Asymptote: the Vector Graphics Language

For version 1.32



This file documents `Asymptote`, version 1.32.

http://asymptote.sourceforge.net

Copyright © 2004-7 Andy Hammerlindl, John Bowman, and Tom Prince.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU General Public License (see the file LICENSE in the top-level source directory).

Table of Contents

1 Description 1

2 Installation 3
2.1 UNIX binary distributions 3
2.2 MacOS X binary distributions 3
2.3 Microsoft Windows 4
2.4 Configuring 4
2.5 Search paths 5
2.6 Compiling from UNIX source 6
2.7 Editing modes 7
2.8 Subversion 7
2.9 Uninstall 8

3 Tutorial 9

4 Drawing commands 15
4.1 draw 15
4.2 fill 17
4.3 clip 18
4.4 label 18

5 Programming 23
5.1 Data types 23
5.2 Paths and guides 28
5.3 Pens 34
5.4 Transforms 41
5.5 Frames and pictures 42
5.6 Files 48
5.7 Variable initializers 50
5.8 Structures 51
5.9 Operators 55
5.9.1 Arithmetic & logical operators 55
5.9.2 Self & prefix operators 56
5.9.3 User-defined operators 56
5.10 Implicit scaling 57
5.11 Functions 57
5.11.1 Default arguments 59
5.11.2 Named arguments 59
5.11.3 Rest arguments 60
5.11.4 Mathematical functions 62
5.12 Arrays 62
5.13 Casts 68

5.14 Import 69
5.15 Static 72

6 LaTeX usage 74

7 Base modules 79

7.1 plain 79
7.2 simplex 79
7.3 math 79
7.4 interpolate 80
7.5 geometry 80
7.6 stats 80
7.7 patterns 80
7.8 markers 80
7.9 tree 82
7.10 binarytree 82
7.11 drawtree 83
7.12 syzygy 83
7.13 feynman 83
7.14 roundedpath 83
7.15 animate 83
7.16 embed 84
7.17 slide 84
7.18 MetaPost 85
7.19 unicode 85
7.20 latin1 85
7.21 babel 85
7.22 labelpath 85
7.23 annotate 86
7.24 CAD 86
7.25 graph 86
7.26 palette 112
7.27 three 116
7.28 light 121
7.29 graph3 121
7.30 grid3 126
7.31 solids 127
7.32 surface 128
7.33 featpost3D 128
7.34 flowchart 128
7.35 contour 130
7.36 contour3 136
7.37 slopefield 136

8 Options 138

9 Interactive mode 141

10 Graphical User Interface 143
10.1 GUI Installation 143
10.2 GUI Usage 143

11 `PostScript` to `Asymptote` 145

12 Help 146

13 Debugger 147

14 Acknowledgments 148

Index 149

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 4 [Drawing commands], page 15).

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 and capabilities) called MetaPost.

The Asymptote vector graphics language provides:

- a standard for typesetting mathematical figures, just as TEX/LaTeX is the de-facto standard for typesetting equations.
- LaTeX typesetting of labels, for overall document consistency;
- a natural coordinate-based framework for technical drawings, inspired by MetaPost, with a much cleaner, powerful C++-like programming syntax;
- 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;

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 7.25 [graph], page 86). Examples of Asymptote code and output, including animations, are available at

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

Links to many external resources, including an excellent user-written `Asymptote` tutorial can be found at

http://asymptote.sourceforge.net/links.html.

2 Installation

After following the instructions for your specific distribution, please see also Section 2.4 [Configuring], page 4.

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

We release both `tgz` and RPM binary distributions of `Asymptote`. The root user can install the `Linux i386 tgz` distribution of version `x.xx` of `Asymptote` with the commands:

```
tar -C / -zxf asymptote-x.xx.i386.tgz
texhash
```

The `texhash` command, which installs LaTeX style files, is optional. The executable file will be `/usr/local/bin/asy`) and example code will be installed by default in `/usr/share/doc/asymptote/examples`.

Fedora Core users can easily install `Asymptote` from the Fedora Extras mirrors:

```
yum install asymptote
```

To install the latest version of `Asymptote` on a Debian-based distribution (e.g. Ubuntu, Mepis, Linspire) follow the instructions for compiling from `UNIX` source (see Section 2.6 [Compiling from UNIX source], page 6). Alternatively, Debian users can install one of Hubert Chan's prebuilt `Asymptote` binaries from

http://ftp.debian.org/debian/pool/main/a/asymptote

2.2 MacOS X binary distributions

MacOS users can either compile the `UNIX` source code (see Section 2.6 [Compiling from UNIX source], page 6) or install the contributed `Asymptote` binary available at

http://www.hmug.org/pub/MacOS_X/X/Applications/Publishing/asymptote/

Because these preconfigured binary distributions have strict architecture and library dependencies that many installations do not satisfy, we recommend installing `Asymptote` directly from the official source:

http://sourceforge.net/project/showfiles.php?group_id=120000

Note that many MacOS X (and FreeBSD) systems inexplicably ship with an extremely old GNU `readline` version (4.1, dated 21 March 2000). For full interactive functionality, `readline` version 4.2 or later (16 April 2001) is required.

2.3 Microsoft Windows

Users of the Microsoft Windows operating system can install the self-extracting Asymptote executable asymptote-x.xx-setup.exe, where x.xx denotes the latest version.

A working TeX implementation (such as the one available at http://www.miktex.org) will be required to typeset labels. You will also need to install GPL Ghostscript from http://sourceforge.net/projects/ghostscript/. To view the default PostScript output, you can install the program gsview available from http://www.cs.wisc.edu/~ghost/gsview/.

The ImageMagick package from

http://www.imagemagick.org/script/binary-releases.php

is required to support alternate output formats (see [convert], page 139), including the graphical user interface and animations. The Python interpreter from http://www.python.org is only required if you wish to try out the graphical user interface (see Chapter 10 [GUI], page 143).

Example code will be installed by default in the examples subdirectory of the installation directory (by default, C:\Program Files\Asymptote).

2.4 Configuring

In interactive mode, or when given the -V option (the default when running Asymptote on a single file under MSDOS), Asymptote will automatically invoke the PostScript viewer gv (under UNIX) or gsview (under MSDOS to display graphical output. These defaults may be overridden with the configuration variable psviewer. The PostScript viewer should be capable of automatically redrawing whenever the output file is updated. The default UNIX PostScript viewer gv supports this (via a SIGHUP signal). Version gv-3.6.2 or later (from http://ftp.gnu.org/gnu/gv/) is required for interactive mode to work properly. Users of ggv will need to enable Watch file under Edit/Postscript Viewer Preferences. Users of gsview 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).

Configuration variables are most easily set as Asymptote variables in the configuration file (by default, .asy/config.asy in the user's home directory or %USERPROFILE%\.asy\config.asy under MSDOS); see [configuration file], page 139. Here are the default values of several important configuration variables:

Under UNIX:

```
import settings;
psviewer="gv";
pdfviewer="acroread";
gs="gs";
python="";
```

Under MSDOS:

```
import settings;
psviewer="gsview32.exe";
pdfviewer="AcroRd32.exe";
gs="gswin32c.exe";
```

```
python="python.exe";
```

For PDF format output, the `gs` setting specifies the location of the `PostScript`-to-PDF processor `Ghostscript`, available from http://sourceforge.net/projects/ghostscript/.

The setting `pdfviewer` specifies the location of the PDF viewer. The graphical user interface may also require setting the variable `python` if `Python` is installed in a nonstandard location.

The configuration variable `dir` can be used to adjust the search path (see Section 2.5 [Search paths], page 5).

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 configuration variable `papertype`. Alignment to other paper sizes can be obtained by setting the configuration variables `paperwidth` and `paperheight`.

The following configuration variables normally do not require adjustment:

```
texpath
texcommand
dvips
convert
display
animate
xasy
```

Configuration variables may also be set or overwritten with a command-line option:

```
asy -psviewer=gsview -V venn
```

Alternatively, system environment versions of the above configuration variables may be set in the conventional way. The corresponding environment variable name is obtained by converting the configuration variable name to upper case and prepending `ASYMPTOTE_`: for example, to set the environment variable

```
ASYMPTOTE_PSVIEWER="C:\Program Files\Ghostgum\gsview\gsview32.exe";
```

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.

2.5 Search paths

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

1. The current directory;
2. The directory `.asy` in the user's home directory (`%USERPROFILE%\.asy` under `MSDOS`);
3. A list of one or more directories specified by the configuration variable `dir` (separated by `:` under UNIX and `;` under `MSDOS`);
4. The `Asymptote` system directory (by default, `/usr/share/asymptote` under `UNIX` and `C:\Program Files\Asymptote` under `MSDOS`).

2.6 Compiling from UNIX source

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

```
gunzip asymptote-x.xx.src.tgz
tar -xf asymptote-x.xx.src.tar
cd asymptote-x.xx
```

Then put http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_source/gc-7.0alpha9.tar.gz in the current directory and

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

Be sure to use GNU make (on non-GNU systems this command may be called gmake). To build the documentation, you may need to install the texinfo-tex package. If you get errors from a broken texinfo or 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 the root user. To install without root privileges, change the ./configure command to

```
./configure --prefix=$HOME/asymptote
```

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 (provided that your system has the libgc and libgccpp libraries). One can disable use of the garbage collector by configuring with ./configure --disable-gc. For a list of other configuration options, say ./configure --help. For example, one can tell configure to look for header files and libraries in nonstandard locations:

```
./configure CFLAGS=-I/opt/usr/include LDFLAGS=-L/opt/usr/lib
```

If you are compiling Asymptote with gcc, you will need a relatively recent version (e.g. 3.2 or later). For full interactive functionality, you will need version 4.2 or later of the GNU readline library. The file gcc3.3.2curses.patch in the patches directory can be used to patch the broken curses.h header file (or a local copy thereof in the current directory) on some AIX and IRIX systems.

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). The GSL library is only required if you require the special functions that it supports.

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 configuration variable dir to point to the directory base (in the top level directory of the Asymptote source code).

A patch supplied in the patches directory fixes several bugs in the UNIX PostScript viewer gv-3.6.2, allowing it to work properly with Asymptote. After putting gv-3.6.2 from http://ftp.gnu.org/gnu/gv/ in the Asymptote source directory, here are the steps to install a properly working gv in /usr/bin (as the UNIX root user):

```
tar -zxf gv-3.6.2.tar.gz
cd gv-3.6.2
patch -p1 < ../patches/gv-3.6.2.patch
./configure --prefix=/usr
make install
```

2.7 Editing modes

Users of emacs can edit Asymptote code with the mode asy-mode, after enabling it by putting the following lines in their .emacs initialization file, replacing ASYDIR with the location of the Asymptote system directory (by default, /usr/share/asymptote or C:\Program Files\Asymptote under MSDOS):

```
(add-to-list 'load-path "ASYDIR")
(autoload 'asy-mode "asy-mode.el" "Asymptote major mode." t)
(autoload 'lasy-mode "asy-mode.el" "hybrid Asymptote/Latex major mode." t)
(autoload 'asy-insinuate-latex "asy-mode.el" "Asymptote insinuate LaTeX." t)
(add-to-list 'auto-mode-alist '("\\.asy$" . asy-mode))
```

Particularly useful key bindings in this mode are C-c C-c, which compiles and displays the current buffer, and the key binding C-c ?, which shows the available function prototypes for the command at the cursor. For full functionality you should also install the Apache Software Foundation package two-mode-mode:

http://www.dedasys.com/freesoftware/files/two-mode-mode.el

Once installed, you can use the hybrid mode lasy-mode to edit a LaTeX file containing embedded Asymptote code (see Chapter 6 [LaTeX usage], page 74). This mode can be enabled within latex-mode with the key sequence M-x lasy-mode <RET>. On UNIX systems, additional keywords will be generated from all asy files in the space-separated list of directories specified by the environment variable ASYMPTOTE_SITEDIR. Further documentation of asy-mode is available within emacs by pressing the sequence keys C-h f asy-mode <RET>.

Fans of vim can customize vim for Asymptote with

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

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

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

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

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

2.8 Subversion

The following commands are needed to install the latest development version of Asymptote using Subversion:

```
svn co https://asymptote.svn.sourceforge.net/svnroot/asymptote/trunk/asymptote

cd asymptote
./autogen.sh
wget http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_source/gc-7.0alpha9.tar.gz
./configure
make all
make install
```

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

2.9 Uninstall

To uninstall an `Linux i386` binary distribution, use the commands

```
tar -zxvf asymptote-x.xx.i386.tgz | xargs --replace=% rm /%
texhash
```

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

```
make uninstall
```

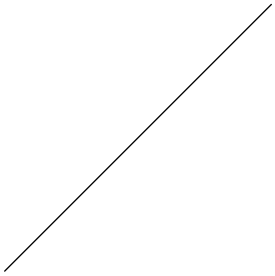
3 Tutorial

An excellent user-written `Asymptote` *tutorial is also available from*

http://www.artofproblemsolving.com/Wiki/index.php/Asymptote:_Basics

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. In `Asymptote` coordinates like `(0,0)` and `(1000,100)` are called *pairs*.

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). The tab key will automatically complete unambiguous words; otherwise, hitting tab again will show the possible choices. Further commands specific to interactive mode are described in Chapter 9 [Interactive mode], page 141.

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 with the `MSDOS` or `UNIX` command

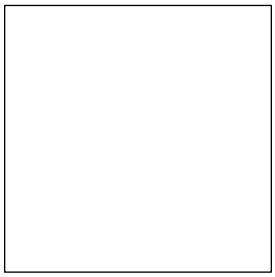
```
asy -V test
```

`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);
```



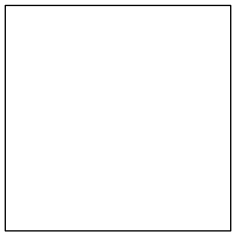
It is often inconvenient to work directly with PostScript coordinates. The next example draws a unit square scaled to width 101 bp and height 101 bp. The output is identical to that of the previous example.

```
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 42):

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



To make the user coordinates represent multiples of exactly `1cm`:

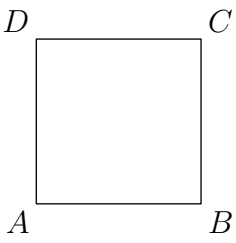
```
unitsize(1cm);
draw(unitsquare);
```

One can also specify different x and y unit sizes:

```
unitsize(1cm,2cm);
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:

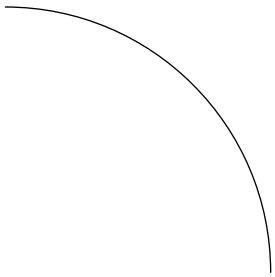
```
size(0,3cm);
draw(unitsquare);
label("$A$",(0,0),SW);
label("$B$",(1,0),SE);
label("$C$",(1,1),NE);
label("$D$",(0,1),NW);
```



`Asymptote` 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 `Asymptote` base module `plain`. A user who has a local variable named `E` may access the compass direction `E` by prefixing it with the name of the module where it is defined: `plain.E`.

This example draws a path that approximates a quarter circle:

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



In general, a path is specified as a list of pairs (or other 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:

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

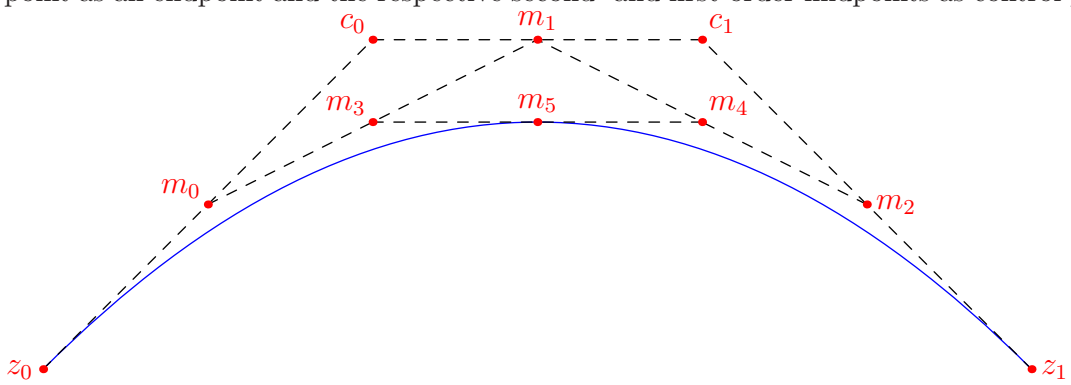
Each interior node of a cubic spline may be given a direction prefix or suffix `{dir}`: the direction of the pair `dir` specifies the direction of the incoming or outgoing tangent, respectively, to the curve at that node. Exterior nodes may be given direction specifiers only on their interior side.

A cubic spline between the node z_0, with postcontrol point c_0, and the node z_1, with precontrol point c_1, is computed as the Bezier curve

$$(1-t)^3 z_0 + 3t(1-t)^2 c_0 + 3t^2(1-t)c_1 + t^3 z_1 \qquad 0 \le t \le 1.$$

As illustrated in the diagram below, the third-order midpoint (m_5) constructed from two endpoints z_0 and z_1 and two control points c_0 and c_1, is the point corresponding to $t = 1/2$ on the Bezier curve formed by the quadruple (z_0, c_0, c_1, z_1). This allows one to recursively construct the desired curve, by using the newly extracted third-order mid-

point as an endpoint and the respective second- and first-order midpoints as control points:



Here m_0, m_1 and m_2 are the first-order midpoints, m_3 and m_4 are the second-order midpoints, and m_5 is the third-order midpoint. The curve is then constructed by recursively applying the algorithm to (z_0, m_0, m_3, m_5) and (m_5, m_4, m_2, z_1).

In fact, an analogous property holds for points located at any fraction t in $[0, 1]$ of each segment, not just for midpoints ($t = 1/2$).

The Bezier curve constructed in this manner has the following properties:

- It is entirely contained in the convex hull of the given four points.
- It starts heading from the first endpoint to the first control point and finishes heading from the second control point to the second endpoint.

The user can specify explicit control points between two nodes like this:

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

However, it is usually more convenient to just use the `..` operator, which tells `Asymptote` to choose its own control points using the algorithms described in Donald Knuth's monograph, The MetaFontbook, Chapter 14. The user can still customize the guide (or path) by specifying direction, tension, and curl values.

The higher the tension, the straighter the curve is, and the more it approximates a straight line. One can change the spline tension from its default value of 1 to any real value greater than or equal to 0.75 (cf. John D. Hobby, Discrete and Computational Geometry 1, 1986):

```
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 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 `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 5.11.3 [Rest arguments], page 60). For example, compare

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



with

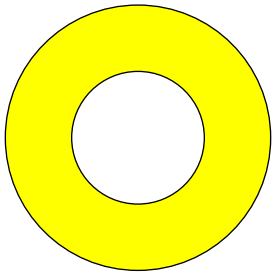
```
draw((0,0){up}::(100,25){right}::(200,0){down});
```



The `---` connector is an abbreviation for `..tension atleast infinity..` and the `&` connector concatenates two paths which meet at a common point.

