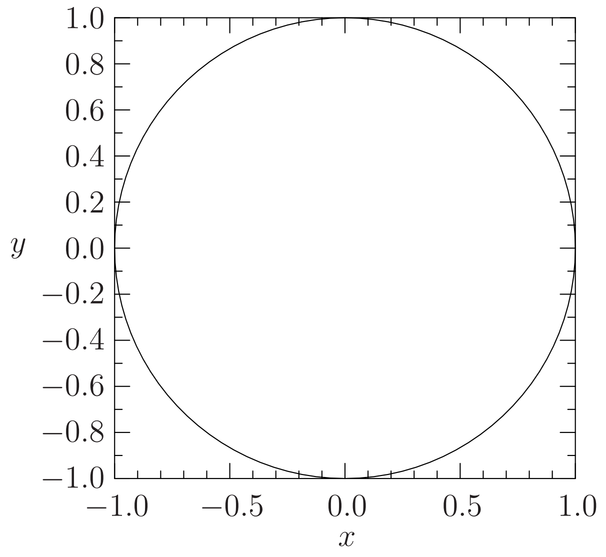
5. This example draws a graph of a parametrized curve.

   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 x(real t) {return cos(2pi*t);}
   real y(real t) {return sin(2pi*t);}

   draw(graph(x,y,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);
```

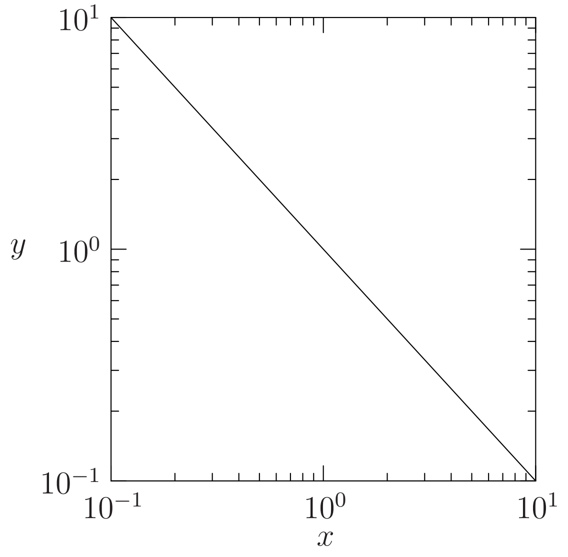
This sets the scalings for picture `pic`. The `graph` routines accept an optional `picture` argument for determining the appropriate scalings to use; if none is given, it uses those set for `currentpicture`. All path coordinates (and any call to `limits`, etc.) refer to scaled data. Two frequently used 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)`:

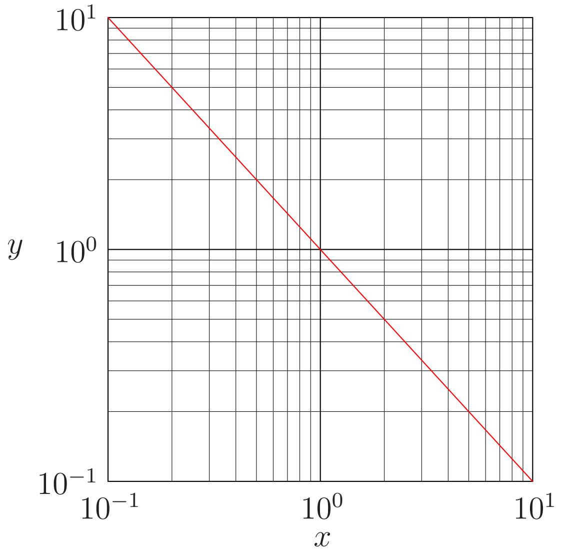
```
import graph;

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);
```



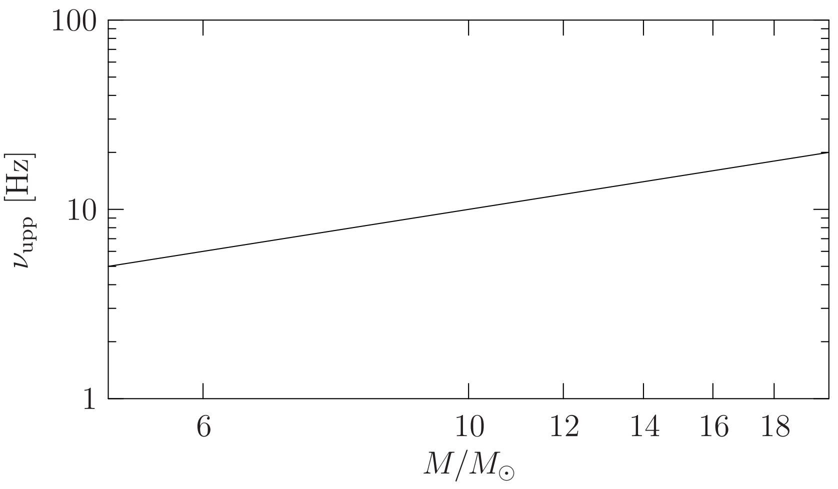
By extending the ticks, one can easily produce a logarithmic grid:

```
import graph;
size(200,200,IgnoreAspect);

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

scale(Log,Log);
draw(graph(f,0.1,10),red);
pen thin=linewidth(0.5*linewidth());
xaxis("$x$",BottomTop,LeftTicks(begin=false,end=false,extend=true,
                                ptick=thin));
yaxis("$y$",LeftRight,RightTicks(begin=false,end=false,extend=true,
                ptick=thin));
```



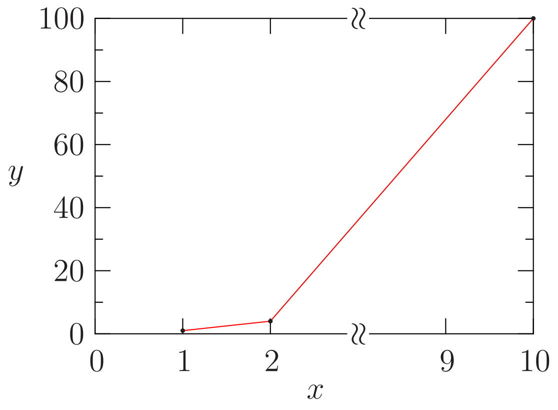
One can also specify custom tick locations and formats for logarithmic axes:

```
import graph;

size(300,175,IgnoreAspect);
scale(Log,Log);
draw(graph(identity,5,20));
xlimits(5,20);
ylimits(1,100);
xaxis("$M/M_\odot$",BottomTop,LeftTicks(new real[] {6,10,12,14,16,18}));
yaxis("$\nu_{\rm upp}$ [Hz]",LeftRight,
      RightTicks(new string(real x){return format(pow10(x));}));
```



Here is an example of a "broken" linear x axis that omits the segment [3,8]:

```
import graph;

size(200,150,IgnoreAspect);

// Break the axis at 3; restart at 8.
real a=3, b=8;

scale(Broken(a,b),Linear);

real[] x={1,2,10};
real[] y=x^2;

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

xaxis("$x$",BottomTop,LeftTicks(new real[]{0,1,2,9,10}));
yaxis("$y$",LeftRight,RightTicks);

label(rotate(90)*Break,(a,currentpicture.userMin.y));
label(rotate(90)*Break,(a,currentpicture.userMax.y));
```

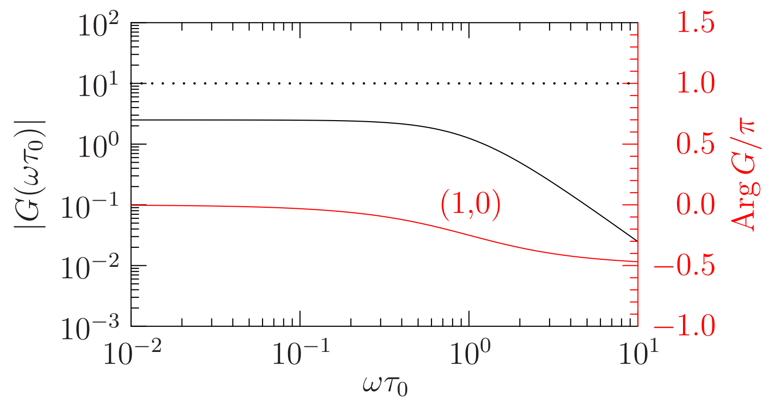


6. `Asymptote` can draw secondary axes with the routines

```
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:

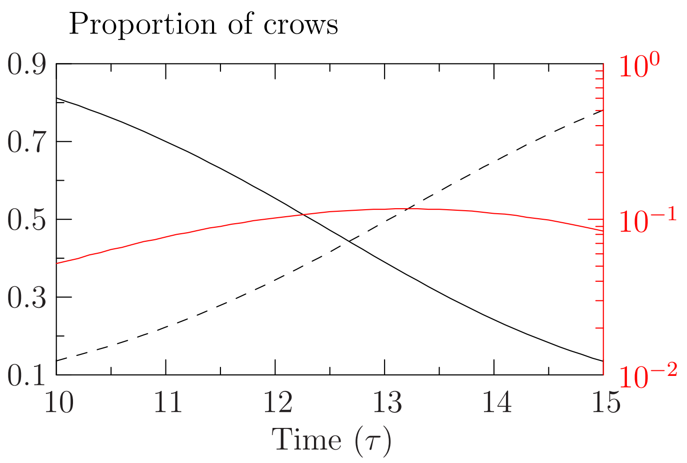
```
import graph;
texpreamble("\def\Arg{\mathop {\rm Arg}\nolimits}");

size(10cm,5cm,IgnoreAspect);

real ampl(real x) {return 2.5/(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 pic) {
                       scale(pic,Log,Linear);
                       draw(pic,graph(pic,phas,0.01,10),red);
                       ylimits(pic,-1.0,1.5);
                       yaxis(pic,"$\Arg G/\pi$",Right,red,
                             LeftTicks("$% #.1f$",
                                       begin=false,end=false));
                       yequals(pic,1,Dotted);
                     });
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);
```

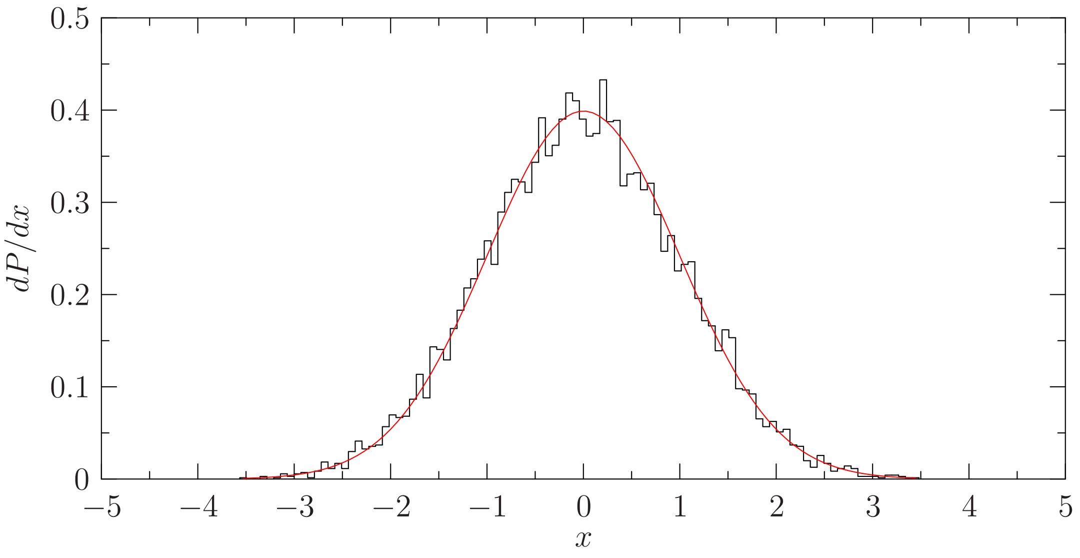
```
real[] t=a[0], susceptible=a[1], infectious=a[2], dead=a[3], larvae=a[4];
real[] susceptibleM=a[5], exposed=a[6],infectiousM=a[7];

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,Right,red,LeftTicks(begin=false,end=false));
});

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



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);
```

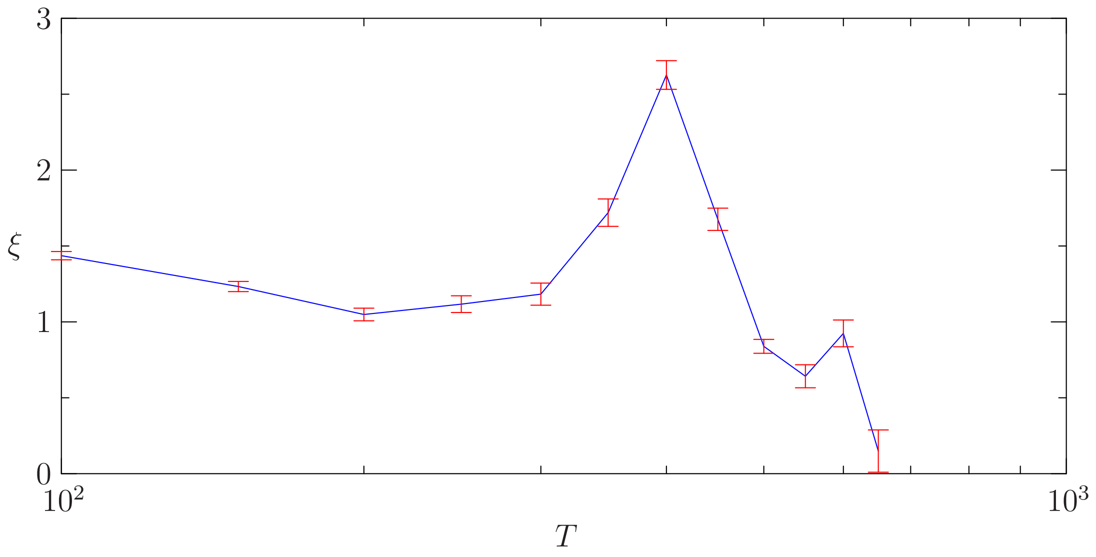
```
freq /= (dx*sum(freq));
histogram(x,freq);

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

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



8. Here is an example of reading column data in from a file and a least-squares fit, using the `stats` module.

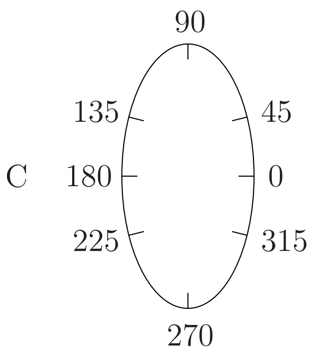
```
size(400,200,IgnoreAspect);

import graph;
import stats;

file fin=line(input("leastsquares.dat"));

real[][] a=dimension(fin,0,0);
a=transpose(a);

real[] t=a[0], rho=a[1];

// Read in parameters from the keyboard:
//real first=getreal("first");
//real step=getreal("step");
//real last=getreal("last");

real first=100;
real step=50;
real last=700;

// Remove negative or zero values of rho:
t=rho > 0 ? t : null;
rho=rho > 0 ? rho : null;

scale(Log,Linear);

int n=step > 0 ? ceil((last-first)/step) : 0;

real[] T,xi,dxi;

for(int i=0; i <= n; ++i) {
  real first=first+i*step;
  real[] logrho=(t >= first && t <= last) ? log(rho) : null;
  real[] logt=(t >= first && t <= last) ? -log(t) : null;

  if(logt.length < 2) break;

  // Fit to the line logt=L.m*logrho+L.b:
  linefit L=leastsquares(logt,logrho);

  T.push(first);
  xi.push(L.m);
  dxi.push(L.dm);
}

draw(graph(T,xi),blue);
errorbars(T,xi,dxi,red);

ylimits(0);

xaxis("$T$",BottomTop,LeftTicks);
```

```
yaxis("$\xi$",LeftRight,RightTicks);
```



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

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

guide g=ellipse((0,0),1,2);
axis(Label("C",align=10W),g,LeftTicks(endlabel=false,8,end=false),
     ticklocate(0,360,new real(real v) {
                path h=(0,0)--max(abs(max(g)),abs(min(g)))*dir(v);
                return intersect(g,h).x;}));
```

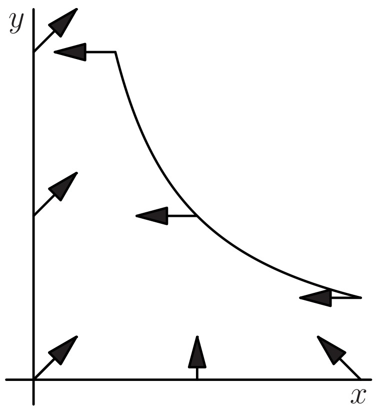


10. To draw a vector field along a path, first define a routine that returns a path as a function of a relative position parameter from [0,1] and use

```
typedef path vector(real);

void vectorfield(picture pic=currentpicture, path g, int n,
                 vector vector, real arrowsize=0, real arrowlength=0,
                  pen p=currentpen);
```

Here is a simple example of a flow field:

```
import graph;
defaultpen(1.0);

size(0,150,IgnoreAspect);

real arrowsize=4mm;
real arrowlength=2arrowsize;

// Return a vector interpolated linearly between a and b.
vector vector(pair a, pair b) {
  return new path(real x) {
    return (0,0)--arrowlength*interp(a,b,x);
  };
}

real alpha=1;
real f(real x) {return alpha/x;}

real epsilon=0.5;
path p=graph(f,epsilon,1/epsilon);

int n=2;
draw(p);
xaxis("$x$");
yaxis("$y$");

vectorfield(p,n,vector(W,W),arrowsize);
vectorfield((0,0)--(currentpicture.userMax.x,0),n,vector(NE,NW),
            arrowsize);
vectorfield((0,0)--(0,currentpicture.userMax.y),n,vector(NE,NE),
            arrowsize);
```



11. `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, Label L, pen p=currentpen,
                    paletteticks ticks=PaletteTicks)
    ```