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);
```



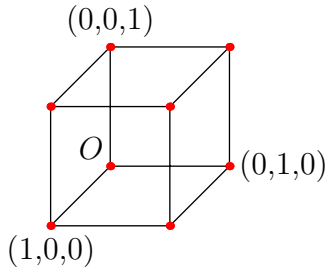
The `PostScript` even-odd fill rule here specifies that only the region bounded between the two unit circles is filled (see [fillrule], page 37). In this example, the same effect can be achieved by using the default zero winding number fill rule, if one is careful to alternate the orientation of the 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=obliqueX;

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

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



See section Section 7.25 [graph], page 86 (or the online `Asymptote` gallery at http://asymptote.sourceforge.net) for further examples, including two-dimensional scientific graphs. Additional examples have been posted by Philippe Ivaldi at http://piprim.tuxfamily.org/asymptote/.

4 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 43).

If you wish to draw PostScript objects on top of labels (or verbatim tex commands; see [tex], page 47), 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).

4.1 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,
          Label 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 Label 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), FillDraw, Fill, NoFill, Draw, and (except 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 44):

```
frame legend(picture pic=currentpicture, int perline=1,
             real xmargin=legendmargin, real ymargin=xmargin,
             real linelength=legendlinelength,
             real hskip=legendhskip, real vskip=legendvskip,
             real maxwidth=0, pen p=currentpen);
```

Here `xmargin` and `ymargin` specify the surrounding x and y margins, `perline` specifies the number of entries per line (default 1; 0 means choose this number automatically), `linelength` specifies the length of the path lines, `hskip` and `vskip` specify the line skip (as a multiple of legend entry size), `maxwidth` specifies an optional upper limit on the width of the resulting legend (0 means unlimited), and `p` specifies the pen used to draw the bounding box. The legend frame can then be added and aligned about a point on a picture `dest` using `add` or `attach` (see [add about], page 45).

To draw a dot, simply draw a path containing a single point. The `dot` command defined in the module `plain` draws a dot having 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, pair[] x, pair[] y, 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, path g, pen p=currentpen);
```

See [pathmarkers], page 96 and Section 7.8 [markers], page 80 for more general methods for 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, Label legend="",
          marker marker=nomarker);
```

4.2 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(pic,opic,origin);
```

The routine

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

fills the region exterior to the path g, out to the current boundary of picture pic.

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

In the second form, the elements of `z` are taken to be successive nodes of path `g`. The pens in `p` must belong to the same color space. Illustrations of Gouraud shading are provided in the example file `Gouraud.asy` and in the solid geometry module `solids.asy`. The edge flags used in Gouraud shading are documented here:

http://partners.adobe.com/public/developer/en/ps/sdk/TN5600.SmoothShading.pdf.

Tensor product shading using fillrule `fillrule` on patches bounded by the n cyclic paths of length 4 in path array `b`, using the vertex colors specified in the $n \times 4$ pen array `p` and internal control points in the $n \times 4$ array `z`, is implemented with

```
void tensorshade(picture pic=currentpicture, path g, pen fillrule=currentpen,
                 pen[][] p, path[] b=g, pair[][] z=new pair[][]);
```

If the array `z` is empty, Coons shading, in which the color control points are calculated automatically, is used. The pens in `p` must belong to the same color space. A simpler interface for the case of a single patch ($n = 1$) is also available:

```
void tensorshade(picture pic=currentpicture, path g, pen fillrule=currentpen,
                 pen[] p, path b=g, pair[] z=new pair[]);
```

Illustrations of tensor product and Coons shading are provided in the example files `tensor.asy`, `Coons.asy`, and `BezierSurface.asy`.

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

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

4.3 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 37). For an illustration of picture clipping, see the first example in Chapter 6 [LaTeX usage], page 74.

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

The text of a Label can be scaled, slanted, rotated, or shifted by multiplying it on the left by an affine transform (see Section 5.4 [Transforms], page 41). 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: `shift(10,0)*L`. The `embed` argument determines how the Label should transform with the embedding picture:

`Shift` only shift with embedding picture;

`Rotate` only shift and rotate with embedding picture (default);

`Rotate(pair z)`
rotate with (picture-transformed) vector `z`.

`Slant` only shift, rotate, slant, and reflect with embedding picture;

`Scale` shift, rotate, slant, reflect, and scale with embedding picture.

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 30) 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. These convenient abbreviations are predefined:

```
position BeginPoint=Relative(0);
position MidPoint=Relative(0.5);
position EndPoint=Relative(1);
```

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 graphic(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 (`draft=bool`) parameters. The `layer()` function can be used to force future objects to be drawn on top of the included image:

```
label(graphic("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.

One can prevent labels from overwriting one another with the `overwrite` pen attribute (see [overwrite], page 41).

The structure `object` defined in `plain_Label.asy` allows Labels and frames to be treated in a uniform manner. A group of objects may be packed together into single frame with the routine

```
frame pack(pair align=2S ... object inset[]);
```

The `string minipage(string s, width=100pt)` function can be used to format string `s` into a paragraph of width `width`. This example uses `minipage`, `clip`, and `graphic` to produce a CD label:




```
usepackage("babel","german");
size(11.7cm,11.7cm);

asy(nativeformat(),"logo");
fill(unitcircle^^(scale(2/11.7)*unitcircle),
     evenodd+rgb(124/255,205/255,124/255));

label(scale(1.1)*minipage(
                          "\centering\scriptsize \textbf{Nonlinear Modelling, Tutorial and
\textsc{G\"unther H. Mehring}\\
(edited by \textsc{Peter Sch\"opf} and \textsc{Jens Schwaiger})\\
with an \textbf{Appendix} written by\\
\textsc{Wolfgang Prager} and \textsc{Jens Schwaiger}",6cm),(0,0.6));
label(scale(1.1)*minipage("\centering\scriptsize Bericht Nr. 349(2005)\\
{\bfseries Grazer Mathematische Berichte}\\
ISSN 1016--7692",4cm),(0.55,0.2));
```

```
label(graphic("logo."+nativeformat(),"height=7cm"),(0,-0.22));
clip(unitcircle^^(scale(2/11.7)*unitcircle),evenodd);
```

5 Programming

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

```
// 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);
}
```

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.

5.1 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. The maximum integer is `intMax`.

`real` a real number; this should be set to the highest-precision native floating-point type on the architecture. The implicit initializer for reals is `0.0`. Real numbers have precision `realEpsilon`, with `realDigits` significant digits. The smallest positive real number is `realMin` and the largest positive real number is `realMax`.

`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 pairs 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);
```

Here `I` is the pair `(0,1)`. 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);
length(z);
```

returns 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, bool warn=true)`
: returns the angle of `z` in degrees in the interval [0,360) or `0` if `warn` is false and `z.x=z.y=0` (rather than producing an error);

`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`;

`pair realmult(pair z, pair w)`
: returns the element-by-element product `(z.x*w.x,z.y*w.y)`;

`real dot(pair z, pair w)`
: returns the dot product `z.x*w.x+z.y*w.y`;

`pair minbound(pair z, pair w)`
: returns `(min(z.x,w.x),min(z.y,w.y))`;

`pair maxbound(pair z, pair w)`
: returns `(max(z.x,w.x),max(z.y,w.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`. The implicit initializer for triples is `(0.0,0.0,0.0)`.

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, bool warn=true)`
: returns the longitude of `v` measured from the x axis in degrees; or `0` if `warn` is false and `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 u, triple v)`
: returns the dot product `u.x*v.x+u.y*v.y+u.z*v.z`;

`triple cross(triple u, triple v)`
: returns the cross product

```
(u.y*v.z-u.z*v.y,u.z*v.x-u.x*v.z,u.x*v.y-v.x*u.y);
```

`triple minbound(triple u, triple v)`
: returns `(min(u.x,v.x),min(u.y,v.y),min(u.z,v.z));`

`triple maxbound(triple u, triple v)`
: returns `(max(u.x,v.x),max(u.y,v.y),max(u.z,v.z))`.

`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 7.21 [babel], page 85):

  - `\"` 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 strings is the empty string `""`. Strings may be concatenated with the `+` operator. 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` using the current locale;

`string format(string s, real x)`
: returns a string containing `x` formatted according to the C-style format string `s` using the current locale (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 format="%a %b %d %T %Z %Y")`
: returns the current time formatted by the ANSI C routine `strftime` according to the string `format` using the current locale. Thus

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

: are equivalent ways of returning the current time in the default format used by the UNIX `date` command.

`int seconds(string t="", string format="")`
: returns the time measured in seconds after the Epoch (Thu Jan 01 00:00:00 UTC 1970) as determined by the ANSI C routine `strptime` according to the string `format` using the current locale, or the current time if `t` is the empty string. Note that the `"%Z"` extension to the POSIX `strptime` specification is ignored by the current GNU C Library. If an error occurs, the value -1 is returned. Here are some examples:

```
seconds("Mar 02 11:12:36 AM PST 2007","%b %d %r PST %Y");
seconds(time("%b %d %r %z %Y"),"%b %d %r %z %Y");
seconds(time("%b %d %r %Z %Y"),"%b %d %r "+time("%Z")+" %Y");
1+(seconds()-seconds("Jan 1","%b %d"))/(24*60*60);
```

: The last example returns today's ordinal date, measured from the beginning of the year.

`string time(int seconds, string format="%a %b %d %T %Z %Y")`
: returns the time corresponding to `seconds` seconds after the Epoch (Thu Jan 01 00:00:00 UTC 1970) formatted by the ANSI C routine `strftime` according to the string `format` using the current locale. For example, to return the date corresponding to 24 hours ago:

```
time(seconds()-24*60*60);
```

`string string(real x, int digits=realDigits)`
: casts `x` using precision `digits` to a string, using the C locale.

`abort(string s);`
: Aborts execution. If string `s` is nonempty, a diagnostic message constructed from the source file, line number, and `s` is printed.

As in C/C++, complicated types may be abbreviated with `typedef` (see the example in Section 5.11 [Functions], page 57).

5.2 Paths and guides

`path`
: a cubic spline resolved into a fixed path. The implicit initializer for paths is `nullpath`.

  The routine `circle(pair c, real r)`, which returns a Bezier curve approximating a circle of radius `r` centered on `c`, is based on unitcircle:

```
path 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`:

```
path Circle(pair c, real r, int n=400);
```

  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

```
path 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

```
path 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`:

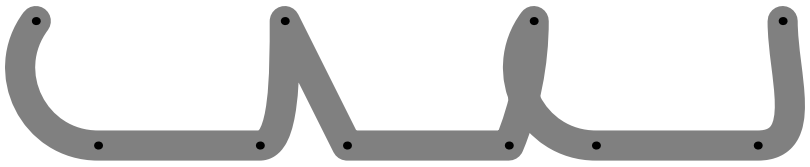
```
path Arc(pair c, real r, real angle1, real angle2,
         int n=400);
```

  An ellipse can be drawn with the routine

```
path ellipse(pair c, real a, real b)
{
  return shift(c)*xscale(a)*yscale(b)*unitcircle;
}
```

Here is an example of all five path connectors discussed in Chapter 3 [Tutorial], page 9:

```
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);
```



Here are some useful functions for paths:

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

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

`bool cyclic(path p);`
: returns `true` iff path `p` is cyclic.

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

`pair point(path p, int t);`
: If `p` is cyclic, return the coordinates of node `t` mod `length(p)`. Otherwise, return the coordinates of node `t`, unless `t < 0` (in which case `point(0)` is returned) or `t > 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-floor(t)` (see [Bezier], page 11). 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 p, int t, int sign=0);`
: If `sign < 0`, the direction (as a pair) of the incoming tangent to path `p` at node `t` is returned; if `sign > 0`, the direction of the outgoing tangent is returned. If `sign=0`, the mean of these two directions is returned. If `p` contains only one point, `(0,0)` is returned.

`pair dir(path p, real t);`
: This returns the direction of the tangent to path `p` at the point between node `floor(t)` and `floor(t)+1` corresponding to the cubic spline parameter `t-floor(t)` (see [Bezier], page 11). If `p` contains only one point, `(0,0)` is returned.

`pair precontrol(path p, int t);`
: returns the precontrol point of `p` at node `t`.

`pair precontrol(path p, real t);`
: returns the effective precontrol point of `p` at parameter `t`.

`pair postcontrol(path p, int t);`
: returns the postcontrol point of `p` at node `t`.

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

`real arclength(path p);`
: returns the length (in user coordinates) of the piecewise linear or cubic curve that path `p` represents.

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

`real dirtime(path p, pair z);`
: 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 the direction of pair `z`, or -1 if this never happens.

`real reltime(path p, real l);`
: returns the time on path `p` at the relative fraction `l` of its arclength.

`pair relpoint(path p, real l);`
: returns the point on path `p` at the relative fraction `l` of its arclength.

`pair midpoint(path p);`
: returns the point on path `p` at half of its arclength.

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

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

`path subpath(path p, real a, real b);`
: returns the subpath of `p` 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.

`real[] intersect(path p, path q, real fuzz=0);`
: If `p` and `q` have at least one intersection point, return a real array of length 2 containing the 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 a real array of length 0.

`pair intersectionpoint(path p, path q, real fuzz=0);`
: returns the intersection point `point(p,intersect(p,q,fuzz)[0])`.

`pair[] intersectionpoints(path p, path q);`
: returns an array containing all intersection points of the paths `p` and `q`.

`slice firstcut(path p, path q);`
: returns 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 {
  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.

`path buildcycle(... path[] p);`
: This returns the path surrounding a region bounded by a list of consecutively intersecting paths, following the behaviour of the `MetaPost buildcycle` command.

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

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

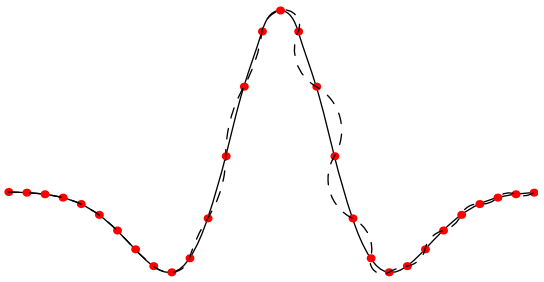
`int windingnumber(path p, pair z);`
: returns the winding number of the cyclic path `g` relative to the point `z`. The winding number is positive if the path encircles `z` in the counterclockwise direction.

`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 37).

`guide` an unresolved cubic spline (list of cubic-spline nodes and control points). The implicit initializer for a guide is `nullpath`; this is useful for building up a guide within a loop.

A guide is similar to a path except that 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. The solid curve in the following example is built up incrementally as a guide, but only resolved at drawing time; the dashed curve is incrementally resolved at each iteration, before the entire set of nodes (shown in red) is known:

```
size(200);

real mexican(real x) {return (1-8x^2)*exp(-(4x^2));}

int n=30;
real a=1.5;
real width=2a/n;

guide hat;
path solved;

for(int i=0; i < n; ++i) {
  real t=-a+i*width;
  pair z=(t,mexican(t));
  hat=hat..z;
  solved=solved..z;
}

draw(hat);
dot(hat,red);
draw(solved,dashed);
```



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.

The following routines can be used to examine the individual elements of a guide without actually resolving the guide to a fixed path (except for internal cycles, which are resolved):

`int size(guide g);`
: Analogous to `size(path g)`.

`int length(guide g);`
: Analogous to `length(path g)`.

`bool cyclic(path p);`
: Analogous to `cyclic(path g)`.

`pair point(guide g, int t);`
: Analogous to `point(path g, int t)`.

`guide reverse(guide g);`
: Analogous to `reverse(path g)`. If `g` is cyclic and also contains a secondary cycle, it is first solved to a path, then reversed. If `g` is not cyclic but contains an internal cycle, only the internal cycle is solved before reversal. If there are no internal cycles, the guide is reversed but not solved to a path.

`pair[] dirSpecifier(guide g, int i);`
: This returns a pair array of length 2 containing the outgoing (in element 0) and incoming (in element 1) direction specifiers (or `(0,0)` if none specified) for the segment of guide `g` between nodes `i` and `i+1`.

`pair[] controlSpecifier(guide g, int i);`
: If the segment of guide `g` between nodes `i` and `i+1` has explicit outgoing and incoming control points, they are returned as elements 0 and 1, respectively, of a two-element array. Otherwise, an empty array is returned.

`tensionSpecifier tensionSpecifier(guide g, int i);`
: This returns the tension specifier for the segment of guide `g` between nodes `i` and `i+1`. The individual components of the `tensionSpecifier` type can be accessed as the virtual members `in`, `out`, and `atLeast`.

`real[] curlSpecifier(guide g);`
: This returns an array containing the initial curl specifier (in element 0) and final curl specifier (in element 1) for guide `g`.

As a technical detail we note that a direction specifier given to `nullpath` modifies the node on the other side: the guides

```
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;
```

5.3 Pens

In `Asymptote`, pens provide a context for the four basic drawing commands (see Chapter 4 [Drawing commands], page 15). 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`. The implicit initializer for pens is `defaultpen`.

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).

`pen colorless(pen);`
: The function `colorless(pen)`; returns a copy of `pen p` with its color attributes stripped (to avoid color mixing).

The default color is `black`; this may be changed with the routine `defaultpen(pen)`. The function `colorspace(pen p)` returns the colorspace of pen `p` as a string (`"gray"`, `"rgb"`, `"cmyk"`, or `""`).

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.