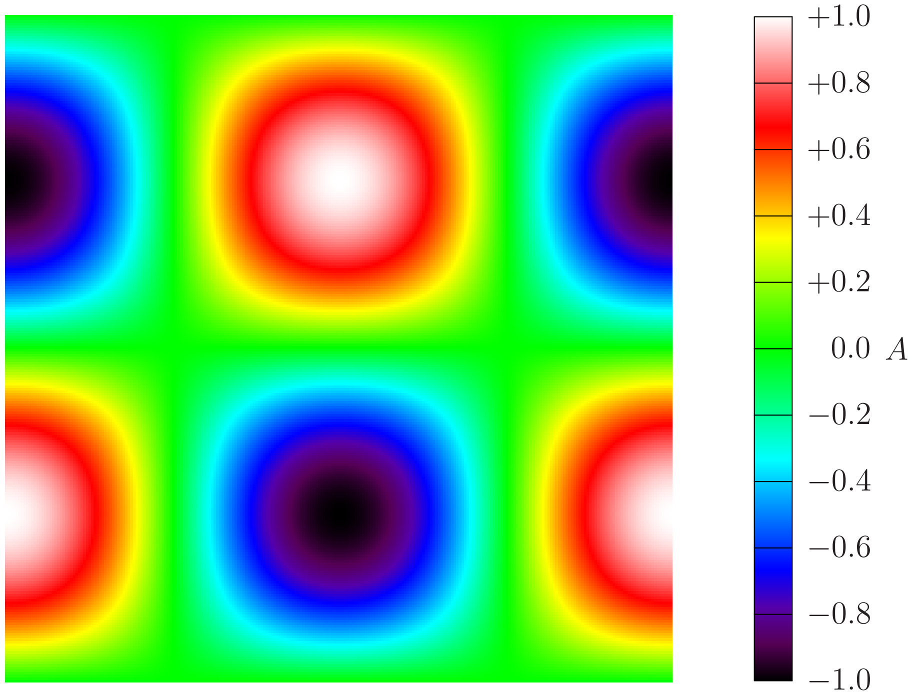
    The argument `paletteticks` is a special tick type (see [ticks], page 49) that takes the following arguments:

    ```
    paletteticks PaletteTicks(int N=0, real Step=0,
                              bool beginlabel=true, bool endlabel=true,
                              Label format="", pen pTick=nullpen);
    ```

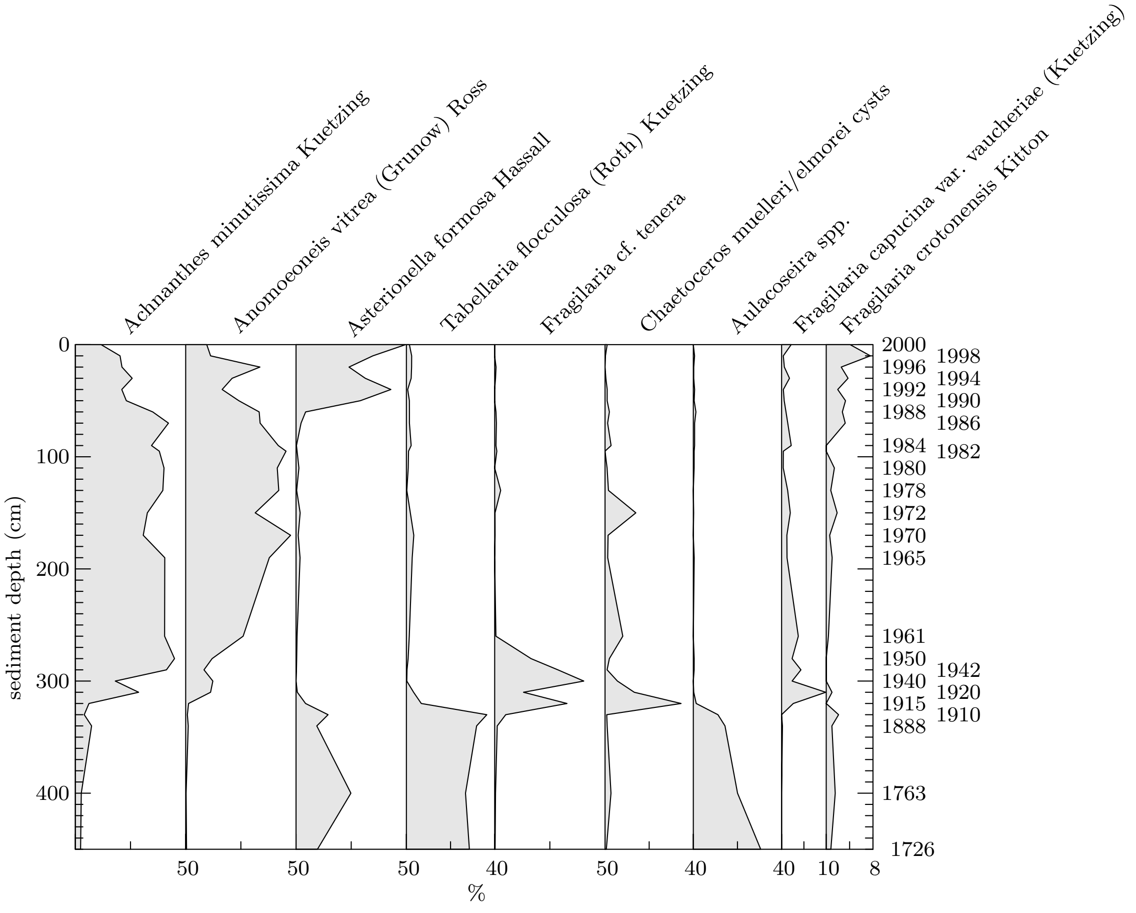
    The image and palette bar can be fit (and optionally aligned) to a frame and added to picture `dest` at the location `origin` using `add(pair origin=0, picture dest=currentpicture, frame)`:

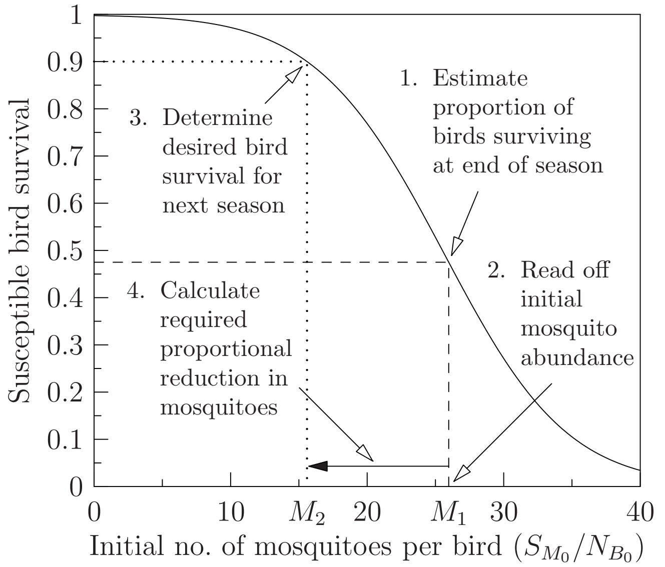
    ```
    import graph;
    import palette;

    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);
```



12. 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`.


Achnanthes minutissima Kuetzing
Anomoeoneis vitrea (Grunow) Ross
Asterionella formosa Hassall
Tabellaria flocculosa (Roth) Kuetzing
Fragilaria cf. tenera
Chaetoceros muelleri/elmorei cysts
Aulacoseira spp.
Fragilaria capucina var. vaucheriae (Kuetzing)
Fragilaria crotonensis Kitton
sediment depth (cm)
0
100
200
300
400
2000
1998
1996
1994
1992
1990
1988
1986
1984
1982
1980
1978
1972
1970
1965
1961
1950
1942
1940
1920
1915
1910
1888
1763
1726
50
50
50
40
50
40
40
10
8
%

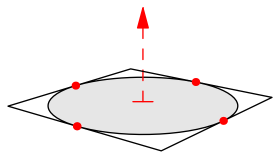

Susceptible bird survival
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1. Estimate proportion of birds surviving at end of season
2. Read off initial mosquito abundance
3. Determine desired bird survival for next season
4. Calculate required proportional reduction in mosquitoes
0
10
M_2
20
M_1
30
40
Initial no. of mosquitoes per bird (S_{M_0}/N_{B_0})

4.13.4 `three`

This module fully extends the notion of guides and paths in `Asymptote` to three dimensions, introducing the new types `guide3` and `path3`, along with a three-dimensional cycle specifier `cycle3`, tension operator `tension3`, and curl operator `curl3`. Just as in two dimensions, the nodes within a `guide3` can be qualified with these operators and also with explicit directions and control points (using braces and `controls`, respectively). This generalization of John Hobby's spline algorithm is shape-invariant under three-dimensional rotation, scaling, and shifting, and reduces in the planar case to the two-dimensional algorithm used in `Asymptote`, `MetaPost`, and `MetaFont`.

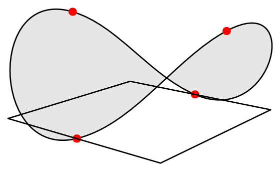
For example, a unit circle in the XY plane may be filled and drawn like this:

```
import three;
size(100,0);
guide3 g=(1,0,0)..(0,1,0)..(-1,0,0)..(0,-1,0)..cycle3;
filldraw(g,lightgrey);
draw(O--Z,red+dashed,BeginBar,Arrow);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle3));
dot(g,red);
```



and then distorted into a saddle:

```
import three;
size(100,0);
guide3 g=(1,0,0)..(0,1,1)..(-1,0,0)..(0,-1,1)..cycle3;
filldraw(g,lightgrey);
dot(g,red);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle3));
```



Here `O` is the triple `(0,0,0)` and `X`, `Y`, and `Z` are the triples `(1,0,0)`, `(0,1,0)`, and `(0,0,1)`, respectively. A general circle can be drawn perpendicular to the direction `normal` with the routine

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

A circular arc centered at `c` with radius `r` from `c+r*dir(theta1,phi1)` to `c+r*dir(theta2,phi2)`, drawing counterclockwise relative to the normal vector `cross(dir(theta1,phi1),dir(theta2,phi2))` if `theta2 > theta1` or if `theta2 == theta1` and `phi2 >= phi1`, can be constructed with

```
path3 arc(triple c, real r, real theta1, real phi1, real theta2, real phi2,
          triple normal=O);
```

The normal must be explicitly specified if `c` and the endpoints are colinear. If `r` < 0, the complementary arc of radius `|r|` is constructed. For convenience, an arc centered at `c` from triple `v1` to `v2` (assuming `|v2-c|=|v1-c|`) in the direction CCW (counter-clockwise) or CW (clockwise) may also be constructed with

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

When high accuracy is needed, the routines `Circle` and `Arc` defined in `graph3` may be used instead. See [surface], page 78 for an example of a three-dimensional circular arc.

A representation of the plane passing through point O with normal cross(u,v) is given by

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

A three-dimensional box with opposite vertices at triples `v1` and `v2` may be drawn with the function

```
guide3[] box(triple v1, triple v2);
```

For example, a unit cube is predefined as

```
guide3[] unitcube=box((0,0,0),(1,1,1));
```

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)`
`orthographic(real x, real y, real z)`
: This projects three dimensions onto two using the view seen at the location `camera` or `(x,y,z)`, respectively. Parallel lines are projected to parallel lines.

`perspective(triple camera)`
`perspective(real x, real y, real z)`
: These project three dimensions onto two taking account of perspective, as seen from the location `camera` or `(x,y,z)`, respectively.

The default projection, `currentprojection`, is initially set to `perspective(5,4,2)`.

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;

`scale3(real s)`
: scales by `s` in the x, y, and z directions;

`rotate(real angle, triple v)`
: rotates by `angle` in degrees about an axis `v` through the origin;

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

`reflect(triple u, triple v, triple w)`
: reflects about the plane through `u`, `v`, and `w`.

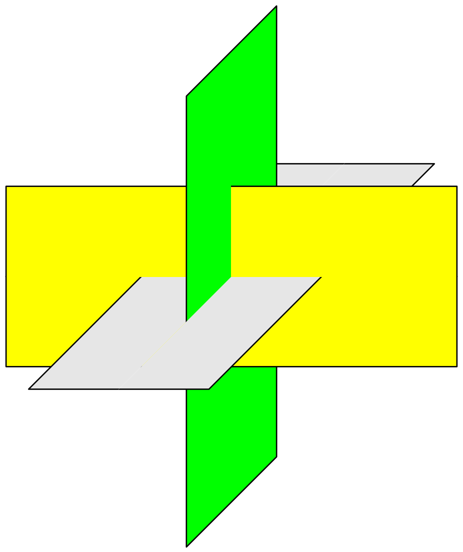
Three-dimensional versions of the path functions `length`, `size`, `point`, `dir`, `precontrol`, `postcontrol`, `arclength`, `arctime`, `reverse`, `subpath`, `intersect`, `intersectionpoint`, `min`, `max`, `cyclic`, and `straight` are also defined in the module `three`.

Planar hidden surface removal is implemented with a binary space partition and picture clipping. A planar path is first converted to a struct `face` derived from `picture`. A `face` may be given to a drawing routine in place of any `picture` argument. An array of such faces may then be drawn, removing hidden surfaces:

```
void add(picture pic=currentpicture, face[] faces,
         projection P=currentprojection);
```

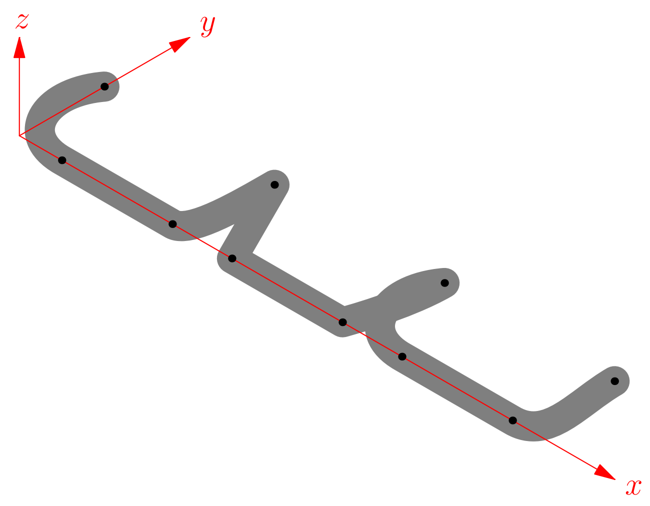
Here is an example showing three orthogonal intersecting planes:

```
size(6cm,0);
import math;
import three;

real u=2.5;
real v=1;

currentprojection=oblique;

path3 y=plane((2u,0,0),(0,2v,0),(-u,-v,0));
path3 l=rotate(90,Z)*rotate(90,Y)*y;
path3 g=rotate(90,X)*rotate(90,Y)*y;

face[] faces;
filldraw(faces.push(y),y,yellow);
filldraw(faces.push(l),l,lightgrey);
filldraw(faces.push(g),g,green);

add(faces);
```



Here is an example showing all five three-dimensional path connectors:

```
import graph3;

size(0,175);

currentprojection=orthographic(500,-500,500);

triple[] z=new triple[10];

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

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

path3 p=z[0]..z[1]---z[2]::{Y}z[3]
     &z[3]..z[4]--z[5]::{Y}z[6]
     &z[6]::z[7]---z[8]..{Y}z[9];

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

bbox3 b=limits(O,(700,200,100));

xaxis(Label("$x$",1),b,red,Arrow);
yaxis(Label("$y$",1),b,red,Arrow);
zaxis(Label("$z$",1),b,red,Arrow);

dot(z);
```



A three-dimensional bounding box structure is returned by calling `bbox3(triple min, triple max)` with the opposite corners `min` and `max`. This can be used to adjust the aspect ratio (see the example helix.asy):

```
void aspect(picture pic=currentpicture, bbox3 b,
            real x=0, real y=0, real z=0);
```

Further three-dimensional examples are provided in the files `near_earth.asy`, `conicurv.asy`, and (in the animations subdirectory) `cube.asy`.

4.13.5 graph3

This module implements three-dimensional versions of the functions in `graph.asy`. They work much like their two-dimensional counterparts, except that the user has to keep track of the three-dimensional axes limits (which in two dimension are stored in the picture) in a `bbox3` bounding box. The function

```
bbox3 autolimits(picture pic=currentpicture, triple min, triple max);
```

can be used to determine "nice" values for the bounding box corners. A user-space bounding box that takes into account of the axes scalings for picture `pic` is returned by

```
bbox3 limits(picture pic=currentpicture, triple min, triple max);
```

To crop a bounding box to a given interval use:

```
void xlimits(bbox3 b, real min, real max);
void ylimits(bbox3 b, real min, real max);
void zlimits(bbox3 b, real min, real max);
void limits(bbox3 b, triple min, triple max);
```

To draw an x axis in three dimensions from triple `min` to triple `max` with ticks in the direction `dir`, use the routine

```
void xaxis(picture pic=currentpicture, Label L="", triple min, triple max,
           pen p=currentpen, ticks ticks=NoTicks, triple dir=Y,
           arrowbar arrow=None, bool put=Above,
           projection P=currentprojection, bool opposite=false);
```

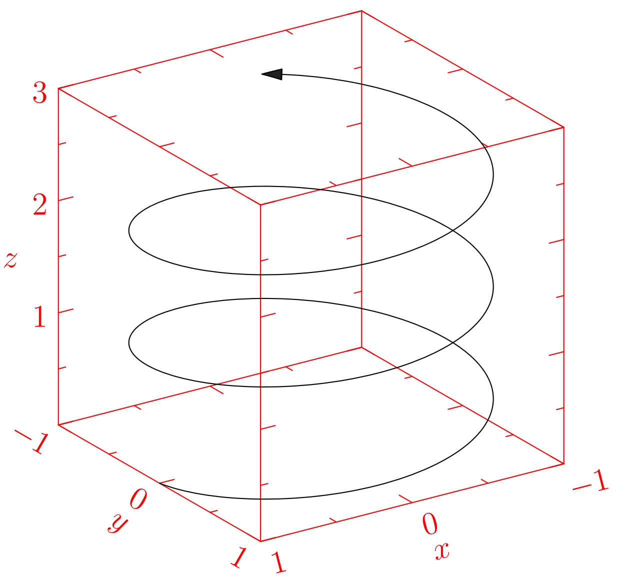
To draw an x axis in three dimensions using `bbox3 b` with ticks in the direction `dir`, use the routine

```
void xaxis(picture pic=currentpicture, Label L="", bool all=false,
           bbox3 b, pen p=currentpen, ticks ticks=NoTicks,
           triple dir=Y, arrowbar arrow=None, bool put=Above,
           projection P=currentprojection);
```

If `all=true`, also draw opposing edges of the three-dimensional bounding box. Analogous routines can be used to draw `y` and `z` axes in three dimensions.

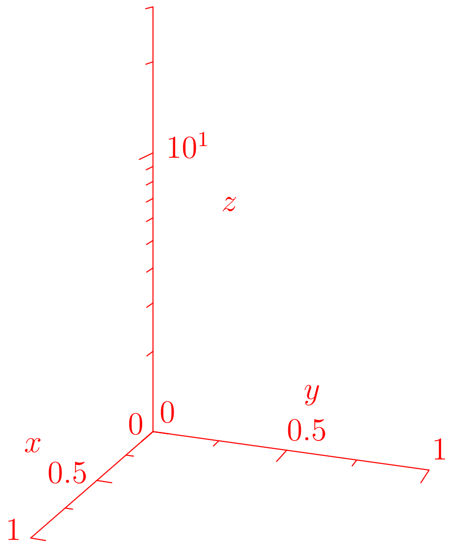
Here is an example of a helix and bounding box axes with rotated tick and axis labels, using orthographic projection:

```
import graph3;

size(0,200);

currentprojection=orthographic(4,6,3);

real x(real t) {return cos(2pi*t);}
real y(real t) {return sin(2pi*t);}
real z(real t) {return t;}

defaultpen(overwrite(SuppressQuiet));

path3 p=graph(x,y,z,0,2.7,Spline);
bbox3 b=autolimits(min(p),max(p));
aspect(b,1,1,1);

xaxis(rotate(X)*"$x$",all=true,b,red,RightTicks(rotate(X)*Label,2,2));
yaxis(rotate(Y)*"$y$",all=true,b,red,RightTicks(rotate(Y)*Label,2,2));
zaxis("$z$",all=true,b,red,RightTicks);

draw(p,Arrow);
```



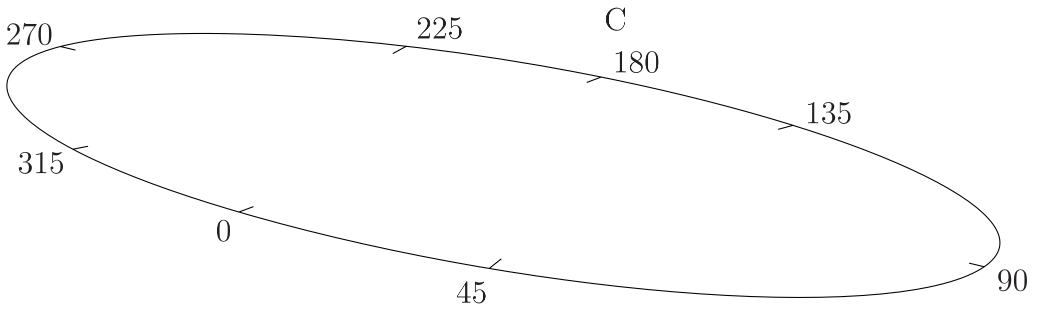
The next example illustrates three-dimensional x, y, and z axes, with autoscaling of the upper z limit disabled:

```
import three;
import graph;
import graph3;

size(0,200,IgnoreAspect);

currentprojection=perspective(5,2,2);

defaultpen(overwrite(SuppressQuiet));

scale(Linear,Linear,Log(automax=false));

bbox3 b=autolimits(Z,X+Y+30Z);

xaxis("$x$",b,red,RightTicks(2,2));
yaxis("$y$",b,red,RightTicks(2,2));
zaxis("$z$",b,red,RightTicks);
```



One can also place ticks along a general three-dimensional axis:

```
import graph3;

size(0,100);

path3 G=xscale3(1)*(yscale3(2)*unitcircle3);

axis(Label("C",align=Relative(5E)),G,
     LeftTicks(endlabel=false,8,end=false),
     ticklocate(0,360,new real(real v) {
                path g=G;
                path h=O--max(abs(max(G)),abs(min(G)))*dir(90,v);
                return intersect(g,h).x;
              },perpendicular(G,Z)));
```

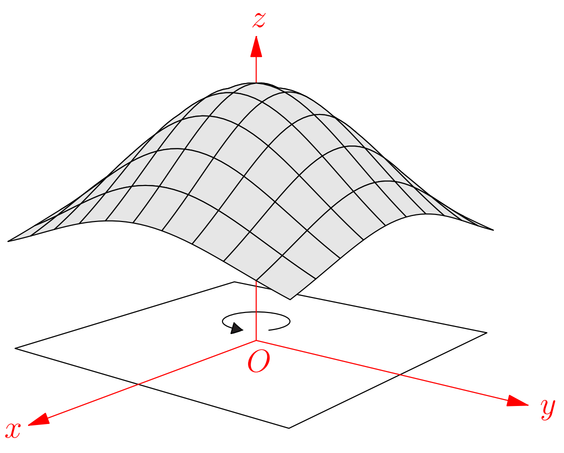


Surface plots of functions and matrices are also implemented:

```
picture surface(real f(pair z), pair a, pair b, int n=nmesh, int nsub=nsub,
                pen surfacepen=lightgray, pen meshpen=currentpen,
                projection P=currentprojection);
picture surface(real[][] f, pair a, pair b,
                pen surfacepen=lightgray, pen meshpen=currentpen,
                projection P=currentprojection);
```

Here is an example of a Gaussian surface:

```
import graph3;

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);

draw(arc(0.12Z,0.2,90,60,90,15),ArcArrow);

picture surface=surface(f,(-1,-1),(1,1),10,4);

bbox3 b=limits(O,1.75(1,1,1));

xaxis(Label("$x$",1),b,red,Arrow);
yaxis(Label("$y$",1),b,red,Arrow);
zaxis(Label("$z$",1),b,red,Arrow);

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

add(surface);
```



4.13.6 featpost3D

To facilitate the conversion of existing MetaPost programs, this module contains a partial port of the MetaPost 3D package featpost3D of L. Nobre G., C. Barbarosie, and J. Schwaiger to Asymptote. However, much (but not all) of the functionality of this port is now obsoleted by the general package three, which fully extends the notion of a path to three-dimensions. The original package is documented at

http://matagalatlante.org/nobre/featpost/doc/featpost.html

4.13.7 math

This package extends Asymptote's mathematical capabilities with radian/degree conversion routines, point-in-polygon and intersection algorithms, 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 intersect(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);
```

If the matrix a is tridiagonal, the routine tridiagonal provides a more efficient algorithm (see [tridiagonal], page 42).

`real[][] solve(real[][] a, real[][] b, bool overwrite=false)`
: Solve the linear equation $\mathtt{a}x = \mathtt{b}$ 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.

4.13.8 `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.

4.13.9 `stats`

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

4.13.10 `patterns`

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

4.13.11 `palette`

This package implements color density images and palette bars, along with several predefined palettes (see [images], page 67).

4.13.12 `tree`

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

4.13.13 `drawtree`

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

4.13.14 `feynman`

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

4.13.15 `MetaPost`

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

4.13.16 `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
```

To use cyrillic fonts, you will need to change the font encoding:

```
import unicode;
texpreamble("\usepackage{mathtext} \usepackage[russian]{babel}");
defaultpen(font("T2A","cmr"));
```

4.13.17 `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.

4.13.18 `babel`

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

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

4.14 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.

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 25).

If you wish to draw `PostScript` objects on top of labels (or verbatim `tex` commands; see [tex], page 29), 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, Label L="", path g,
          align align=NoAlign, pen p=currentpen,
          arrowbar arrow=None, arrowbar bar=None, margin margin=NoMargin,
          string legend="", marker marker=nomarker);
```

Draw the path `g` on the picture `pic` using pen `p` for drawing, with optional drawing attributes (Label `L`, explicit label alignment `align`, arrows and bars `arrow` and `bar`, margins `margin`, legend, and markers `marker`). 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.

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`, `MidArrow`, `EndArrow` (or equivalently `Arrow`), and `Arrows` (which draws an arrow at both ends of the path). 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 (except also for `MidArrow` and `Arrows`) 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`), `MidArcArrow`, 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`.

The legend for a picture `pic` can be fit and aligned to a frame with the routine (see [filltype], page 26)

```
frame legend(picture pic=currentpicture, pair dir=0,
             real xmargin=legendmargin, real ymargin=xmargin,
             pen p=currentpen);
```

this legend frame can then be added about the point `origin` to a picture `dest` using `add` or `attach` (see [add about], page 26).