Various shades and mixtures of the grayscale primary colors `black` and `white`, RGB primary colors `red`, `green`, and `blue`, and RGB secondary colors `cyan`, `magenta`, and `yellow` are defined as named colors, along with the CMYK primary colors `Cyan`, `Magenta`, `Yellow`, and `Black`, in the module `plain`:



The standard 140 RGB X11 colors can be imported with the command

```
import x11colors;
```

and the standard 68 CMYK TEX colors can be imported with the command

```
import texcolors;
```

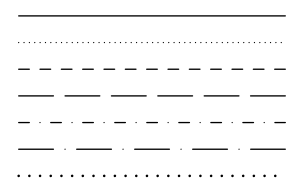
Note that there is some overlap between these two standards and the definitions of some colors (e.g. Green) actually disagree.

Asymptote also comes with a asycolors.sty LaTeX package that defines to LaTeX CMYK versions of Asymptote's predefined colors, so that they can be used directly within LaTeX strings. Normally, such colors are passed to LaTeX via a pen argument; however, to change the color of only a portion of a string, say for a slide presentation, (see Section 7.17 [slide], page 84) it may be desirable to specify the color directly to LaTeX. This file can be passed to LaTeX with the Asymptote command

```
usepackage("asycolors");
```

- Line types are specified with the function pen linetype(string s, real offset=0, bool scale=true, bool adjust=true), where s is a string of integer or real numbers separated by spaces. The optional parameter offset specifies where in the pattern to begin. 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. If adjust is true, 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");
pen Dotted=dotted+1.0;
pen Dotted(pen p=currentpen) {return dotted+2*linewidth(p);}
```



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). The line width of a pen is returned by real linewidth(pen p=currentpen);. For convenience, in the module plain 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);
  ```

  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. 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 lineskip=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 (this requires the fix-cm package available from

  http://www.ctan.org/tex-archive/help/Catalogue/entries/fix-cm

  and included in recent LaTeX distributions). The font size and line skip of a pen can be examined with the routines real fontsize(pen p=currentpen) and real lineskip(pen p=currentpen), respectively.
- 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)`. Support for standardized international characters is provided by the `unicode` package (see Section 7.19 [unicode], page 85).

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 nonstandard font command can be generated with `pen fontcommand(string)`.

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");
```

- The transparency of a pen can be changed with the command:

  ```
  pen opacity(real opacity=1, string blend="Compatible");
  ```

  The opacity can be varied from `0` (fully transparent) to the default value of `1` (opaque), and `blend` specifies one of the following foreground–background blending operations:

  ```
  "Compatible","Normal","Multiply","Screen","Overlay","SoftLight",
  "HardLight","ColorDodge","ColorBurn","Darken","Lighten","Difference",
  "Exclusion","Hue","Saturation","Color","Luminosity",
  ```

  as described in

  http://partners.adobe.com/public/developer/en/pdf/PDFReference16.pdf.

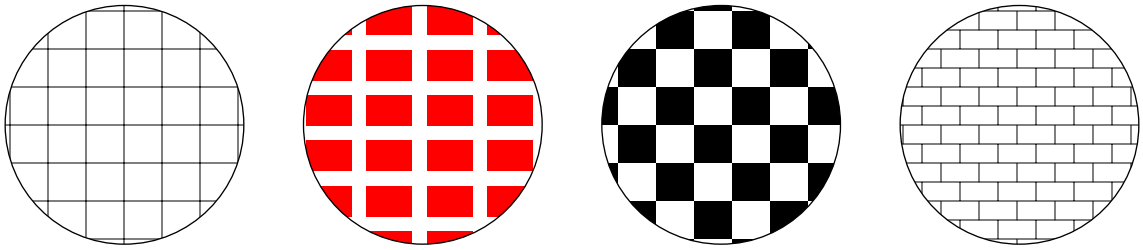
  Since `PostScript` does not support transparency, this feature is only effective with the `-f pdf` output format option; other formats can be produced from the resulting PDF file with the `ImageMagick convert` program. Labels are always drawn with an `opacity` of 1. A simple example of transparent filling is provided in the example file `transparency.asy`.

- `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)
  ```

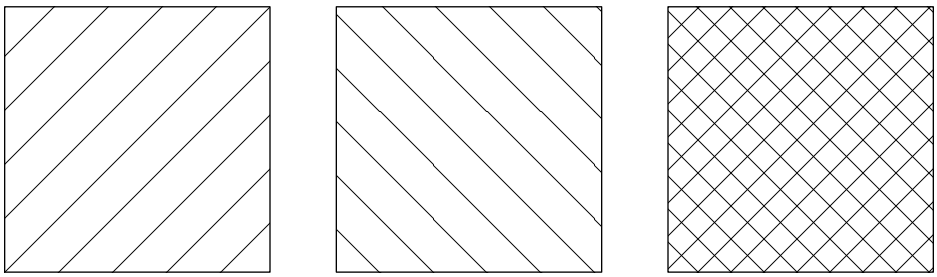
  To `fill` or `draw` using pattern `name`, use the pen `pattern("name")`. For example, rectangular tilings can be constructed using the routines `picture tile(real Hx=5mm, real Hy=0, pen p=currentpen, filltype filltype=NoFill)`, `picture checker(real Hx=5mm, real Hy=0, pen p=currentpen)`, and `picture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

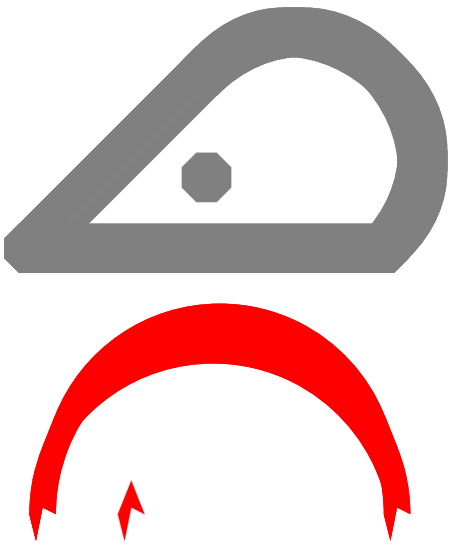


You may need to turn off aliasing in your PostScript viewer for patterns to appear correctly. Custom patterns can easily be constructed, following the examples in pattern.asy. The tiled pattern can even incorporate shading (see [gradient shading], page 17), as illustrated in this example (not included 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 specify a custom pen nib as an arbitrary polygonal path with pen makepen(path); this path represents the mark to be drawn for paths containing a single point. This pen nib path can be recovered from a pen with path nib(pen). Unlike in MetaPost, the path need not be convex:

```
size(200);
pen convex=makepen(scale(10)*polygon(8))+grey;
draw((1,0.4),convex);
draw((0,0)---(1,1)..(2,0)--cycle,convex);

pen nonconvex=scale(10)*
  makepen((0,0)--(0.25,-1)--(0.5,0.25)--(1,0)--(0.5,1.25)--cycle)+red;
draw((0.5,-1.5),nonconvex);
draw((0,-1.5)..(1,-0.5)..(2,-1.5),nonconvex);
```



The value `nullpath` represents a circular pen nib (the default); an elliptical pen can be achieved simply by multiplying the pen by a transform: `yscale(2)*currentpen`.

- 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.

5.4 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
```

This is equivalent to the `PostScript` transformation `[t.xx t.yx t.xy t.yy t.x t.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 28 for an example). Transforms can be composed with one another and inverted with the function `transform inverse(transform t)`; they can also be raised to any integer power with the `^` operator.

The built-in transforms are:

`transform identity();`
the identity transform;

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

`transform shift(real x, real y);`
translates by the pair `(x,y)`;

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

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

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

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

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

`transform reflect(pair a, pair b);`
reflects about the line `a--b`.

The implicit initializer for transforms is `identity()`. The routines `shift(transform t)` and `shiftless(transform t)` return the transforms `(t.x,t.y,0,0,0,0)` and `(0,0,t.xx,t.xy,t.yx,t.yy)` respectively.

5.5 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 `pair min(frame f)` and `pair 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 `align`, in a manner analogous to the `align` argument of `label` (see Section 4.4 [label], page 18), is returned by

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

`picture` Pictures are high-level structures (see Section 5.8 [Structures], page 51) defined in the module `plain` 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=x,
          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 make the user coordinates of picture pic represent multiples of x units in the x direction and y units in the y direction, use

```
void unitsize(picture pic=currentpicture, real x, real y=x);
```

When nonzero, these x and y values override the corresponding size parameters of picture pic.

The routine

```
void size(picture pic=currentpicture, real xsize, real ysize,
          pair min, pair max);
```

forces the final picture scaling to map the user coordinates box(min,max) to a region of width xsize and height ysize (when these parameters are nonzero).

Alternatively, calling the routine

```
transform fixedscaling(picture pic=currentpicture, pair min,
                       pair max, pen p=nullpen);
```

will cause picture pic to use a fixed scaling to map user coordinates in box(min,max) to the (already specified) picture size, taking account of the width of pen p. A warning will be issued if the final picture exceeds the specified size.

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=orientation,
             string format="", bool wait=NoWait, bool view=true);
void shipout(string prefix=defaultfilename,
             orientation orientation=orientation,
             string format="", bool wait=NoWait, bool view=true);
```

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 transformation that would currently be used to fit a picture pic to a frame is returned by the member function pic.calculateTransform().

In certain cases (e.g. 2D graphs) where only an approximate size estimate for pic is available, the picture fitting routine

```
frame pic.scale(real xsize=this.xsize, real ysize=this.ysize,
```

```
                bool keepAspect=this.keepAspect);
```

(which scales the resulting frame, including labels and fixed-size objects) will enforce perfect compliance with the requested size specification, but should not normally be required.

The default page orientation is `Portrait`; this may be modified by changing the variable `orientation`. To output in landscape mode, simply set `orientation=Landscape` or issue the command

```
shipout(Landscape);
```

To rotate the page by −90 degrees, use the orientation `Seascape`. To rotate the page by 180 degrees, use the orientation `UpsideDown`.

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:

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

`FillDraw(real xmargin=0, real ymargin=xmargin, pen p=nullpen)`
If `p` is `nullpen`, fill with the pen used to draw the boundary; otherwise fill with pen `p`. An optional margin of `xmargin` and `ymargin` can be specified.

`Fill` Fill with the drawing pen.

`Fill(real xmargin=0, real ymargin=xmargin, pen p=nullpen)`
If `p` is `nullpen`, fill with the drawing pen; otherwise fill with pen `p`. An optional margin of `xmargin` and `ymargin` can be specified.

`NoFill` Do not fill.

`Draw` Draw only the boundary.

`Draw(real xmargin=0, real ymargin=xmargin, pen p=nullpen)`
If `p` is `nullpen`, draw the boundary with the drawing pen; otherwise draw with pen `p`. An optional margin of `xmargin` and `ymargin` can be specified.

`UnFill` Clip the region.

`UnFill(real xmargin=0, real ymargin=xmargin`
Clip the region and surrounding margins `xmargin` and `ymargin`.

`RadialShade(pen penc, pen penr)`
Fill varying radially from `penc` at the center of the bounding box to `penr` at the edge.

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 functions

```
pair min(picture pic);
pair max(picture pic);
```

calculate the `PostScript` bounds that picture `pic` would have if it were currently fit to a frame using its default size specification.

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`, ignoring the contributions from fixed-size objects (such as labels and arrowheads), in the direction `dir` relative to its center.

The function

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

is identical to `point`, except that it also accounts for fixed-size objects, using the scaling transform that picture `pic` would have if currently fit to a frame using its default size specification.

The function

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

works similarly, except the result is a point in `PostScript` coordinates on the bounding box that picture `pic` would have if 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 10 [GUI], page 143), `filltype` requests optional background filling or clipping, and `put` specifies whether to add `src` above or below existing objects.

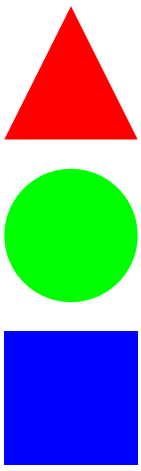
There are also routines to add a picture or frame `src` specified in postscript coordinates to another picture about the user coordinate `position`:

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

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

The optional `align` 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 4.4 [label], page 18). However, one key difference is that when `align` is not specified, labels are centered, whereas frames and pictures are aligned so that their origin is at `position`. Illustrations of frame alignment can be found in the examples [errorbars], page 97 and [image], page 113. 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(),(0,0),N);
add(pic,pic2.fit(),(0,0),10S);

add(pic.fit(),(0,0),N);
add(pic3.fit(),(0,0),10S);
```



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

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

```
path box(frame f, Label L="", real xmargin=0,
         real ymargin=xmargin, pen p=currentpen,
         filltype filltype=NoFill, bool put=Above);
path 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 module `plain`.

It is possible to insert verbatim `PostScript` commands in a picture with one of the routines

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

Here `min` and `max` can be used to specify explicit bounds associated with the resulting `PostScript` code.

Verbatim TEX commands can be inserted in the intermediate `LaTeX` output file with one of the functions

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

Here `min` and `max` can be used to specify explicit bounds associated with the resulting `TeX` code.

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);
```

The TEX environment can be reset to its initial state, clearing all macro definitions, with the function

```
void texreset();
```

The routine

```
void usepackage(string s, string options="");
```

provides a convenient abbreviation for

```
texpreamble("\usepackage["+options+"]{"+s+"}");
```

that can be used for importing LaTeX packages.

5.6 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 name, 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 first character of the string `commentchar` specifies a comment character. If this character is encountered in a data file, the remainder of the line is ignored. When reading strings, a comment character followed immediately by another comment character is treated as a single literal comment character.

If the `-globalwrite` (or `-unsafe`) option is enabled, one can change the current working directory to the contents of the string `s` with the function `string cd(string s)`, which returns the new working directory. If `string s` is empty, the path is reset to the value it had at program startup.

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 (treating the comment character as an ordinary character) and return it as a string.

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

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

If `update=false`, any existing data in the file will be erased and only write operations can be used on the file. If `update=true`, existing data will be preserved, the position will be set to the end-of-file, and both reading and writing operations will be enabled. For security reasons, writing to files in directories other than the current directory is allowed only if the `-globalwrite` (or `-unsafe`) command-line option is specified.

There are two special files: `stdin`, which reads from the keyboard, and `stdout`, which writes to the terminal. The implicit initializer for files is `null`.

Data of a built-in type `T` can be written to an output file by calling one of the functions

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

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

If `file` is not specified, `stdout` is used and terminated by default with a newline. If specified, the optional identifying string `s` is written before the data `x`. An arbitrary number of data values may be listed when writing scalars or one-dimensional arrays. The `suffix` may be one of the following: `none` (do nothing), `endl` (terminate with a newline), or `tab` (terminate with a tab). Here are some simple examples 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"
```

A file may also be opened with `xinput` or `xoutput`, instead of `input` or `output`, to read or write double precision values in Sun Microsystem's XDR (External Data Representation) portable binary format (available on all UNIX platforms). A file may also be opened with `binput` or `boutput` to read or write double precision values in the native (nonportable) machine binary format. The function `file single(file)` may be used to set a file to read single precision XDR or binary values; calling `file single(file,false)` sets it back to read doubles again.

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 function `int tell(file)` returns the current position in a file relative to the beginning. The function `seek(file file, int pos)` can be used to change this position, where a negative value for the position `pos` is interpreted as relative to the end-of-file. For example, one can rewind a file `file` with the command `seek(file,0)` and position to the final character in the file with `seek(file,-1)`. The command `seekeof(file)` sets the position to the end of the file.

Assigning `settings.scroll=n` for a positive integer `n` requests a pause after every `n` output lines to `stdout`. One may then press `Enter` to continue or `q` followed by `Enter` to quit. If `n` is negative, the output scrolls a page at a time (i.e. by one less than the current number of display lines). The default value, `settings.scroll=0`, specifies continuous scrolling.

The routines

```
string getstring(string name="", string default="", string prompt="",
                 bool store=true);
int getint(string name="", int default=0, string prompt="",
           bool store=true);
real getreal(string name="", real default=0, string prompt="",
             bool store=true);
pair getpair(string name="", pair default=0, string prompt="",
             bool store=true)
```

defined in the module `plain` may be used to prompt for a value from `stdin` using the GNU `readline` library. If `store=true`, the history of values for `name` is stored in the file `".asy_history"+name` (see [history], page 141). The most recent value in the history will be used to provide a default value for subsequent runs. The default value (initially `default`) is displayed after `prompt`. These functions are based on the internal routines

```
string readline(string prompt="", string name="", bool tabcompletion=false);
void saveline(string name, string value, bool store=true);
```

Here, `readline` prompts the user with the default value formatted according to `prompt`, while `saveline` is used to save the string `value` in a local history named `name`, optionally storing the local history in a file `".asy_history_"+name`.

The routine `history(string name, int n=1)` can be used to look up the `n` most recent values (or all values up to `historylines` if `n=0`) entered for string `name`.

5.7 Variable initializers

A variable can be assigned a value when it is declared, as in `int x=3;` where the variable `x` is assigned the value `3`. As well as literal constants such as `3`, arbitary expressions can be used as initializers, as in `real x=2*sin(pi/2);`.

A variable is not added to the namespace until after the initializer is evaluated, so for example, in

```
int x=2;
int x=5*x;
```

the `x` in the initializer on the second line refers to the variable `x` declared on the first line. The second line, then, declares a variable `x` shadowing the original `x` and initializes it to the value `10`.

Variables of most types can be declared without an explicit initializer and they will be initialized by the default initializer of that type:

- Variables of the numeric types `int`, `real`, and `pair` are all initialized to zero; variables of type `triple` are initialized to `O=(0,0,0)`.
- `boolean` variables are initialized to `false`.
- `string` variables are initialized to the empty string.
- `transform` variables are initialized to the identity transformation.
- `path` and `guide` variables are initialized to `nullpath`.
- `pen` variables are initialized to the default pen.
- `frame` and `picture` variables are initialized to empty frames and pictures, respectively.
- `file` variables are initialized to `null`.

The default initializers for user-defined array, structure, and function types are explained in their respective sections. Some types, such as `code`, do not have default initializers. When a variable of such a type is introduced, the user must initialize it by explicitly giving it a value.

The default initializer for any type `T` can be redeclared by defining the function `T operator init()`. For instance, `int` variables are usually initialized to zero, but in

```
int operator init() {
  return 3;
}
int y;
```

the variable `y` is initialized to `3`. This example was given for illustrative purposes; redeclaring the initializers of built-in types is not recommended. Typically, `operator init` is used to define sensible defaults for user-defined types.