To draw a dot, simply draw a path containing a single point. The `dot` command defined in `plain.asy` draws a dot having a a diameter equal to an explicit pen linewidth or the default linewidth magnified by `dotfactor` (6 by default):

```
void dot(picture pic=currentpicture, pair z, pen p=currentpen);
void dot(picture pic=currentpicture, pair[] z, pen p=currentpen);
void dot(picture pic=currentpicture, Label L, pair z, align align=NoAlign,
         string format=defaultformat, pen p=currentpen)
void dot(picture pic=currentpicture, Label L, pen p=currentpen)
```

The third routine draws a dot at every point of a pair array `z`. If the special variable `Label` is given as the `Label` argument to the fourth routine, the `format` argument will be used to format a string based on the dot location (here `defaultformat` is `"$%.4g$"`). One can also draw a dot at every node of a path:

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

See [markers], page 54 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, Label L="", path g,
          align align=NoAlign, pen p=currentpen, arrowbar arrow=None,
          arrowbar bar=None, margin margin=NoMargin, string legend="",
          marker marker=nomarker);
```

5.2 fill

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

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

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 fillpen=currentpen,
              pen drawpen=currentpen);
```

This fixed-size version of `fill` allows one to fill an object described in `PostScript` coordinates about the user coordinate `origin`:

```
void fill(pair origin, picture pic=currentpicture, path g, pen p=currentpen);
```

This is just a convenient abbreviation for the commands:

```
picture opic;
fill(opic,g,p);
add(origin,pic,opic);
```

Lattice gradient shading varying smoothly over a two-dimensional array of pens `p`, using fillrule `fillrule`, can be produced with

```
void latticeshade(picture pic=currentpicture, path g,
                  pen fillrule=currentpen, pen[][] p)
```

The pens in `p` must belong to the same color space. One can use the functions `rgb(pen)` or `cmyk(pen)` to promote pens to a higher color space, as illustrated in the example file `latticeshading.asy`.

Axial gradient shading varying smoothly from `pena` to `penb` in the direction of the line segment `a--b` can be achieved with

```
void axialshade(picture pic=currentpicture, path g,
                pen pena, pair a,
                pen penb, pair b);
```

Radial gradient shading varying smoothly from `pena` on the circle with center `a` and radius `ra` to `penb` on the circle with center `b` and radius `rb` is similar:

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

Illustrations of radial shading are provided in the example files `shade.asy` and `ring.asy`.

Gouraud shading using fillrule `fillrule` and the vertex colors in the pen array `p` on a triangular lattice defined by the vertices `z` and edge flags `edges` is implemented with

```
void gouraudshade(picture pic=currentpicture, path g,
                  pen fillrule=currentpen, pen[] p, pair[] z,
                  int[] edges);
```

The pens in `p` must belong to the same color space. An illustration of Gouraud shading is provided in the example file `Gouraud.asy`.

The following routine uses `evenodd` clipping together with the `^^` operator to unfill a region:

```
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 20). For an illustration of picture clipping, see the first example in Chapter 6 [LaTeX usage], page 90.

5.4 label

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

Draw Label `L` on picture `pic` using pen `p`. If `align` is `NoAlign`, the label will be centered at user coordinate `position`; otherwise it will be aligned in the direction of `align` and displaced from `position` by the `PostScript` offset `align*labelmargin(p)`. If `p` is `nullpen`, the pen specified within the Label, which defaults to `currentpen`, will be used. The Label `L` can either be a string or the structure obtained by calling one of the functions

```
Label Label(string s="", pair position, align align=NoAlign,
            pen p=nullpen, filltype filltype=NoFill);
Label Label(string s="", align align=NoAlign,
            pen p=nullpen, filltype filltype=NoFill);
Label Label(Label L, pair position, align align=NoAlign,
            pen p=nullpen, filltype filltype=NoFill);
Label Label(Label L, align align=NoAlign,
            pen p=nullpen, filltype filltype=NoFill);
```

The text of a Label can be scaled horizontally and/or vertically by multiplying it on the left with `xscale(real)`, `yscale(real)`, or `scale(real)`. After optionally scaling a Label, it can be rotated by an angle by multiplying it on the left with a rotation (in degrees): for example, `rotate(45)*xscale(2)*L` first scales `L` in the x direction and then rotates it counterclockwise by 45 degrees. The final position of a Label can also be shifted by a `PostScript` coordinate translation like this: `shift(10,0)*L`.

To add a label to a path, use

```
void label(picture pic=currentpicture, Label L, path g, align align=NoAlign,
           pen p=nullpen, filltype filltype=NoFill);
```

By default the label will be positioned at the midpoint of the path. An alternative label location (an `arctime` value between 0 and `length(g)` see [arctime], page 17) may be specified as real value for `position` in constructing the Label. The position `Relative(real)` specifies a location relative to the total arclength of the path.

Path labels are aligned in the direction `align`, which may be specified as an absolute compass direction (pair) or a direction `Relative(pair)` measured relative to a north axis in the local direction of the path. For convenience `LeftSide`, `Center`, and `RightSide` are defined as `Relative(W)`, `Relative((0,0))`, and `Relative(E)`, respectively. Multiplying `LeftSide`, `Center`, `RightSide` on the left by a real scaling factor will move the label further away from or closer to the path.

A label with a fixed-size arrow of length `arrowlength` pointing to `b` from direction `dir` can be produced with the routine

```
void arrow(picture pic=currentpicture, Label L="", pair b, pair dir,
           real length=arrowlength, align align=NoAlign,
           pen p=currentpen, arrowbar arrow=Arrow, margin margin=EndMargin);
```

If no alignment is specified (either in the Label or as an explicit argument), the optional Label will be aligned in the direction `dir`, using margin `margin`.

The function `string includegraphics(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:

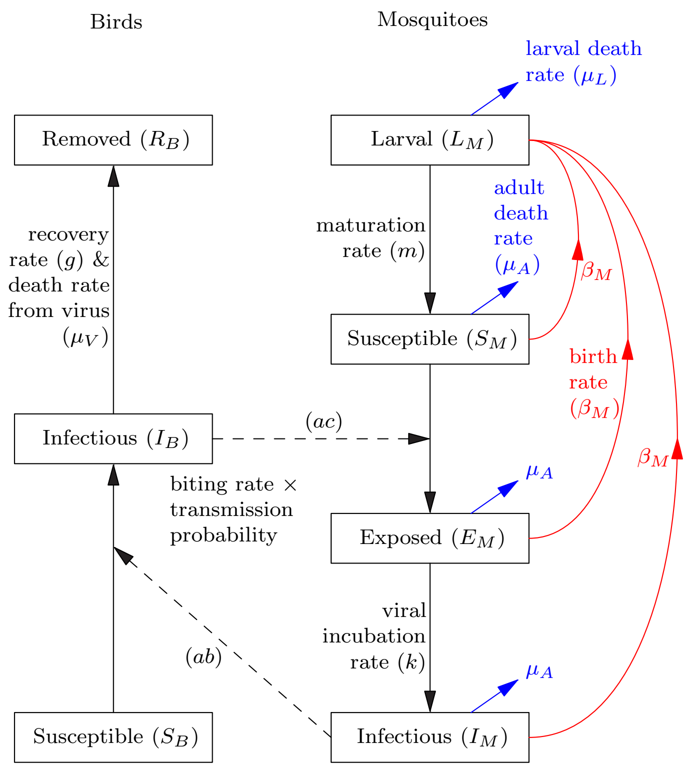
```
label(includegraphics("file.eps","width=1cm"),(0,0),NE);
layer();
```

The `string baseline(string s, align 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),
     Label(minipage("\flushleft{larval death rate ($\mu_L$)}",45pt),1),
     arrow,blue,Arrow);
draw(point(susceptibleM,NNE),
     Label(minipage("\flushleft{adult death rate ($\mu_A$)}",20pt),1),
     arrow,N,blue,Arrow);
draw(point(exposed,NNE),Label("$\mu_A$",1),arrow,blue,Arrow);
draw(point(infectiousM,NNE),Label("$\mu_A$",1),arrow,blue,Arrow);
```



One can prevent labels from overwriting one another with the `overwrite` pen attribute (see [overwrite], page 23).

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. Simply add the line \usepackage{asymptote} at the beginning of your file and enclose your Asymptote code within a \begin{asy}...\end{asy} environment. As with the LaTeX comment environment, the \end{asy} command must appear on a line by itself, with no leading spaces or trailing commands/comments.

The sample LaTeX file below, named latexusage.tex, can be run as follows:

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

If the [inline] package option is given to asymptote.sty, and the -t option is given to asy, inline LaTeX code is generated instead of EPS files. This makes LaTeX symbols visible to the \begin{asy}...\end{asy} environment. In this mode, Asymptote correctly aligns LaTeX symbols defined outside of \begin{asy}...\end{asy}, but treats their size as zero; an optional second string can be given to Label to provide an estimate of the unknown label size. Note that labels might not show up in DVI viewers that cannot handle raw PostScript code; use dvips/dvipdf to produce PostScript/PDF output. We recommend using the modified version of dvipdf in the Asymptote patches directory, which accepts the dvips -z hyperdvi option.

Here now is latexusage.tex:

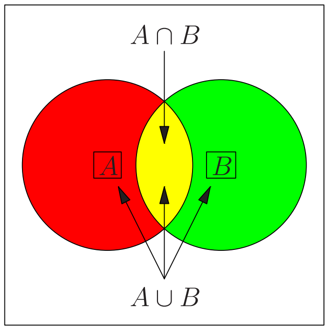
```
\documentclass[12pt]{article}

% Use this with 'asy latexusage' to include eps files:
\usepackage{asymptote}

% Use this with 'asy -t latexusage' to include inline LaTeX code.
%\usepackage[inline]{asymptote}

% Enable this line to produce pdf hyperlinks
%\usepackage[hypertex]{hyperref}

\begin{document}
\begin{asydef}
// Global definitions can be put here.
\end{asydef}

Here is a venn diagram
%(Figure~\ref{venn})
produced with Asymptote, drawn to width 5cm:

\def\A{A}
\def\B{B}
```

```
%\begin{figure}
\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);

//box(Label("$\A$",z1));              // Requires [inline] package option.
//box(Label("$\B$","$B$",z2));        // Requires [inline] package option.
box(Label("$A$",z1));
box(Label("$B$",z2));

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

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

shipout(bbox(0.25cm));
\end{asy}
%\caption{Venn diagram}\label{venn}
\end{center}
%\end{figure}

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

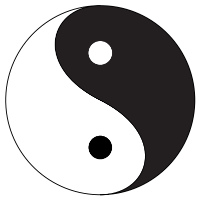
```
\begin{center}
\begin{asy}[3cm,0]
guide center = (0,1){W}..tension 0.8..(0,0){(1,-.5)}..tension 0.8..{W}(0,-1);

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,operator ..)--(1,0);
fill(p--cycle,lightgray);
draw(p);
draw((0,0)--graph(g,0,1,operator ..)--(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);

dot(Label,(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("$x$",0,c.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}
```

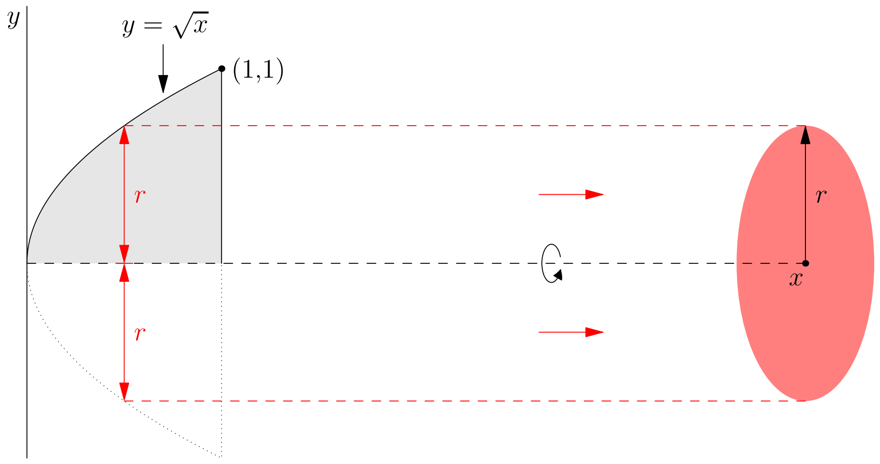
Here is a venn diagram produced with Asymptote, drawn to width 5cm:



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 (MSDOS default)
-n, -noView       Don't view output file (UNIX default)
-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 (- denotes standard output)
-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
-t                Produce LaTeX file for \usepackage[inline]{asymptote}
-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
-localhistory     Use a local interactive history file
-noplain          Disable automatic importing of plain
```

If no arguments are given, `Asymptote` runs in interactive mode (see Chapter 8 [Interactive mode], page 97). In this case, the default output file is `out.eps`.

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 (version 6.2.4 or later recommended).

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.

Default values for these options may also be entered in the file `.asy/options` in the user's home directory (in the same format as on the command line, except that the options may be distributed over multiple lines). Command-line options override these defaults.

8 Interactive mode

Interactive mode is entered by executing the command `asy` with no file arguments. Each line must be a complete `Asymptote` statement; however, it is not necessary to terminate each line with a semicolon.

The following special commands are supported only in interactive mode and must be entered immediately after the prompt:

`help` view the manual

`reset` reset `Asymptote` to its initial state, except that a prior call to `scroll` is respected (see [scroll], page 43).

`input FILE`
resets the environment and does an `erase(); include FILE`. If the file name `FILE` contains nonalphanumeric characters, enclose it with quotation marks. For convenience, a trailing semi-colon followed by optional `Asymptote` commands may be entered on the same line.

`quit` exit interactive mode (abbreviated as `q`; `exit` is a synonym). A history of the most recent `1000` previous commands will be retained in the file `.asy/history` in the user's home directory (unless the command line option `-localhistory` was specified, in which case the history will be stored in the file `.asy_history` in the current directory).

Typing `ctrl-C` interrupts the execution of `Asymptote` code and returns control to the interactive prompt.

Interactive mode is implemented with the GNU readline library. To customize the key bindings, see: `http://cnswww.cns.cwru.edu/php/chet/readline/readline.html`

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`.

To use `xasy`, one must first deconstruct `Asymptote` pictures into transparent GIF images with the command `asy -xN`, where `N` denotes the magnification (a positive real number, say 2). The command `asy -VxN` automatically invokes `xasy` once deconstruction is complete. Alternatively, one may turn on the `-xN` option in interactive mode or from within a module using the function `gui()` or `gui(N)`. One can turn GUI mode off again with `gui(0)`.

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 (included with `Python` under `MSDOS`). Under `Fedora Core 4`, you can either install `tkinter` with the command

```
yum install tkinter
```

or manually install the individual packages:

```
rpm -i tkinter-2.4.1-2.i386.rpm
rpm -U --nodeps tix-8.1.4-100.i386.rpm
rpm -U --nodeps tk-8.4.9-3.i386.rpm
```

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 be individually deconstructed (see [add], page 26).

10 PostScript to Asymptote

The excellent PostScript editor pstoedit (version 3.42 or later; available from http://pstoedit.net) includes an Asymptote backend. Unlike virtually all other pstoedit backends, this driver includes native clipping, even-odd fill rule, and PostScript subpath support.

For example, try:

```
asy -V /usr/share/doc/asymptote/examples/venn.asy
```

```
pstoedit -f asy 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.

Full image support can be added to the Asymptote backend with the patch pstoedit-3.42asy.patch. Denoting the location of the Asymptote source directory by ASYMPTOTE_SOURCE, one can install this patch with the commands (as the UNIX root user):