5.8 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 `public` (may be read and modified anywhere in the code), but may be optionally declared `restricted` (readable anywhere but writeable only inside the structure where they are defined) or `private` (readable and writable only inside the structure). In a structure definition, the keyword `this` can be used as an expression to refer to the enclosing structure. Any code at the top-level scope within the structure is executed on initialization.

Variables hold references to structures. That is, in the example:

```
struct T {
  int x;
}

T foo=new T;
T bar=foo;
bar.x=5;
```

The variable `foo` holds a reference to an instance of the structure `T`. When `bar` is assigned the value of `foo`, it too now holds a reference to the same instance as `foo` does. The assignment `bar.x=5` changes the value of the field `x` in that instance, so that `foo.x` will also be equal to `5`.

The expression `new T` creates a new instance of the structure `T` and returns a reference to that instance. In creating the new instance, any code in the body of the record definition is executed. For example:

```
int Tcount=0;
struct T {
  int x;
  ++Tcount;
}

T foo=new T;
```

Here, the expression `new T` will produce a new instance of the class, but will also cause `Tcount` to be incremented, so that it keeps track of the number of instances produced.

The expression `null` can be cast to any structure type to yield a null reference, a reference that does not actually refer to any instance of the structure. Trying to use a field of a null reference will cause an error.

The function `bool alias(T,T)` checks to see if two structure references refer to the same instance of the structure (or both to `null`). For example, in the example code at the start

of the section, `alias(foo,bar)` would return true, but `alias(foo,new T)` would return false, as `new T` creates a new instance of the structure `T`. The boolean operators `==` and `!=` are by default equivalent to `alias` and `!alias` respectively, but may be overwritten for a particular type (for example, to do a deep comparison).

After the definition of a structure `T`, a variable of type `T` is initialized to a new instance (`new T`) by default. During the definition of the structure, however, variables of type `T` are initialized to `null` by default. This special behaviour is to avoid infinite recursion of creating new instances in code such as

```
struct tree {
  int value;
  tree left;
  tree right;
}
```

Here is a simple example that illustrates the use of structures:

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

S s;                              // Initializes s with new S;

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
```

It is often convenient to have functions that construct new instances of a structure. Say we have a `Person` structure:

```
struct Person {
  string firstname;
  string lastname;
}

Person joe=new Person;
joe.firstname="Joe";
joe.lastname="Jones";
```

Creating a new Person is a chore; it takes three lines to create a new instance and to initialize its fields (that's still considerably less effort than creating a new person in real life, though).

We can reduce the work by defining a constructor function `Person(string,string)`:

```
struct Person {
  string firstname;
  string lastname;

  static Person Person(string firstname, string lastname) {
    Person p=new Person;
    p.firstname=firstname;
    p.lastname=lastname;
    return p;
  }
}

Person joe=Person.Person("Joe", "Jones");
```

While it is now easier than before to create a new instance, we still have to refer to the constructor by the qualified name `Person.Person`. If we add the line

```
from Person unravel Person;
```

immediately after the structure definition, then the constructor can be used without qualification: `Person joe=Person("Joe", "Jones");`.

The constructor is now easy to use, but it is quite a hassle to define. If you write a lot of constructors, you will find that you are repeating a lot of code in each of them. Fortunately, your friendly neighbourhood Asymptote developers have devised a way to automate much of the process.

If, in the body of a structure, Asymptote encounters the definition of a function of the form `void operator init(`*`args`*`)`, it implicitly defines a constructor function of the arguments *`args`* that uses the `void operator init` function to initialize a new instance of the structure. That is, it essentially defines the following constructor (assuming the structure is called `Foo`):

```
static Foo Foo(args) {
  Foo instance=new Foo;
  instance.operator init(args);
  return instance;
}
```

This constructor is also implicitly copied to the enclosing scope after the end of the structure definition, so that it can used subsequently without qualifying it by the structure name. Our `Person` example can thus be implemented as:

```
struct Person {
  string firstname;
  string lastname;

  void operator init(string firstname, string lastname) {
    this.firstname=firstname;
    this.lastname=lastname;
  }
}
```

```
Person joe=Person("Joe", "Jones");
```

The use of `operator init` to implicitly define constructors should not be confused with its use to define default values for variables (see Section 5.7 [Variable initializers], page 50). Indeed, in the first case, the return type of the `operator init` must be `void` while in the second, it must be the (non-`void`) type of the variable.

The function `cputime()` returns a structure `cputime` with cumulative CPU times broken down into the fields `parent.user`, `parent.system`, `child.user`, and `child.system`. For convenience, the incremental fields `change.user` and `change.system` indicate the change in the corresponding total parent and child CPU times since the last call to `cputime()`. The function

```
void write(file file=stdout, string s="", cputime c,
           string format=cputimeformat, suffix suffix=none);
```

displays the incremental user cputime followed by "u", the incremental system cputime followed by "s", the total user cputime followed by "U", and the total system cputime followed by "S".

Much like in C++, casting (see Section 5.13 [Casts], page 68) provides for an elegant implementation of structure inheritance, including virtual functions:

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

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;}

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;
```

For further examples of structures, see `Legend` and `picture` in the `Asymptote` base module `plain`.

5.9 Operators

5.9.1 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 (with conditional evaluation of right-hand argument)

`&` and

`||` or (with conditional evaluation of right-hand argument)

`|` or

`^` xor

`!` not

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

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

The function `T interp(T a, T b, real t)` returns `a+t*(b-a)` for all built-in arithmetic types `T`.

`Asymptote` also defines bitwise functions `int AND(int,int)`, `int OR(int,int)`, `int XOR(int,int)`, and `int NOT(int)`.

5.9.2 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.

5.9.3 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=",g);
```

5.10 Implicit scaling

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

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

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

This produces the output

```
6
5
6
-3.204e-19
(1,1)
8
10
18
2.72789228047704
2.48046543129542
283.464008929116
```

5.11 Functions

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

Functions arguments are passed by value. To pass an argument by reference, simply enclose it in a structure (see Section 5.8 [Structures], page 51).

Here are some significant features of `Asymptote` functions:

1. Variables with signatures (functions) and without signatures (nonfunction variables) are distinct:

   ```
   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. One may redefine a function `f`, even for calls to `f` in previously declared functions, by assigning another (anonymous or named) function to it. However, if `f` is overloaded by a new function definition, previous calls will still access the original version of `f`, as illustrated in this example:

```
void f() {
  write("hi");
}

void g() {
  f();
}

g(); // writes "hi"

f=new void() {write("bye");};

g(); // writes "bye"

void f() {write("overloaded");};

f(); // writes "overloaded"
g(); // writes "bye"
```

6. 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); // Writes true, then writes false.
```

`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 a fully working version of the GNU library `libsigsegv` (e.g. 2.4 or later) is installed at configuration time).

5.11.1 Default arguments

`Asymptote` supports a more flexible mechanism for default function arguments than C++: they may appear anywhere in the function prototype. Because certain data types are implicitly cast to more sophisticated types (see Section 5.13 [Casts], page 68) 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.

5.11.2 Named arguments

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

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

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

outputs 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.

5.11.3 Rest arguments

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

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

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

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

```
    start -= subs[i];
  return start;
}

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

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

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

One can even combine normal arguments with rest arguments:

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

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

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

is illegal, as the start formal is not matched.

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

The overloading resolution in `Asymptote` is similar to the function matching rules used in C++. Every argument match is given a score. Exact matches score better than matches with casting, and matches with formals (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
```

5.11.4 Mathematical functions

`Asymptote` has built-in versions of the standard `libm` mathematical real(real) functions `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log`, `pow10`, `log10`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`, `sqrt`, `cbrt`, `fabs`, as well as the identity function `identity`. `Asymptote` also defines the 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 `degrees(real radians)` and `radians(real degrees)` can be used to convert between radians and degrees. The function `Degrees(real radians)` returns the angle in degrees in the interval [0,360). For convenience, `Asymptote` defines variants `Sin`, `Cos`, `Tan`, `aSin`, `aCos`, and `aTan` of the standard trigonometric functions using degrees rather than radians.

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`.

When configured with the GNU Scientific Library (GSL), available from http://www.gnu.org/software/gsl/, `Asymptote` contains an internal module `gsl` that defines the airy functions `Ai(real)`, `Bi(real)`, `Ai_deriv(real)`, `Bi_deriv(real)`, `zero_Ai(int)`, `zero_Bi(int)`, `zero_Ai_deriv(int)`, `zero_Bi_deriv(int)`, the Bessel functions `I(int, real)`, `K(int, real)`, `j(int, real)`, `y(int, real)`, `i_scaled(int, real)`, `k_scaled(int, real)`, `J(real, real)`, `Y(real, real)`, `I(real, real)`, `K(real, real)`, `zero_J(real, int)`, the elliptic functions `F(real, real)`, `E(real, real)`, and `P(real, real)`, the exponential/trigonometric integrals `Ei`, `Si`, and `Ci`, the Legendre polynomials `Pl(int, real)`, and the Riemann zeta function `zeta(real)`. For example, to compute the sine integral `Si` of 1.0:

```
import gsl;
write(Si(1.0));
```

5.12 Arrays

Appending `[]` to a built-in or user-defined type yields an array. The array element `i` of an array `A` can be accessed as `A[i]`. By default, attempts to access or assign to an array

element using a negative index generates an error. Reading an array element with an index beyond the length of the array also generates an error; however, assignment to an element beyond the length of the array causes the array to be resized to accommodate the new element. One can also index an array `A` with an integer array `B`: the array `A[B]` is 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 b)`, `bool cyclicflag`, `T push(T x)`, `void append(T[] a)`, `T pop()`, `void insert(int i ... T[] x)`, `void delete(int i, int j=i)`, `void delete()`, and `bool initialized(int n)`. 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, while `A.insert(int i ... T[] x)` inserts `x` into the array at index `i`. For convenience `A.push` returns the pushed item. The function `A.pop()` pops and returns the last element, while `A.delete(int i, int j=i)` deletes elements with indices in the range [i,j], shifting the position of all higher-indexed elements down. If no arguments are given, `A.delete()` provides a convenient way of deleting all elements of `A`. The routine `A.initialized(int n)` can be used to examine whether the element at index `n` is initialized. Like all `Asymptote` functions, `cyclic`, `push`, `append`, `pop`, `insert`, `delete`, and `initialized` 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.
A.delete(0);        // A now contains {2,1,2}.
A.delete(0,1);        // A now contains {2}.
A.insert(1,3);      // A now contains {2,3}.
A.insert(1 ... A); // A now contains {2,2,3,3}
A.insert(2,4,5);   // A now contains {2,2,4,5,3,3}.
```

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[]`.

In the following list of built-in array functions, `T` represents a generic type. Note that the internal functions `alias`, `copy`, `concat`, `sequence`, `map`, and `transpose`, which depend on type `T[]`, are defined only after the first declaration of a variable of type `T[]`.

`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[]`. These `n` array elements are not initialized unless they are arrays themselves (in which case they are each initialized to empty arrays).

`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), 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[] complement(int[] a, int n)`
: returns the complement of the integer array `a` in `{1,2,...,n}`, so that `b[complement(a,b.length)]` yields the complement of `b[a]`.

`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[] a, T key)`
: For built-in ordered types `T`, searches a sorted ordered array `a` 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[][] copy(T[][] a)`
: returns a deep copy of the array `a`;

`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 built-in ordered types `T`, returns a copy of `a` sorted in ascending order;

`T[][] sort(T[][] a)`
: For built-in 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"}};
// 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[][][] transpose(T[][][] a, int[] perm)`
: returns the 3D transpose of `a` obtained by applying the permutation `perm` of `new int[]{0,1,2}` to the indices of each entry.

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

`T min(T[] a)`
`T min(T[][] a)`
`T min(T[][][] a)`
: For built-in ordered types `T`, returns the minimum element of `a`.

`T max(T[] a)`
`T max(T[][] a)`
`T max(T[][][] a)`
: For built-in 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 built-in 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 built-in 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=1)`
: 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 $L\mathtt{x} = \mathtt{f}$ and return the solution `x`, 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[] solve(real[][] a, real[] b)`
: Solve the linear equation $\mathtt{a}x = \mathtt{b}$ by LU decomposition and return 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 66).

`real[][] solve(real[][] a, real[][] b)`
: Solve the linear equation $\mathtt{a}x = \mathtt{b}$ and return the solution x, where `a` is an $n \times n$ matrix and `b` is an $n \times m$ matrix.

`real[][] identity(int n);`
: returns an $n \times n$ identity matrix.

`real[][] inverse(real[][] a)`
: returns the inverse of an $n \times n$ matrix `a`.

`real[][] multdiagonal(real[][] a, real[] b)`
: returns `a*diagonal(b)`.

`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 $ax^3 + bx^2 + cx + d = 0$.

`real newton(int iterations=100, real f(real), real fprime(real),real x, bool verbose=false);`
: Use Newton-Raphson iteration to solve for a root of a real-valued differentiable function `f`, given its derivative `fprime` and an initial guess `x`. Diagnostics for each iteration are printed if `verbose=true`. If the iteration fails after the maximum allowed number of loops (`iterations`), `realMax` is returned.

`real newton(int iterations=100, real f(real), real fprime(real),real x1, real x2, bool verbose=false);`
: Use bracketed Newton-Raphson bisection to solve for a root of a real-valued differentiable function `f` within an interval [`x1`,`x2`] (on which the endpoint values of `f` have opposite signs), given its derivative `fprime`. Diagnostics for each iteration are printed if `verbose=true`. If the iteration fails after the maximum allowed number of loops (`iterations`), `realMax` is returned.

`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);
```

Since string reads by default read up to the end of line anyway, line mode normally has no effect on string array reads. However, there is a white-space delimiter mode for reading strings, set with `word(file)` and cleared with `word(file,false)`, which causes string reads to respect white-space delimiters, instead of the default end-of-line delimiter:

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

Another useful mode is comma-separated-value mode, set with `csv(file)` and cleared with `csv(file,false)`, which causes reads to respect comma delimiters:

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

5.13 Casts

`Asymptote` implicitly casts `int` to `real`, `int` to `pair`, `real` to `pair`, `pair` to `path`, `pair` to `guide`, `path` to `guide`, `guide` to `path`, `real` to `pen`, `pair[]` to `guide[]`, `pair[]` to `path[]`, `path` to `path[]`, and `guide` to `path[]`, along with various three-dimensional casts defined in `three.asy`. Implicit casts are automatically attempted on assignment and 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 {
  real radius;
  real angle;
}

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`.

5.14 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;
```

Except for a few built-in modules, such as `settings`, 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 2.5 [Search paths], page 5) 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 import an `Asymptote` module determined by the string `s` at runtime like this:

```
eval("import "+s,true);
```

To conditionally execute an array of asy files, use

```
void asy(string format, bool overwrite ... string[] s);
```

The file will only be processed, using output format `format`, if overwrite is `true` or the output file is missing.

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

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

It is not necessary to terminate the string `s` with a semicolon. If `embedded` is `true`, the string will be evaluated at the top level of the current environment. If `embedded` is `false` (the default), the string will be evaluated in an independent environment, sharing the same `settings` module (see [settings], page 139).

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";
```

To list all global functions and variables defined in a module named by the contents of the string `s`, use the function

```
void list(string s, bool imports=false);
```

Imported global functions and variables are also listed if `imports` is `true`.

5.15 Static

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

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 to 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;
    }
  }
  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.

Loop constructs allocate a new frame in every iteration. This is so that higher-order functions can refer to variables of a specific iteration of a loop:

```
void f();
for(int i=0; i < 10; ++i) {
  int x=i;
  if(x==5) {
    f=new void () { write(x); }
  }
}
f();
```

Here, every iteration of the loop has its own variable `x`, so `f()` will write `5`. If a variable in a loop is declared static, it will be allocated where the enclosing function or structure was defined (just as if it were declared static outside of the loop). For instance, in:

```
void f() {
  static int x;
  for(int i=0; i < 10; ++i) {
    static int y;
  }
}
```

both `x` and `y` will be allocated in the same place, which is also where `f` is also allocated.

Statements may also be declared static, in which case they are run at the place where the enclosing function or structure is defined. Declarations or statements not enclosed in a function or structure definition are already at the top level, so static modifiers are meaningless. A warning is given in such a case.

Since structures can have static fields, it is not always clear for a qualified name whether the qualifier is a variable or a type. For instance, in:

```
struct A {
  static int x;
}
pair A;

int y=A.x;
```

does the `A` in `A.x` refer to the structure or to the pair variable. It is the convention in Asymptote that, if there is a non-function variable with the same name as the qualifier, the qualifier refers to that variable, and not to the type. This is regardless of what fields the variable actually possesses.

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
```

or

```
pdflatex latexusage
asy latexusage
pdflatex latexusage
```

If the inline option is given to the asymptote.sty package, 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 if latex is used with the inline option, the labels might not show up in DVI viewers that cannot handle raw PostScript code. One can 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).

An excellent tutorial by Dario Teixeira on integrating Asymptote and LaTeX is available at http://dario.dse.nl/projects/asylatex/.

Here now is latexusage.tex:

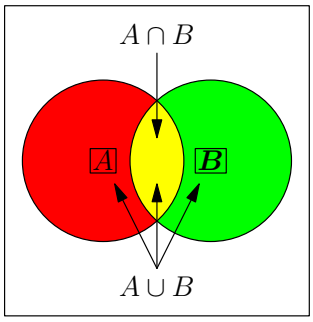
```
\documentclass[12pt]{article}

% Use this form to include eps (latex) or pdf (pdflatex) files:
\usepackage{asymptote}

% Use this form with latex or pdflatex to include inline LaTeX code:
%\usepackage[inline]{asymptote}

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

% Enable this line to produce pdf hyperlinks with pdflatex:
%\usepackage[pdftex]{hyperref}

\begin{document}
```

```
\begin{asydef}
// Global Asymptote definitions can be put here.
usepackage("bm");
\end{asydef}

Here is a venn diagram produced with Asymptote, drawn to width 4cm:

\def\A{A}
\def\B{\bm{B}}

%\begin{figure}
\begin{center}
\begin{asy}
size(4cm,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("$\bm{B}$",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);
```

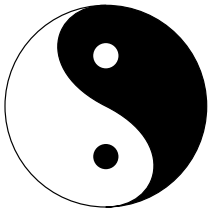
```
draw(z--z1,Arrow,Margin(0,m));
draw(z--z2,Arrow,Margin(0,m));

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}

To scale the figure to the full line width:

\begin{center}
\begin{asy}[\the\linewidth]
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}
```

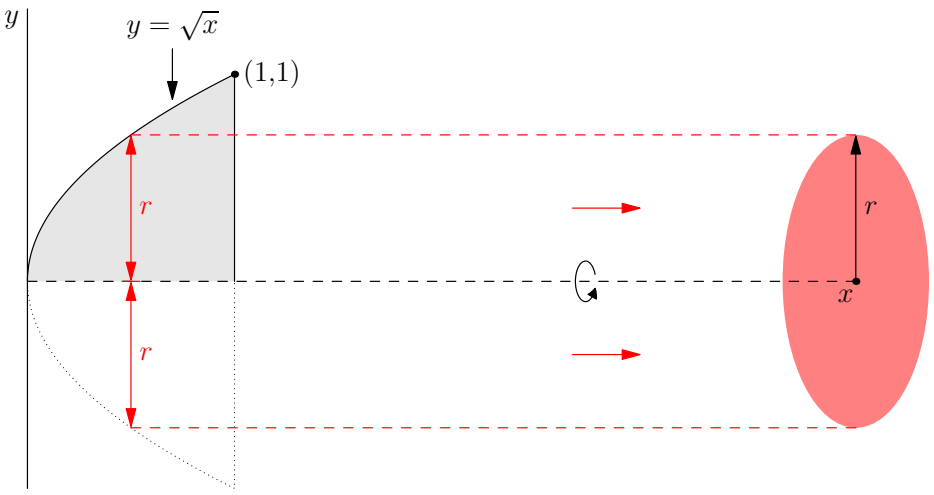
Here is a venn diagram produced with Asymptote, drawn to width 4cm:



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



To scale the figure to the full line width:



7 Base modules

`Asymptote` currently ships with the following base modules:

7.1 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 `-noautoplain` command-line option to disable this feature.

7.2 simplex

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

7.3 math

This package extends `Asymptote`'s mathematical capabilities with point-in-polygon and intersection algorithms and matrix arithmetic.

Unlike `MetaPost`, `Asymptote` does not implicitly solve linear equations and therefore does not have the notion of a `whatever` unknown. Use the built-in explicit linear equation solver `solve` instead. 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);
```

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)`.

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

`real[][] diagonal(... real[] a)`
: returns a diagonal matrix with diagonal entries given by a.

7.4 interpolate

This module implements Lagrange, Hermite, and standard cubic spline interpolation in `Asymptote`, as illustrated in the example `interpolate1.asy`.

7.5 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.

7.6 stats

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

7.7 patterns

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

7.8 markers

This package implements specialized routines for marking paths and angles. The principal mark routine provided by this package is

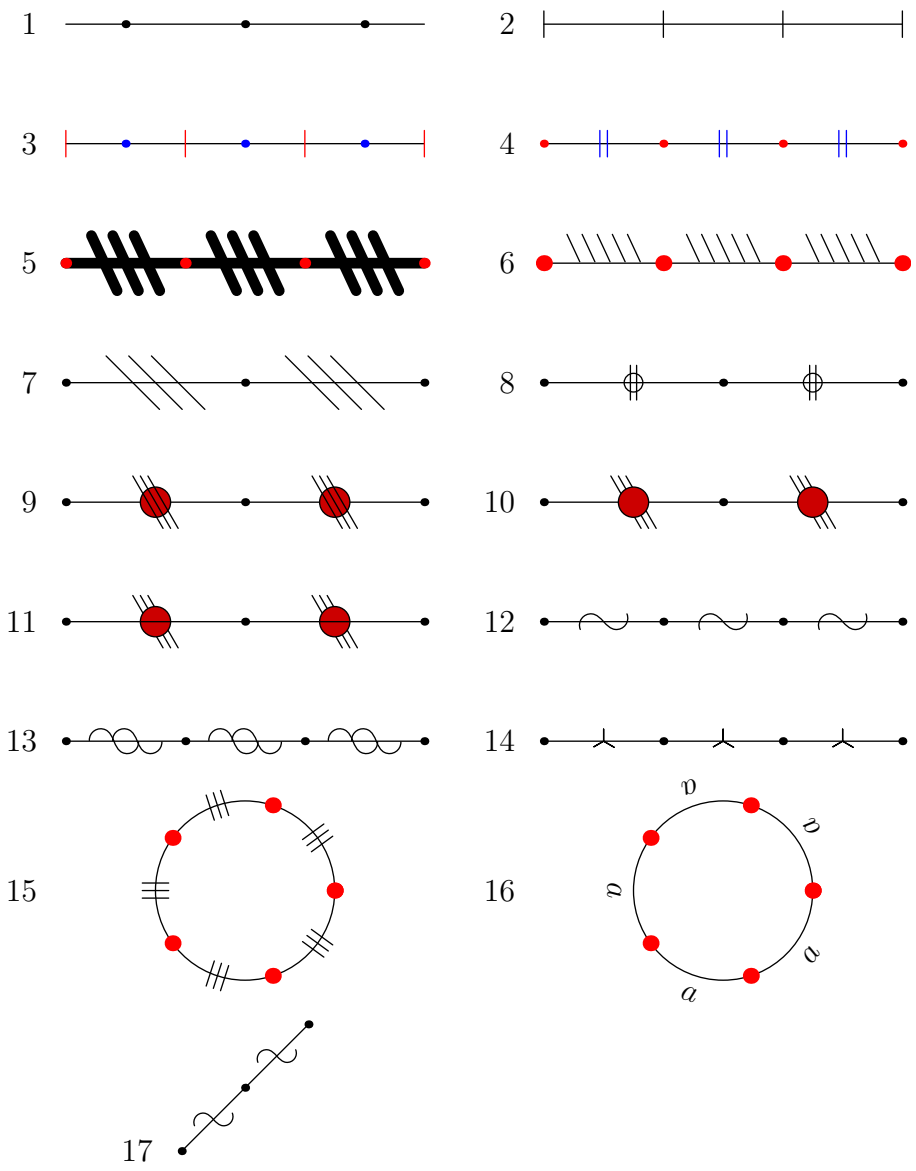
```
markroutine markinterval(int n=1, frame f, bool rotated=false);
```

which centers `n` copies of frame `f` within uniformly space intervals in arclength along the path, optionally rotated by the angle of the local tangent.

The `marker` (see [marker], page 96) routine can be used to construct new markers from these predefined frames:

```
frame stickframe(int n=1, real size=0, pair space=0, real angle=0,
                 pair offset=0, pen p=currentpen);

frame circlebarframe(int n=1, real barsize=0,
                     real radius=0,real angle=0,
                     pair offset=0, pen p=currentpen,
                     filltype filltype=NoFill, bool above=false);

frame crossframe(int n=3, real size=0, pair space=0,
                 real angle=0, pair offset=0, pen p=currentpen);

frame tildeframe(int n=1, real size=0, pair space=0,
                 real angle=0, pair offset=0, pen p=currentpen);
```

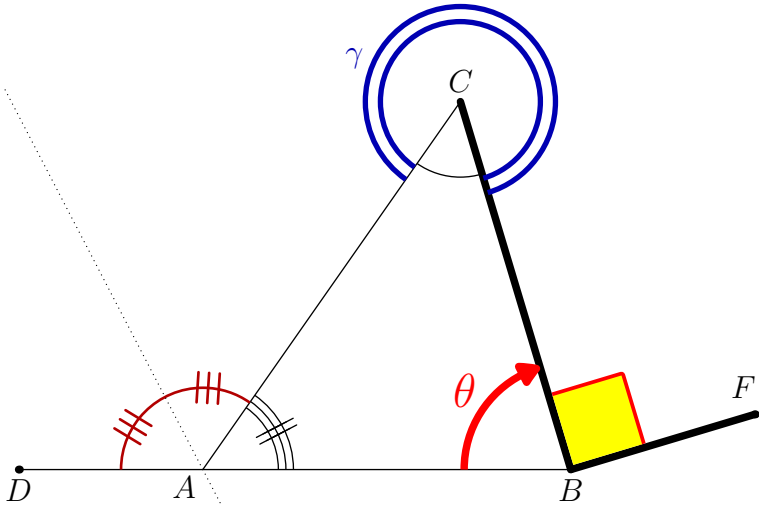
For convenience, this module also constructs the markers `StickIntervalMarker`, `CrossIntervalMarker`, `CircleBarIntervalMarker`, and `TildeIntervalMarker` from the above frames. The example `markers1.asy` illustrates the use of these markers:



This package also provides a routine for marking an angle AOB:

```
void markangle(picture pic=currentpicture, Label L="",
               int n=1, real radius=0, real space=0,
               pair A, pair O, pair B, arrowbar arrow=None,
               pen p=currentpen, margin margin=NoMargin,
               marker marker=nomarker);
```

as illustrated in the example `markers2.asy`.



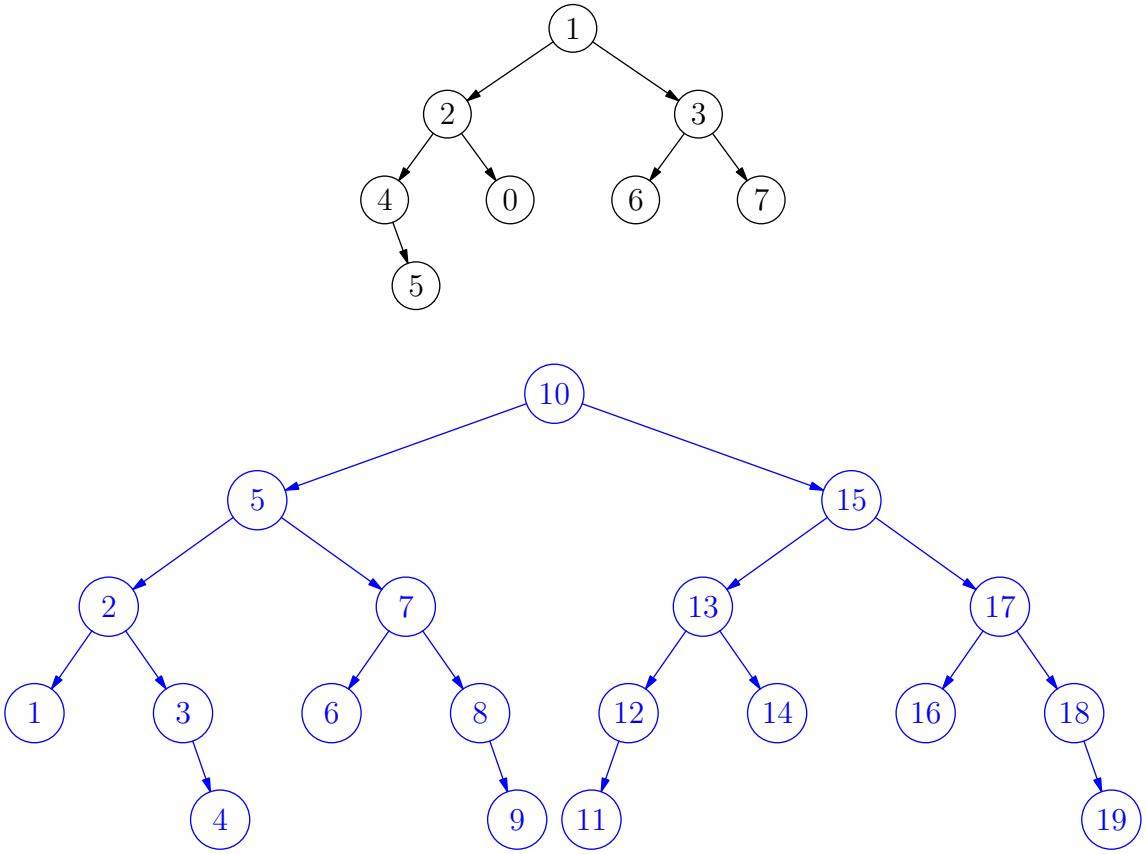
7.9 tree

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

7.10 binarytree

This module can be used to draw an arbitrary binary tree and includes an input routine for the special case of a binary search tree, as illustrated in the example `binarytreetest.asy`:

```
import binarytree;

picture pic,pic2;

binarytree bt=binarytree(1,2,4,nil,5,nil,nil,0,nil,nil,3,6,nil,nil,7);
draw(pic,bt);

binarytree st=searchtree(10,5,2,1,3,4,7,6,8,9,15,13,12,11,14,17,16,18,19);
draw(pic2,st,blue);

add(pic.fit(),(0,0),10N);
add(pic2.fit(),(0,0),10S);
```



7.11 drawtree

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

7.12 syzygy

This module automates the drawing of braids, relations, and syzygies, along with the corresponding equations, as illustrated in the example `knots.asy`.

7.13 feynman

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

7.14 roundedpath

This package, contributed by Stefan Knorr, is useful for rounding the sharp corners of paths, as illustrated in the example file `roundpath.asy`.

7.15 animate

This module allows one to generate animations, as illustrated by the files `wheel.asy`, `wavelet.asy`, and `cube.asy` in the `animations` subdirectory of the examples directory.

These animations use the ImageMagick convert program to merge multiple images into a GIF or MPEG movie.

Higher-quality portable clickable PDF movies, with optional controls, can be internally generated and embedded by Asymptote. This requires installing the package

http://www.ctan.org/tex-archive/macros/latex/contrib/animate/animate.sty

(version 2007/05/24 or later) in a new directory animate in the LaTeX source directory (for example, in /usr/share/texmf/tex/latex/animate). On UNIX systems, one must then execute the command texhash.

The example pdfmovie.asy in the animations directory, along with the slide presentations slidemovies.asy and intro.asy illustrate the use of embedded PDF movies. The member function

```
string pdf(real delay=animationdelay, string options="")
```

of the animate structure accepts any of the animate.sty options, as described here:

http://www.ctan.org/tex-archive/macros/latex/contrib/animate/doc/animate.pdf

7.16 embed

This module provides an interface to the LaTeX package (included with MikTeX)

http://www.tug.org/tex-archive/macros/latex/contrib/movie15

for embedding movies, sounds, and 3D objects into a PDF document. The latest version of the movie15 package requires both pdflatex version 1.20 or later and the file

http://www.tug.org/tex-archive/macros/latex/contrib/oberdiek/ifdraft.dtx

which can be installed by placing it in a directory ifdraft in the LaTeX source directory (e.g. /usr/share/texmf/tex/latex/ifdraft) and executing in that directory the commands:

```
tex ifdraft.dtx
texhash
```

An example of embedding U3D code is provided in the file embeddedu3d.asy. As of version 7.0.8, Adobe Reader now supports the U3D format under Linux.

Unfortunately, Adobe has not yet made available an embedded movie plugin for the Linux version of Adobe Reader. A portable method for embedding movie files, which should work on any platform and does not require the movie15 or ifdraft packages, is provided by using the external module instead of embed. An example of these interfaces is provided in the file embeddedmovie.asy and externalmovie.asy in the animations subdirectory of the examples directory. For a high-quality movie generated by Asymptote, use the animate module along with the animate.sty package to embed a portable PDF animation (see Section 7.15 [animate], page 83).

7.17 slide

This package provides a simple yet high-quality facility for making presentation slides, including portable embedded PDF animations (see the file slidemovies.asy). A simple example is provided in the file slidedemo.asy.

7.18 MetaPost

This package provides some useful routines to help MetaPost users migrate old MetaPost code to Asymptote. Further contributions here are welcome.

7.19 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"));
```

Support for Chinese, Japanese, and Korean fonts is provided by the CJK package:

http://www.tug.org/tex-archive/languages/chinese/CJK/

The following commands enable the CJK song family (within a label, you can also temporarily switch to another family, say kai, by prepending "\CJKfamily{kai}" to the label string):

```
texpreamble("\usepackage{CJK}
\AtBeginDocument{\begin{CJK*}{GBK}{song}}
\AtEndDocument{\clearpage\end{CJK*}}");
```

7.20 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.

7.21 babel

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

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

7.22 labelpath

This module uses the PSTricks pstextpath macro to fit labels along a path (properly kerned, as illustrated in the example file curvedlabel.asy), using the command

```
void labelpath(picture pic=currentpicture, Label L, path g,
               string justify=Centered, pen p=currentpen);
```

Here justify is one of LeftJustified, Centered, or RightJustified. The x component of a shift transform applied to the Label is interpreted as a shift along the curve, whereas

the y component is interpreted as a shift away from the curve. All other Label transforms are ignored. This package requires the `latex` tex engine and inherits the limitations of the `PSTricks \pstextpath` macro.

7.23 annotate

This module supports PDF annotations for viewing with `Adobe Reader`, via the function

```
void annotate(picture pic=currentpicture, string title, string text,
              pair position);
```

Annotations are illustrated in the example file `annotation.asy`. Currently, annotations are only implemented for the `latex` (default) and `tex` TEX engines.

7.24 CAD

This package, contributed by Mark Henning, provides basic pen definitions and measurement functions for simple 2D CAD drawings according to DIN 15. It is documented separately, in the file `CAD.pdf`.

7.25 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 99) 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);
  - `Hermite` (standard cubic spline interpolation using boundary condition `notaknot`, `natural`, `periodic`, `clamped(real slopea, real slopeb)`), or `monotonic`. The abbreviation `Hermite` is equivalent to `Hermite(notaknot)` for nonperiodic data and `Hermite(periodic)` for periodic data).

- ```
  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(picture pic=currentpicture, 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. These parametric versions also accept an arbitrary sampling function:

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

- ```
  guide polargraph(picture pic=currentpicture, 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 18), 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 optional `arrow` argument takes the same values as in the `draw` command (see [arrows], page 15). If `put`=`Below` and the `extend` flag for `axis` is `false`, the axis is drawn before any existing objects in the current picture. 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.

Custom axis types can be created by following the examples in `graph.asy`. One can easily override the default values for the standard axis types:

```
import graph;

YZero=new axis(bool extend=true) {
  return new void(picture pic, axisT axis) {
    real y=pic.scale.x.scale.logarithmic ? 1 : 0;
    axis.value=I*pic.scale.y.T(y);
    axis.position=1;
    axis.side=right;
    axis.align=2.5E;
    axis.value2=Infinity;
    axis.extend=extend;
  };
};
YZero=YZero();
```

The default tick option is `NoTicks`. The options `LeftTicks`, `RightTicks`, or `Ticks` can be used to draw ticks on the left, right, or both sides 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, tickmodifier modify=None,
                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 27; 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 major 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 99), divide the values evenly into this many intervals, separated by major ticks;

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

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

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

`bool begin`
: include the first major tick;

`bool end`
: include the last major tick;

`tickmodifier modify;`
: an optional function that takes and returns a `tickvalue` structure having real[] members `major` and `minor` consisting of the tick values (to allow modification of the automatically generated tick values);

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

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

`bool extend;`
: extend the ticks between two axes (useful for drawing a grid on the graph);

`pen pTick`
: an optional pen used to draw the major ticks;

`pen ptick`
: an optional pen used to draw the minor ticks.