```
tar -zxf pstoedit-3.42.tar.gz
cd pstoedit-3.42
patch -p1 < ASYMPTOTE_SOURCE/patches/pstoedit-3.42asy.patch
autoconf
./configure --prefix=/usr
make install
```

11 Help

Questions on installing and using `Asymptote` should be sent to the `Asymptote` forum.

> `http://sourceforge.net/forum/forum.php?forum_id=409349`

Contributions in the form of patches or `Asymptote` modules can be posted here:

> `http://sourceforge.net/tracker/?atid=685685&group_id=120000`

To receive announcements of upcoming releases, please subscribe to `Asymptote` at

> `http://freshmeat.net/subscribe/50750`

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 36) 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

!

! . . . 33
!= . . . 33

%

% . . . 32
%= . . . 33

&

& . . . 14
&& . . . 33

*

* . . . 19, 32
** . . . 32
*= . . . 33

+

+ . . . 19, 32
++ . . . 33
+= . . . 33

-

- . . . 32
-- . . . 12, 33
--- . . . 14
-= . . . 33
-V . . . 2, 6

.

.. . . . 12

/

/ . . . 32
/= . . . 33

:

:: . . . 14

<

< . . . 33
<= . . . 33

=

== . . . 33

>

> . . . 33
>= . . . 33

^

^ . . . 32, 33
^= . . . 33
^^ . . . 15

|

|| . . . 33

2

2D graphs . . . 47

3

3D graphs . . . 74

A

abs . . . 9, 39
access . . . 44
acknowledgments . . . 101
acos . . . 39
acosh . . . 39
add . . . 26
alias . . . 41
Allow . . . 23
and . . . 13
angle . . . 9
animation . . . 8
append . . . 29, 39
ArcArrow . . . 83
ArcArrows . . . 83
arclength . . . 17, 72
arctime . . . 17, 72
arguments . . . 36
arithmetic operators . . . 32
arrays . . . 39
arrow . . . 83, 86
Arrow . . . 83
Arrows . . . 83
as . . . 45
asin . . . 39
asinh . . . 39
aspect . . . 74

assignment . . . 8
`asy-mode` . . . 4
`asy.vim` . . . 4
`asymptote.sty` . . . 90
`ASYMPTOTE_DIR` . . . 3
`ASYMPTOTE_GS` . . . 2
`ASYMPTOTE_PAPERTYPE` . . . 96
`ASYMPTOTE_PDFVIEWER` . . . 2
`ASYMPTOTE_PSVIEWER` . . . 2
`atan` . . . 39
`atan2` . . . 39
`atanh` . . . 39
`atleast` . . . 13
`attach` . . . 53
`autolimits` . . . 74
automatic scaling . . . 57
axial shading . . . 85
`axis` . . . 62, 65, 76, 77
`azimuth` . . . 10

B

`babel` . . . 81
`Bar` . . . 83
`Bars` . . . 83
`basealign` . . . 21
`baseline` . . . 87
`baselineskip` . . . 21
batch mode . . . 6
`bbox3` . . . 74
`BeginArcArrow` . . . 83
`BeginArrow` . . . 83
`BeginBar` . . . 83
`BeginDotMargin` . . . 83
`BeginMargin` . . . 83
`BeginPenMargin` . . . 83
`beveljoin` . . . 20
binary operators . . . 32
`Blank` . . . 83
`bool` . . . 8
boolean operators . . . 32
`Bottom` . . . 48
`BottomTop` . . . 49
`box` . . . 26
`bp` . . . 6
`break` . . . 8
brick . . . 22
broken axis . . . 60
bug reports . . . 100
`Button-1` . . . 98
`Button-2` . . . 98
`BWRainbow` . . . 67
`BWRainbow2` . . . 67

C

`C` string . . . 10
casts . . . 43
`cbrt` . . . 39
`cd` . . . 29
`ceil` . . . 39
`Center` . . . 86
checker . . . 22
`circle` . . . 12
`clear` . . . 30
`clip` . . . 86
`cm` . . . 7
`cmyk` . . . 19
`colatitude` . . . 10
color . . . 19
`colors` . . . 19
comma-separated-value . . . 43
comment character . . . 29
Compiling from UNIX source . . . 4
`concat` . . . 41
conditional . . . 8
`conj` . . . 9
constructors . . . 31
`continue` . . . 8
`controls` . . . 13, 70
`convert` . . . 8
`copy` . . . 41
`cos` . . . 39
`cosh` . . . 39
`crop` . . . 56
cropping graphs . . . 56
`cross` . . . 10, 54
crosshatch . . . 22
csv . . . 43
`cubicroots` . . . 42
`curl` . . . 15
`curl3` . . . 70
`currentpen` . . . 19
`currentprojection` . . . 71
custom tick locations . . . 50
CVS . . . 5
`cycle` . . . 6, 12
`cycle3` . . . 70
`cyclic` . . . 18, 39, 72
`cyclicflag` . . . 39
Cyrillic . . . 80

D

`dashdotted` . . . 19
`dashed` . . . 19
data types . . . 8
Debian . . . 2
declaration . . . 8
default arguments . . . 36
`defaultpen` . . . 19, 20, 21, 23
`degrees` . . . 9
description . . . 1
`dimension` . . . 43
`dir` . . . 9, 10, 16, 72
directory . . . 29

dirtime 17
do 8
dot 10, 84
DotMargin 83
DotMargins 83
dotted 19
double 30
draw 83, 84
drawing commands 83
drawline 79
drawtree 80

E

Editing modes 4
else 8
emacs 4
EndArcArrow 83
EndArrow 83
EndBar 83
EndDotMargin 83
endl 29
EndMargin 83
EndPenMargin 83
environment variables 2
eof 30, 42
eol 30, 42
EPS 87
erase 11, 28
erf 39
erfc 39
error 29, 30
errorbars 54
eval 46
evenodd 15
evenodd 20
evenoddoverlap 20
examples 6
execute 46
exit 97
exp 39
expi 9, 10
explicit 43
explicit casts 44
extendcap 20
extension 79

F

fabs 39
face 72
featpost3D 79
feynman 80
fft 41
file 29
fill 84, 85
Fill 26, 83
filldraw 85
fillrule 20
find 11, 40
firstcut 17
fit 25
floor 39
flush 30
fmod 39
font 21
fontsize 21
for 8
format 12
forum 100
frame 24
from 44
function declarations 35
functions 35, 39

G

gamma 39
Gaussrand 39
geometry 80
getc 29
gifmerge 8
Gouraud shading 85
gradient shading 85
graph 47
graph3 74
graphical user interface 98
gray 19
grayscale 19
Grayscale 67
grid 22, 58
GUI 98
guide 12
guide3 70

H

hatch 22
help 97
help 100
hidden surface removal 72
histogram 39
hypot 39

I

identity 24, 39
if 8
ImageMagick 8
implicit casts 43
implicit linear solver 79
implicit scaling 34
import 45
inches 7
include 87
including images 87

inheritance 31
input 29, 97
insert 11
inside 18, 72
installation 2
int 8
integer division 32
interactive mode 97
intersect 17, 72, 79
intersection 79
intersectionpoint 17
intersectpoint 72
invisible 19

J

J 39

K

keywords 36

L

label 86
Label 51
Label 84
labelx 51
labely 51
landscape mode 25
lastcut 18
LaTeX fonts 21
LaTeX usage 90
latin1 81
latitude 10
lattice shading 85
layer 83
leastsquares 63, 80
Left 50
LeftRight 50
LeftSide 86
LeftTicks 53
legend 83
length 9, 11, 16, 39, 72
libm routines 39
libsigsegv 36, 100
limits 56, 74
line 42
line mode 42
Linear 57
linecap 20
linejoin 20
lineskip 21
linewidth 20
log 39
Log 57
log-log graph 57
log10 39
logarithmic graph 57
logical operators 32
longdashdotted 19
longdashed 19
longitude 10
Longitude 10
loop 8

M

map 41
Margin 83
Margins 83
marker 54
marknodes 54
markuniform 54
math 79
mathematical functions 39
max 18, 41, 72
merge 8
MetaPost 80
MetaPost 14
MetaPost cutafter 18
MetaPost cutbefore 18
MetaPost pickup 19
MetaPost whatever 79
Microsoft Windows 2
MidArrow 83
min 18, 41, 72
minipage 87
miterjoin 20
mm 7
mouse 98
Move 23
MoveQuiet 23

N

named arguments 36
new 30, 40
newframe 24
NFSS 21
nobasealign 21
NoFill 26, 83
NoMargin 83
none 29
None 83
NoTicks 49

O

oblique 71
offset 96
open 29
operator 33
operator -- 47
operator 47
operator cast 44

operator ecast . . . 44
operator init . . . 30
operators . . . 32
options . . . 95
orthographic . . . 71
output . . . 29
overwrite . . . 23

P

packing . . . 38
pair . . . 9
palette . . . 80
parametrized curve . . . 56
path . . . 12
path[] . . . 15
patterns . . . 21
patterns . . . 80
pen . . . 19
PenMargin . . . 83
PenMargins . . . 83
perpendicular . . . 80
perspective . . . 71
picture . . . 25
picture alignment . . . 25
plain . . . 47
point . . . 16, 72
polar . . . 9
polygon . . . 54
pop . . . 39
portrait mode . . . 25
postcontrol . . . 17, 72
postfix operators . . . 33
postscript . . . 28
PostScript fonts . . . 21
PostScript subpath . . . 15
pow10 . . . 39
precision . . . 30
precontrol . . . 16, 72
prefix operators . . . 33
private . . . 30
programming . . . 8
pt . . . 7
public . . . 30
push . . . 39

Q

quadraticroots . . . 42
quit . . . 97
quotient . . . 32

R

radial shading . . . 85
Rainbow . . . 67
rand . . . 39
randMax . . . 39
read1 . . . 43
read2 . . . 43
read3 . . . 43
reading . . . 29
real . . . 8
recursion . . . 36
reflect . . . 24
Relative . . . 86
remainder . . . 39
replace . . . 11
resetdefaultpen . . . 23
rest arguments . . . 37
restore . . . 28
reverse . . . 11, 17, 40, 72
rfind . . . 11
rgb . . . 19
Right . . . 50
RightSide . . . 86
RightTicks . . . 53
round . . . 39
roundcap . . . 20
roundjoin . . . 20

S

save . . . 28
scale . . . 24, 57
scale3 . . . 72
scientific graph . . . 53
scroll . . . 43
search . . . 40
search paths . . . 3
seascape mode . . . 25
secondary axis . . . 60
secondaryX . . . 60
secondaryY . . . 60
segmentation fault . . . 100
self operators . . . 33
sequence . . . 40
sgn . . . 39
shading . . . 85
shift . . . 24, 71
shipout . . . 25
simplex . . . 47
sin . . . 39
single . . . 30
sinh . . . 39
size . . . 16, 25, 72
slant . . . 24
slice . . . 17
solid . . . 19
solve . . . 79
sort . . . 41
Spline . . . 47
sqrt . . . 39
squarecap . . . 20
srand . . . 39
stack overflow . . . 36, 100

static 81
stats 80
stdin 30
stdout 30
straight 18, 72, 80
Straight 47
string 10
struct 30
structures 30
subpath 17, 72
subpictures 25
substr 11
sum 41
superpath 15
Suppress 23
SuppressQuiet 23

T

tab 29
tan 39
tanh 39
tension 13
tension3 70
tex 29
TeX fonts 21
TEX string 10
texpreamble 29
textbook graph 52
this 30
three 70
tick 51
ticks 49
tile 22
tilings 21
time 12
Top 48
transform 24, 71
transpose 41
tree 80
triangle 80
tridiagonal 42
triple 9
TrueMargin 83
type1cm 21
typedef 12, 35

U

unfill 85
UnFill 26
UnFill(real xmargin=0, real ymargin=xmargin 26
unicode 80
Uninstall 5
unit 9, 10
unitcircle 12
UNIX binary distributions 2
unpacking 38
unravel 45
user-defined operators 33

V

vector field 65
verbatim 28
vim 4
virtual functions 31
void 8

W

wheel mouse 98
while 8
write 29, 43

X

xasy 98
xaxis 74
xequals 50
XEquals 50
xinput 30
xlimits 56, 74
xoutput 30
xpart 9, 10
xscale 24
xscale3 71
xtick 51
XZero 50

Y

Y 39
yequals 50
YEquals 48
ylimits 56, 74
ypart 9, 10
yscale 24
yscale3 71
ytick 51
YZero 48

Z

zerowinding 20
zerowindingoverlap 20
zlimits 74
zpart 10
zscale3 71