For convenience, the predefined tickmodifier `OmitTick(... real[] x)` tickmodifier can be used to remove specific auto-generated ticks and their labels. The `OmitFormat(string s=defaultformat ... real[] x)` ticklabel can be used to remove specific tick labels but not the corresponding ticks. The tickmodifier `NoZero`

is an abbreviation for `OmitTick(0)` and the ticklabel `NoZeroFormat` is an abbrevation for `OmitFormat(0)`.

It is also possible to specify custom tick locations with `LeftTicks`, `RightTicks`, and `Ticks` by passing explicit real arrays `Ticks` and (optionally) `ticks` containing the locations of the major and minor 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 18). An infinite value of `ymin` or `ymax` specifies that the corresponding axis limit will be automatically determined from the picture limits. The optional `arrow` argument takes the same values as in the `draw` command (see [arrows], page 15). If `put`=`Below` and the `extend` flag for `axis` is `false`, the axis is drawn before any existing objects in the current picture. 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.

- 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="", path 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 locator `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):

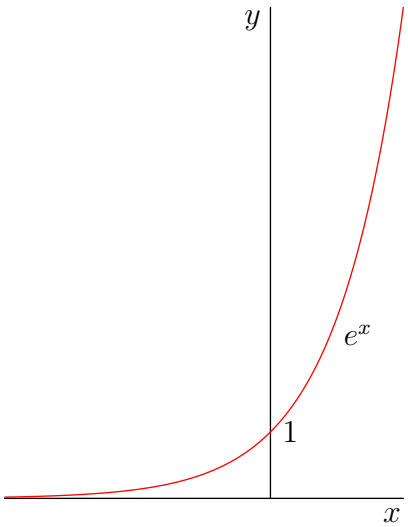
```
void xtick(picture pic=currentpicture, Label L="", explicit pair z,
           pair dir=N, string format="",
           real size=Ticksize, pen p=currentpen);
void xtick(picture pic=currentpicture, Label L="", real x,
           pair dir=N, string format="",
           real size=Ticksize, pen p=currentpen);
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="", explicit pair z,
            align align=S, string format="", pen p=nullpen);
void labelx(picture pic=currentpicture, Label L="", real x,
            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:

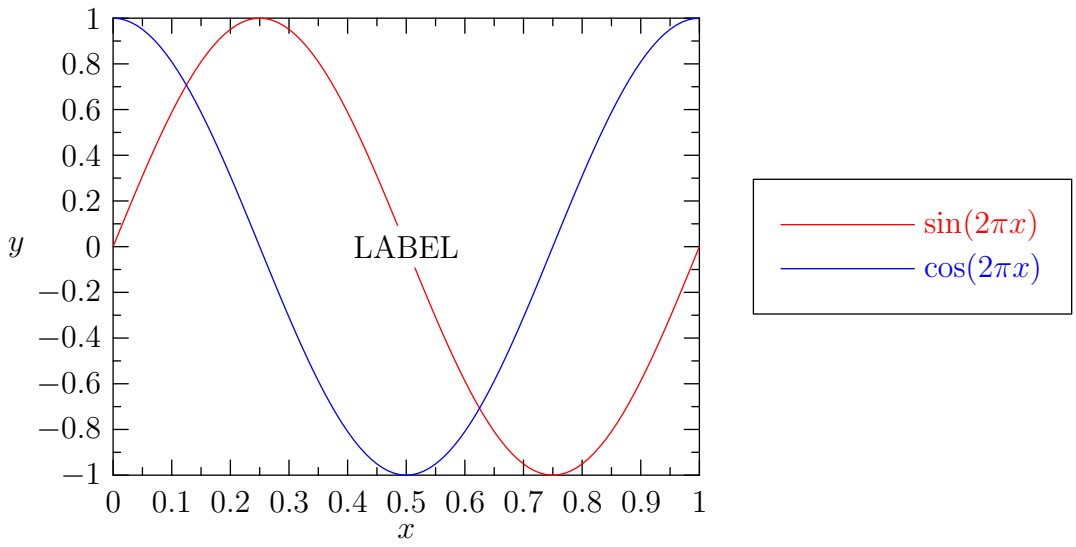
1. This example draws a textbook-style graph of $y = \exp(x)$, with the y axis starting at $y = 0$:

```
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 10 [GUI], page 143). 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(legend(),point(E),20E,UnFill);
```



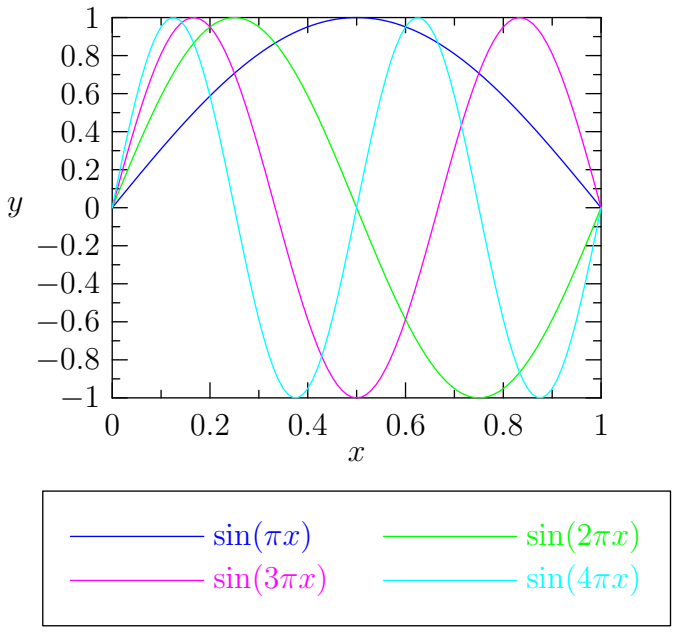
To specify a fixed size for the graph proper, use `attach`:

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

A legend can have multiple entries per line:

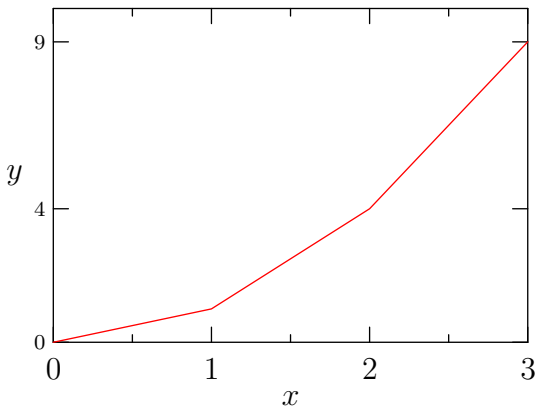
```
import graph;
size(8cm,6cm,IgnoreAspect);

typedef real realfcn(real);
realfcn F(real p) {
  return new real(real x) {return sin(p*x);};
};

for(int i=1; i < 5; ++i)
  draw(graph(F(i*pi),0,1),Pen(i),
       "$\sin("+(i == 1 ? "" : (string) i)+"\pi x)$");
xaxis("$x$",BottomTop,LeftTicks);
yaxis("$y$",LeftRight,RightTicks);

attach(legend(2),(point(S).x,truepoint(S).y),10S,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);
```

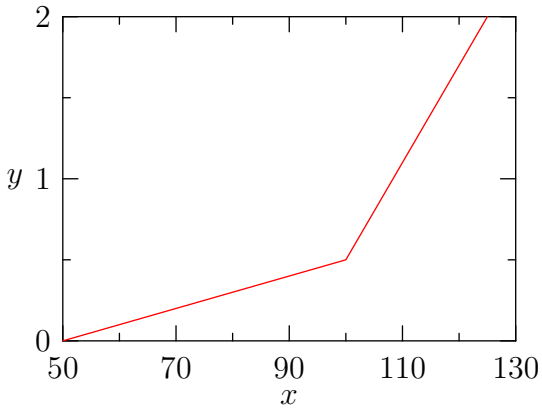
```
real[] x={0,1,2,3};
real[] y=x^2;

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

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



4. This example shows how to graph columns of data read from a file.

```
import graph;

size(200,150,IgnoreAspect);

file in=line(input("filegraph.dat"));
real[][] a=dimension(in,0,0);
a=transpose(a);

real[] x=a[0];
real[] y=a[1];

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

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

```
yaxis("$y$",LeftRight,RightTicks);
```



5. 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 44), 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 cyclic cross `cross(int n, bool round=true, real r=0)` (where `r` is an optional "inner" radius) are used to build a custom marker frame. Here `markuniform(bool centered=false, int n, bool rotated=false)` adds this frame at `n` uniformly spaced points along the arclength of the path, optionally rotated by the angle of the local tangent to the path (if centered is true, the frames will be centered within `n` evenly spaced arclength intervals). These markers are predefined:

```
marker[] Mark={
  marker(scale(circlescale)*unitcircle),
  marker(polygon(3)),marker(polygon(4)),
  marker(polygon(5)),marker(invert*polygon(3)),
  marker(cross(4)),marker(cross(6))
};

marker[] MarkFill={
  marker(scale(circlescale)*unitcircle,Fill),marker(polygon(3),Fill),
  marker(polygon(4),Fill),marker(polygon(5),Fill),
  marker(invert*polygon(3),Fill)
};
```

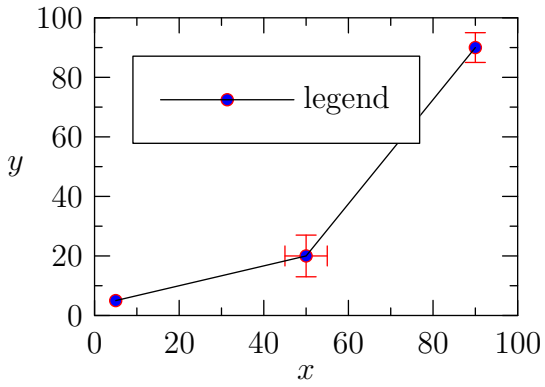
The example also illustrates the `errorbar` routines:

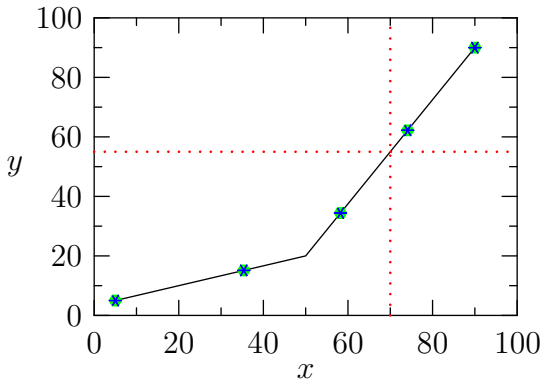
```
void errorbars(picture pic=currentpicture, pair[] z, pair[] dp,
```

```
                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.

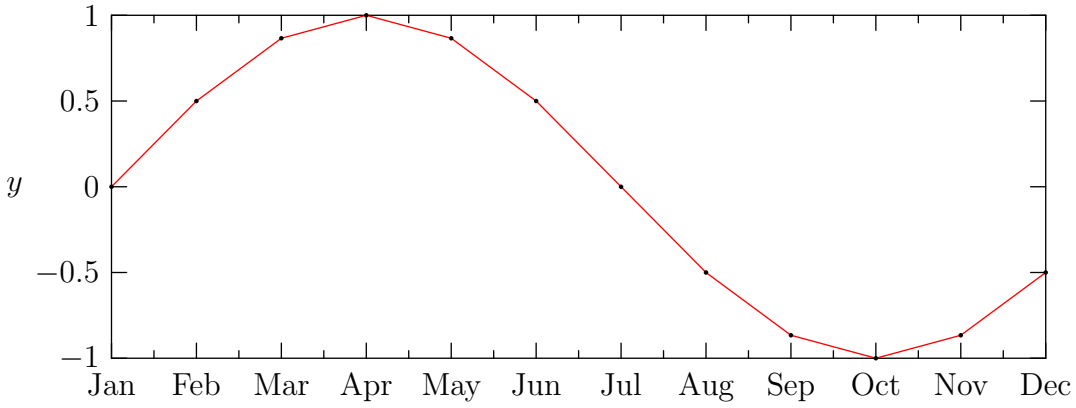
```
import graph;

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,red,FillDraw(blue),Below));

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

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

frame mark;
filldraw(mark,scale(0.8mm)*polygon(6),green,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(),(0,0),W);

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





6. This example shows how to label an axis with arbitrary strings.

```
import graph;

size(400,150,IgnoreAspect);

real[] x=sequence(12);
real[] y=sin(2pi*x/12);

scale(false);

string[] month={"Jan","Feb","Mar","Apr","May","Jun",
                "Jul","Aug","Sep","Oct","Nov","Dec"};

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

xaxis(BottomTop,LeftTicks(new string(real x) {
      return month[round(x % 12)];}));
yaxis("$y$",LeftRight,RightTicks(4));
```

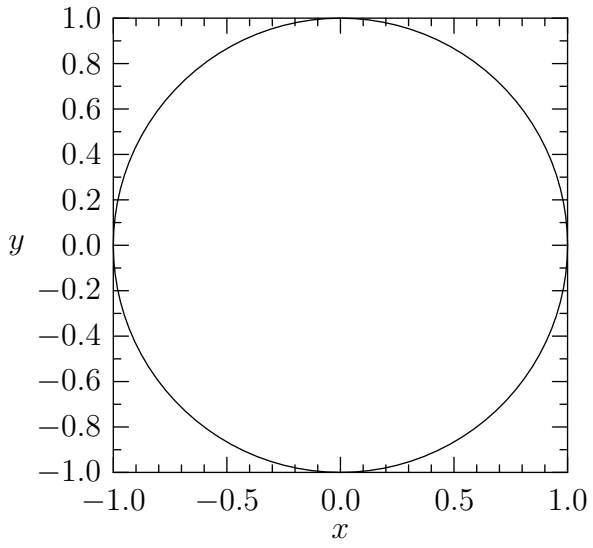


7. This example draws a graph of a parametrized curve.
   The calls to

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

and the analogous function `ylimits` can be uncommented to set 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 if `crop=Crop`. The function `crop(picture pic)` is equivalent to calling both `xlimits(Crop)` and `ylimits(Crop)`.

```
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,Crop);
//ylimits(-1,0,Crop);

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 one of these `scale` routines:

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

void scale(picture pic=currentpicture, bool xautoscale=true,
```

```
            bool yautoscale=xautoscale, bool zautoscale=yautoscale);
```

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`.

Two frequently used scaling routines `Linear` and `Log` are predefined in `graph`.

All picture coordinates (including those in paths and those given to the `label` and `limits` functions) are always treated as linear (post-scaled) coordinates. Use

```
pair Scale(picture pic=currentpicture, pair z);
```

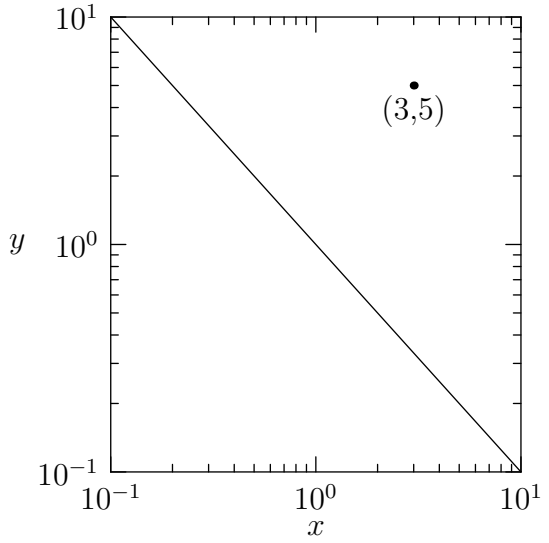
to convert a graph coordinate into a scaled picture coordinate.

```
The @math{x} and @math{y} components can be individually scaled using
the analogous routines
@verbatim
real ScaleX(picture pic=currentpicture, real x);
real ScaleY(picture pic=currentpicture, real y);
```

The predefined 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 optional final arguments a multiplicative scaling factor and intercept (e.g. for a depth axis, `Linear(-1)` requests axis reversal).

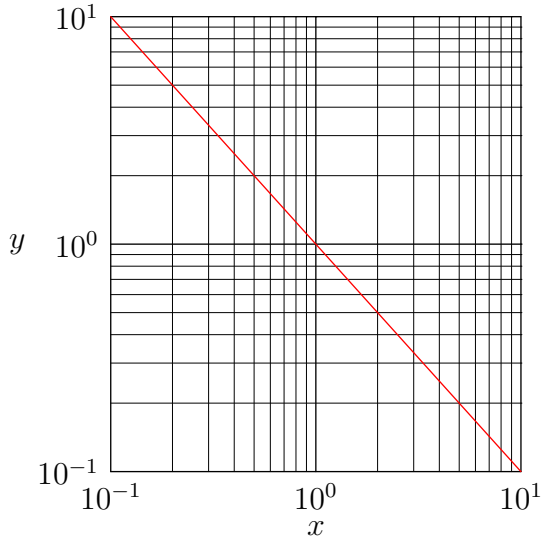
For example, to draw a log/log graph of a function, use `scale(Log,Log)`:

```
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);

dot(Label("(3,5)",align=S),Scale((3,5)));

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



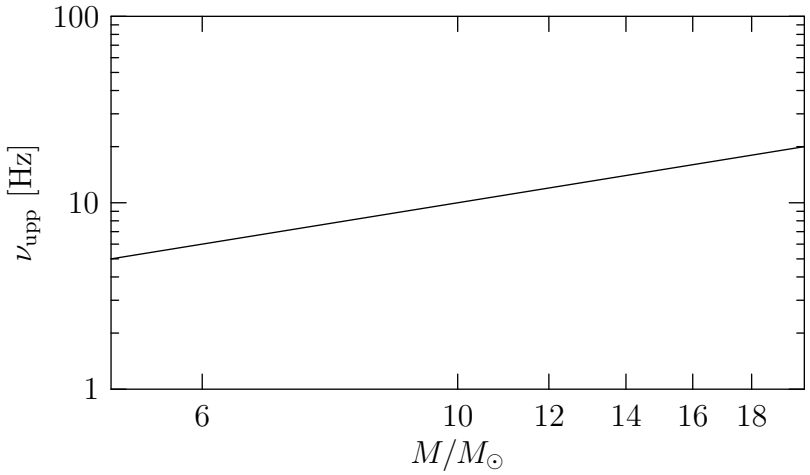
By extending the ticks, one can easily produce a logarithmic grid:

```
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));
```



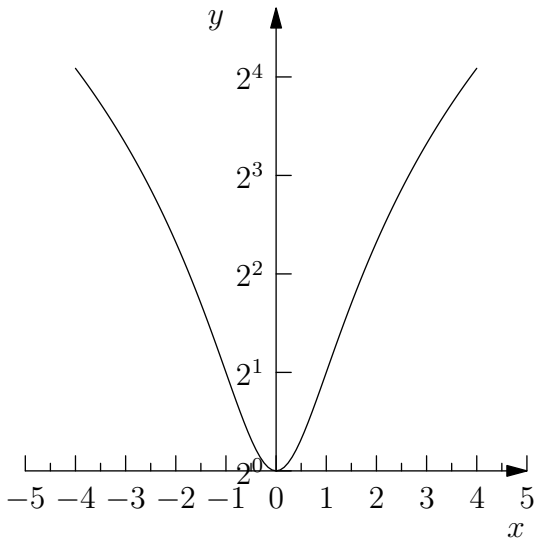
One can also specify custom tick locations and formats for logarithmic axes:

```
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(DefaultFormat,
                                        new real[] {6,10,12,14,16,18}));
yaxis("$\nu_{\rm upp}$ [Hz]",LeftRight,RightTicks(DefaultFormat));
```



It is easy to draw logarithmic graphs with respect to other bases:

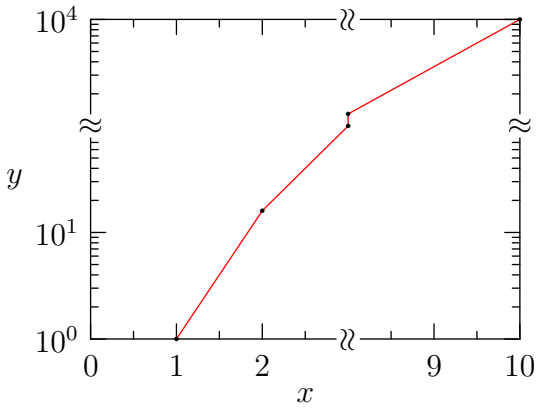
```
import graph;
size(200,IgnoreAspect);

// Base-2 logarithmic scale on y-axis:
scaleT yscale;

real log2(real x) {static real log2=log(2); return log(x)/log2;}
real pow2(real x) {return 2^x;}

yscale.init(log2,pow2,logarithmic=true);
scale(Linear,yscale);

real f(real x) {return 1+x^2;}

draw(graph(f,-4,4));

xaxis("$x$",xmin=-5,xmax=5,LeftTicks,EndArrow);
yaxis("$y$",ymin=1,ymax=f(5),RightTicks,EndArrow);
```



Here is an example of "broken" linear x and logarithmic y axes that omit the segments [3,8] and [100,1000], respectively. In the case of a logarithmic axis, the break endpoints are automatically rounded to the nearest integral power of the base.

```
import graph;

size(200,150,IgnoreAspect);

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

// Break the y axis at 100; restart at 1000:
```

```
real c=100, d=1000;

scale(Broken(a,b),BrokenLog(c,d));

real[] x={1,2,4,6,10};
real[] y=x^4;

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

xaxis("$x$",BottomTop,LeftTicks(Break(a,b)));
yaxis("$y$",LeftRight,RightTicks(Break(c,d)));

label(rotate(90)*Break,(a,point(S).y));
label(rotate(90)*Break,(a,point(N).y));
label(Break,(point(W).x,ScaleY(c)));
label(Break,(point(E).x,ScaleY(c)));
```



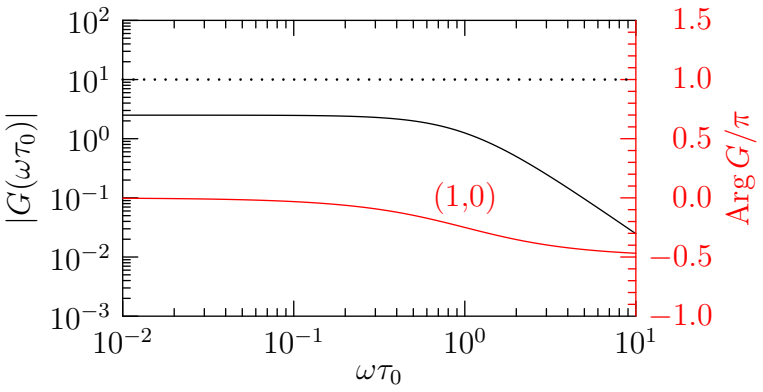
8. `Asymptote` can draw secondary axes with the routines

   ```
   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(0.001,100);
   ```

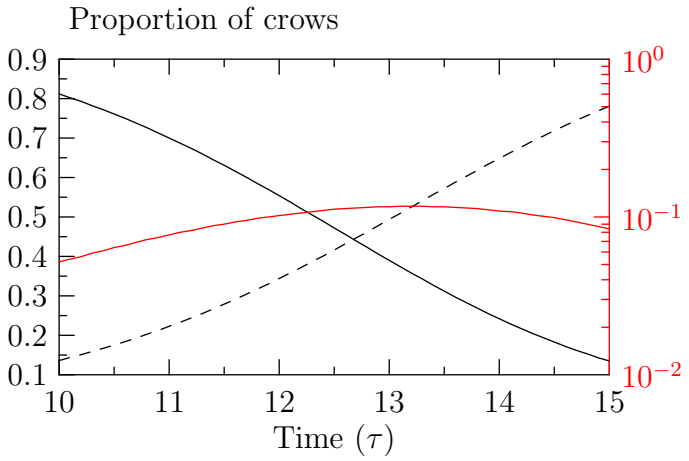
```
xaxis("$\omega\tau_0$",BottomTop,LeftTicks);
yaxis("$|G(\omega\tau_0)|$",Left,RightTicks);

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:

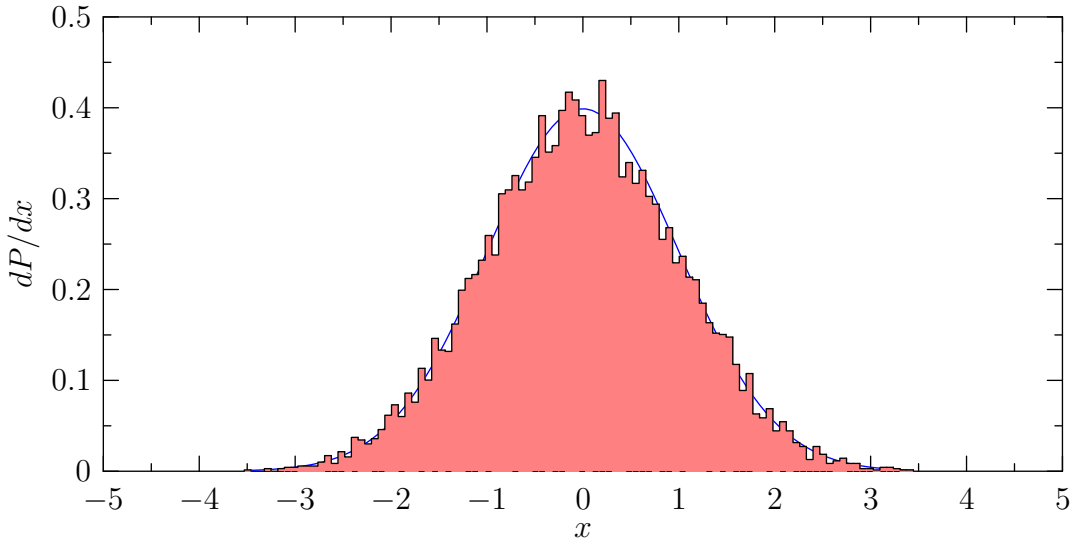
```
import graph;

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);
```



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

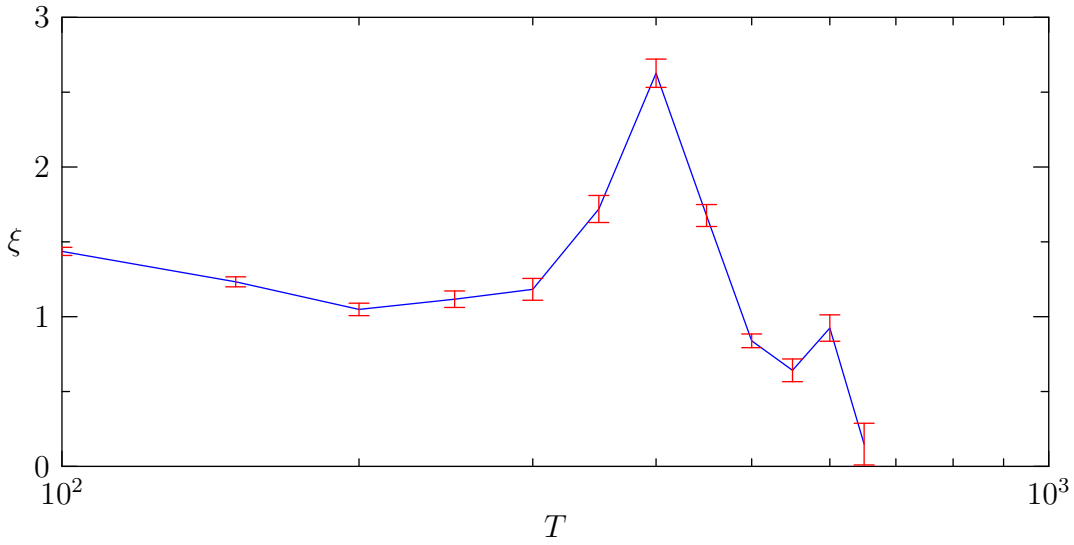
```
import graph;
import stats;

size(400,200,IgnoreAspect);

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

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

histogram(a,min(a),max(a),n=100,normalize=true,low=0,lightred,black,bars=false);

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



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

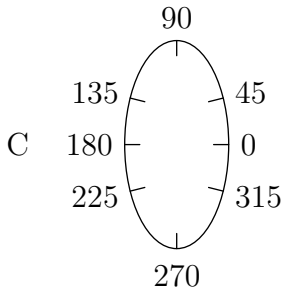
```
size(400,200,IgnoreAspect);

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);

crop();

ylimits(0);

xaxis("$T$",BottomTop,LeftTicks);
yaxis("$\xi$",LeftRight,RightTicks);
```



11. 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)[0];}));
```



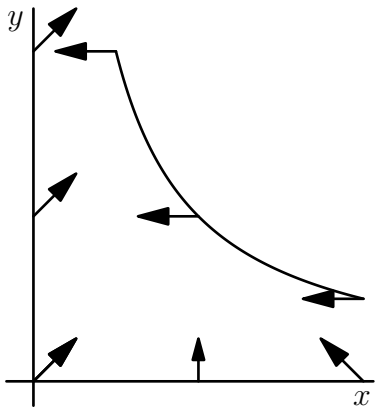
12. To draw a vector field along a path, first define a routine that returns a path as a function of a relative position parameter from [0,1] and use

```
typedef path vector(real);

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

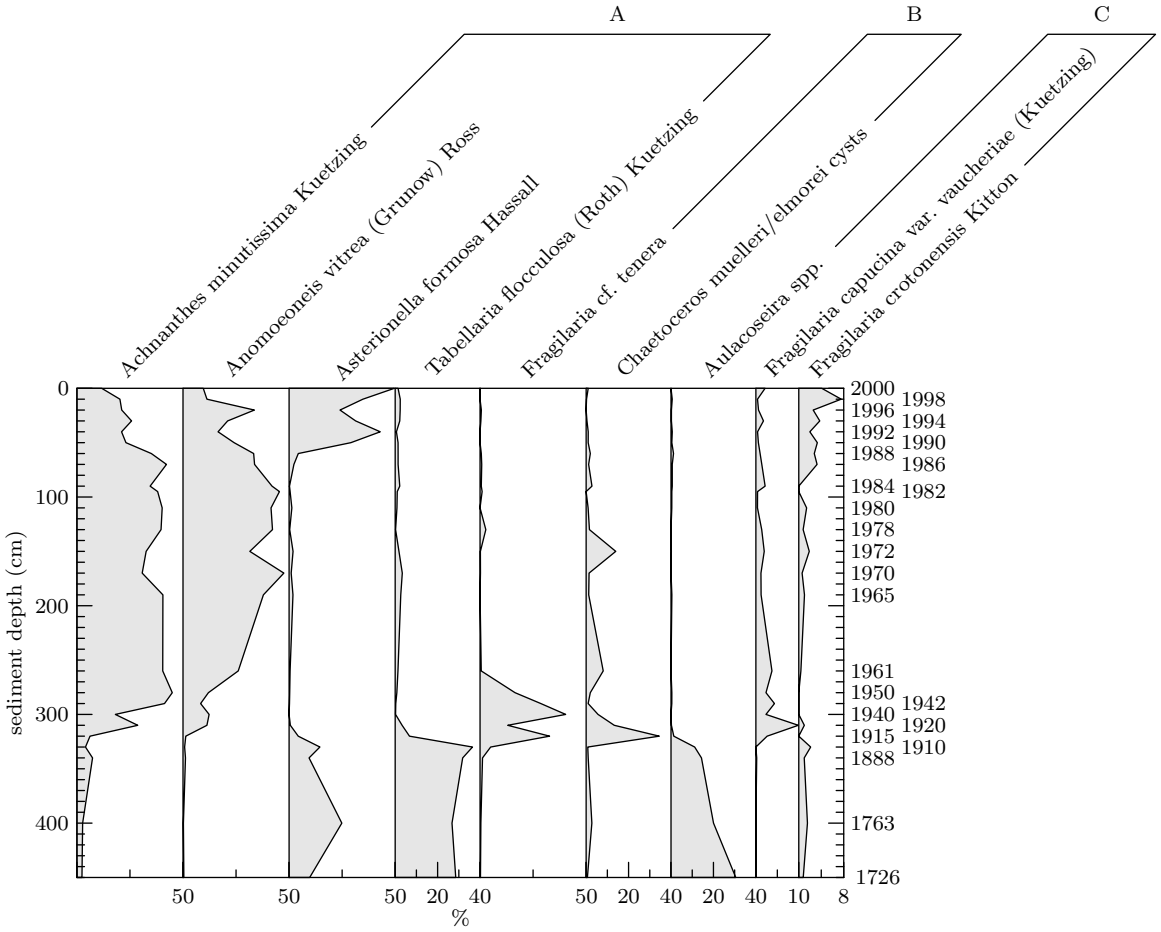
Here is a simple example of a flow field:

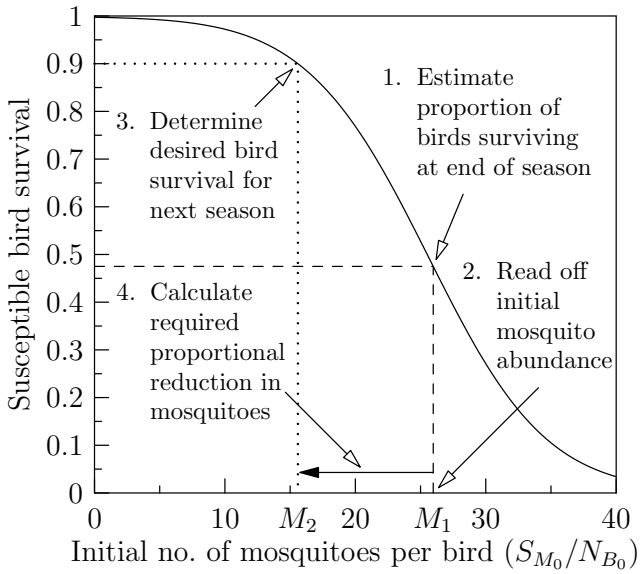
```
import graph;
defaultpen(1.0);

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)--(point(E).x,0),n,vector(NE,NW),arrowsize);
vectorfield((0,0)--(0,point(N).y),n,vector(NE,NE),arrowsize);
```



13. 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`.


A
B
C
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
20
40
50
20
40
20
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})

7.26 palette

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 using palette `palette` can be generated from a function `f`(x,y) and added to a picture `pic`:

```
bounds image(picture pic=currentpicture, real f(real,real),
             range range=Full, pair initial, pair final,
             int nx=ngraph, int ny=nx, pen[] palette);
```

The function `f` will be sampled at `nx` and `ny` evenly spaced points over a rectangle defined by the points `initial` and `final`, respecting the current graphical scaling of `pic`. The color space is scaled according to the z axis scaling (see [automatic scaling], page 99). A bounds structure for the function values is returned:

```
struct bounds {
  real min;
  real max;
  // Possible tick intervals:
  int[] divisor;
}
```

This information can be used for generating an optional palette bar. The palette color space corresponds to a range of values specified by the argument `range`, which can be `Full`, `Automatic`, or an explicit range `Range(real min, real max)`. Here `Full` specifies a range varying from the minimum to maximum values of the function over the sampling interval, while `Automatic` selects "nice" limits. The example `imagecontour.asy` illustrates how level sets (contour lines) can be drawn on a color density plot (see Section 7.35 [contour], page 130).

A color density plot can also be generated from an explicit real[][] array `data`:

```
bounds image(picture pic=currentpicture, real[][] f, range range=Full,
             pair initial, pair final, pen[] palette,
             bool transpose=(initial.x < final.x && initial.y < final.y));
```

If the initial point is to the left and below the final point, by default the array indices are interpreted according to the Cartesian convention (first index: x, second index: y) rather than the usual matrix convention (first index: $-y$, second index: x).

To construct an image from an array of irregularly spaced points and an array of values `f` at these points, use one of the routines

```
bounds image(picture pic=currentpicture, pair[] z, real[] f,
             range range=Full, pen[] palette)
bounds image(picture pic=currentpicture, real[] x, real[] y, real[] f,
             range range=Full, pen[] palette)
```

An optionally labelled palette bar may be generated with the routine

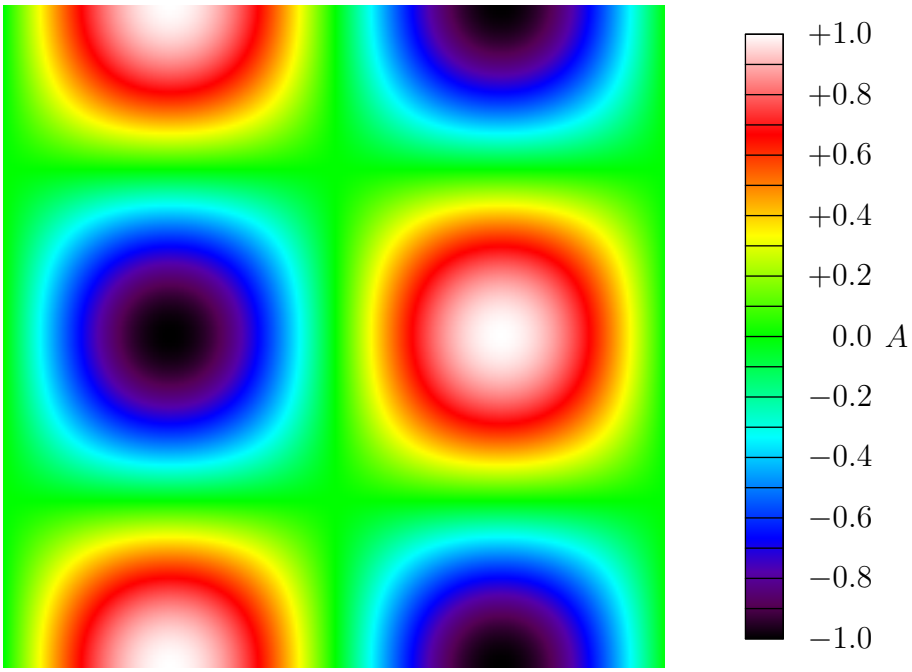
```
void palette(picture pic=currentpicture, Label L="", bounds bounds,
             pair initial, pair final, axis axis=Right, pen[] palette,
             pen p=currentpen, paletteticks ticks=PaletteTicks);
```

The color space of `palette` is taken to be over bounds `bounds` with scaling given by the z scaling of `pic`. The palette orientation is specified by `axis`, which may be one of `Right`, `Left`, `Top`, or `Bottom`. The bar is drawn over the rectangle from `initial` to `final`. The argument `paletteticks` is a special tick type (see [ticks], page 88) that takes the following arguments:

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

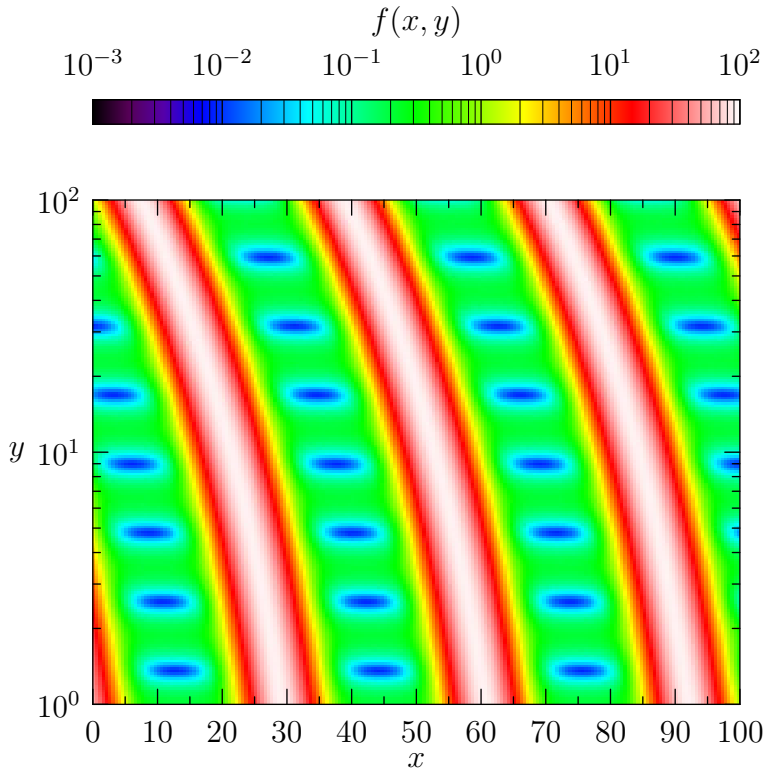
The image and palette bar can be fit to a frame and added and optionally aligned to a picture at the desired location:

```
size(12cm,12cm);

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 bar;

bounds range=image(v,(0,0),(1,1),Palette);
palette(bar,"$A$",range,(0,0),(0.5cm,8cm),Right,Palette,
        PaletteTicks("$%+#.1f$"));
add(bar.fit(),point(E),30E);
```



Here is an example that uses logarithmic scaling of the function values:

```
import graph;
import palette;

size(10cm,10cm,IgnoreAspect);

real f(real x, real y) {
  return 0.9*pow10(2*sin(x/5+2*y^0.25)) + 0.1*(1+cos(10*log(y)));
}

scale(Linear,Log,Log);

pen[] Palette=BWRainbow();

bounds range=image(f,Automatic,(0,1),(100,100),nx=200,Palette);

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

palette("$f(x,y)$",range,(0,200),(100,250),Top,Palette,
        PaletteTicks(ptick=linewidth(0.5*linewidth())));
```

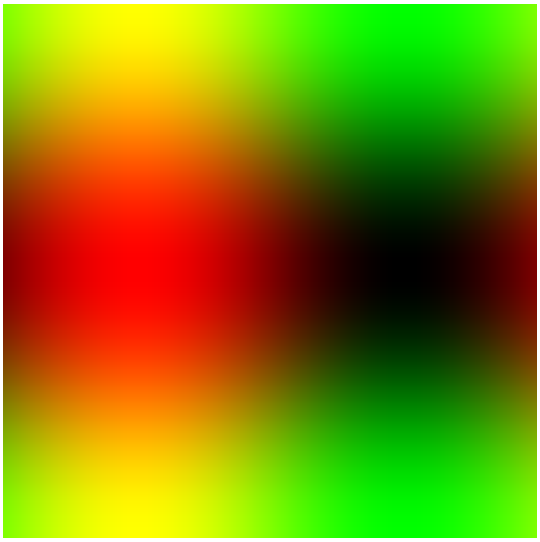


One can also draw an image directly from a two-dimensional pen array:

```
void image(picture pic=currentpicture, pen[][] data,
           pair initial, pair final,
           bool transpose=(initial.x < final.x && initial.y < final.y));
```

as illustrated in the following example:

```
size(200);

import palette;

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

for(int i=0; i < n; ++i)
  for(int j=0; j < n; ++j)
    v[i][j]=rgb(0.5*(1+sin(i*ninv)),0.5*(1+cos(j*ninv)),0);

picture bar;

image(v,(0,0),(1,1));
```

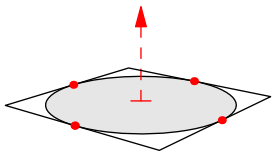


7.27 three

This module fully extends the notion of guides and paths in Asymptote to three dimensions, introducing the new types guide3 and path3. Guides in three dimensions are specified with the same syntax as in two dimensions but with triples (x,y,z) in place of pairs (x,y) for the nodes and direction specifiers. 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:

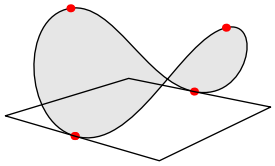
```
import three;
size(100,0);
guide3 g=(1,0,0)..(0,1,0)..(-1,0,0)..(0,-1,0)..cycle;
filldraw(g,lightgrey);
draw(O--Z,red+dashed,BeginBar,Arrow);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle));
dot(g,red);
```



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)..cycle;
filldraw(g,lightgrey);
dot(g,red);
```

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



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 [GaussianSurface], page 125 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

```
path3[] box(triple v1, triple v2);
```

For example, a unit cube is predefined as

```
path3[] unitcube=box((0,0,0),(1,1,1));
```

These projections to two dimensions are predefined:

`oblique`

`oblique(real angle)`
: The point `(x,y,z)` is projected to `(x-0.5z,y-0.5z)`. If an optional real argument is given, the negative z axis is drawn at this angle in degrees. The projection `obliqueZ` is a synonym for `oblique`.

`obliqueX`

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

`obliqueY`

`obliqueY(real angle)`
: The point `(x,y,z)` is projected to `(x+0.5y,z+0.5y)`. If an optional real argument is given, the positive y axis is drawn at this angle in degrees.

`orthographic(triple camera, triple up=Z)`
: This projects from three to two dimensions using the view as seen at a point infinitely far away in the direction `unit(camera)`, orienting the camera so that, if possible, the vector `up` points upwards. Parallel lines are projected to parallel lines.

`orthographic(real x, real y, real z, triple up=Z)`
: This is equivalent to `orthographic((x,y,z),up)`.

`perspective(triple camera, triple up=Z, triple target=0)`
: This projects from three to two dimensions, taking account of perspective, as seen from the location `camera` looking at `target`, orienting the camera so that, if possible, the vector `up` points upwards.

`perspective(real x, real y, real z, triple up=Z, triple target=0)`
: This is equivalent to `perspective((x,y,z),up,target)`.

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

A triple or path3 can be implicitly cast to a pair or path, respectively, using `currentprojection`. To use another projection, call `project(triple, projection P=currentprojection)` or `project(path, projection P=currentprojection)`.

It is occasionally useful to be able to invert a projection, sending a pair `z` onto the plane perpendicular to `normal` and passing through `point`:

```
triple invert(pair z, triple normal, triple point,
              projection P=currentprojection);
```

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`.

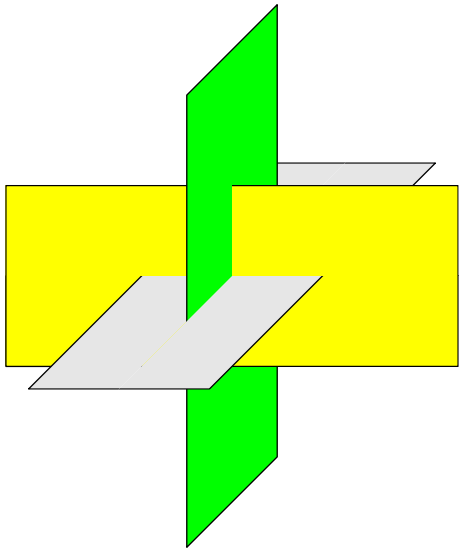
Three-dimensional versions of the path functions `length`, `size`, `point`, `dir`, `precontrol`, `postcontrol`, `arclength`, `arctime`, `reverse`, `subpath`, `intersect`, `intersectionpoint`, `min`, `max`, `cyclic`, and `straight` are also defined in the module `three`.

Planar hidden surface removal is implemented with a binary space partition and picture clipping. A planar path is first converted to a structure `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);
```

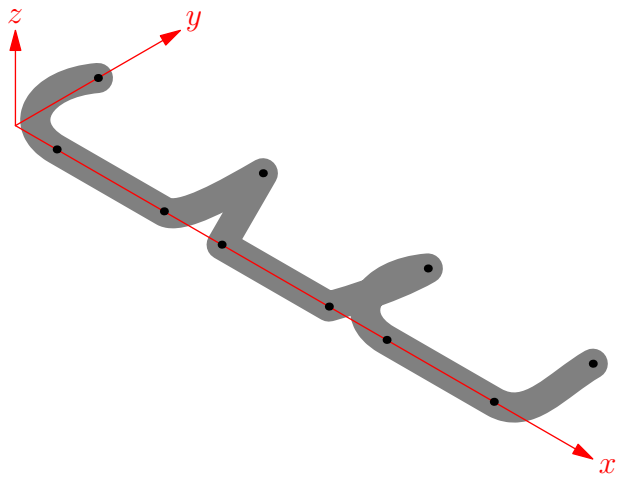
Here is an example showing three orthogonal intersecting planes:

```
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 3D 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`.

7.28 light

This module provides a simple implementation of three-dimensional lighting effects. An illustration is provided in the solid geometry package `solids.asy`.

7.29 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 from triple `min` to triple `(max,min.y,min.z)` with ticks in the direction `dir`, use the routine

```
void xaxis(picture pic=currentpicture, Label L="", triple min, real 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);
```

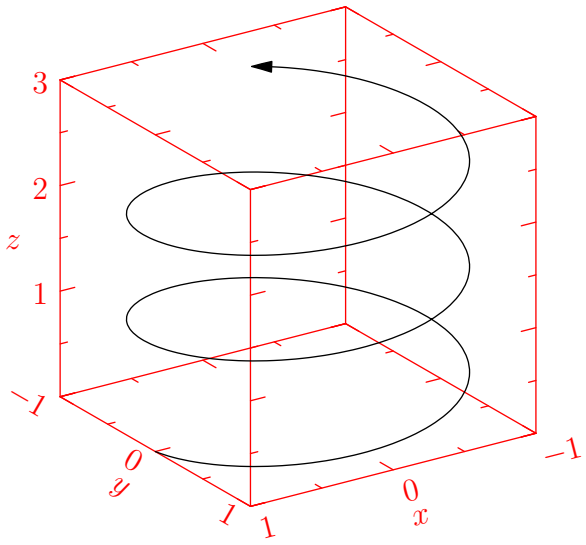
If `all=true`, also draw opposing edges of the three-dimensional bounding box. Analogous routines `yaxis` and `zaxis` can be used to draw y and z axes in three dimensions. There are also some convenient routines for drawing all three axis:

```
void axes(Label xlabel="$x$", Label ylabel="$y$", Label zlabel="$z$",
          triple min, triple max, pen p=currentpen, arrowbar arrow=None,
          bool put=Below, projection P=currentprojection);

void axes(Label xlabel="$x$", Label ylabel="$y$", Label zlabel="$z$",
          bbox3 b, pen p=currentpen, arrowbar arrow=None,
          bool put=Below, projection P=currentprojection);
```

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

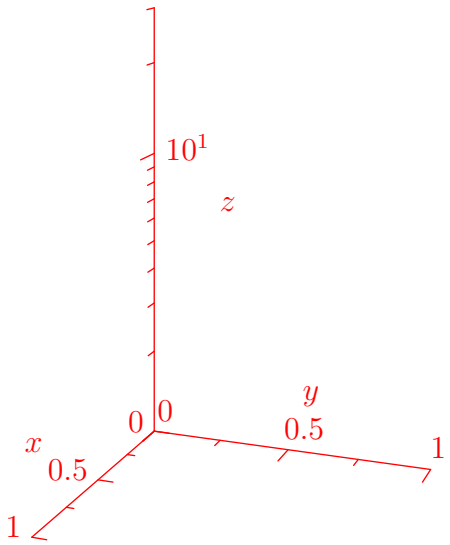
```
import graph3;

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);
```



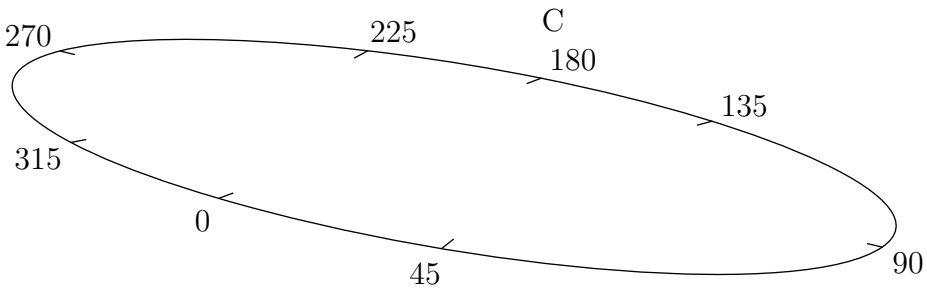
The next example illustrates three-dimensional x, y, and z axes, with autoscaling of the upper z limit disabled:

```
import 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)[0];
       },perpendicular(G,Z)));
```



Surface plots of matrices and functions over the region `box(a,b)` in the XY plane are also implemented:

```
picture surface(real[][] f, pair a, pair b, bool outward=false,
```

```
                pen surfacepen=lightgray, pen meshpen=nullpen,
                light light=currentlight, projection P=currentprojection);
picture surface(triple[][] f, bool outward=false,
                pen surfacepen=lightgray, pen meshpen=nullpen,
                light light=currentlight, projection P=currentprojection);
picture surface(real f(pair z), pair a, pair b, int nx=nmesh,
                int ny=nx, bool outward=false,
                pen surfacepen=lightgray, pen meshpen=nullpen,
                light light=currentlight, projection P=currentprojection);
picture surface(real f(pair z), int nsub, pair a, pair b, int nx=nmesh,
                int ny=nx, bool outward=false,
                pen surfacepen=lightgray, pen meshpen=nullpen,
                light light=currentlight, projection P=currentprojection);
```

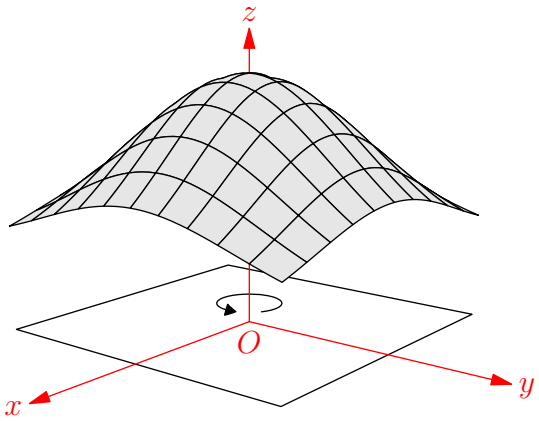
A parametric surface for a function $f(u,v)$ over the parameter space `box(a,b)` may be drawn with

```
picture surface(triple f(pair z), pair a, pair b, int nu=nmesh,
                int nv=nu, bool outward=false,
                pen surfacepen=lightgray, pen meshpen=nullpen,
                light light=currentlight, projection P=currentprojection);
picture surface(triple f(pair z), int nsub, pair a, pair b, int nu=nmesh,
                int nv=nu, bool outward=false,
                pen surfacepen=lightgray, pen meshpen=nullpen,
                light light=currentlight, projection P=currentprojection);
```

as illustrated in the example `parametricsurface.asy`. The optional argument `nsub` allows one to subsample the function `nsub` times along the cell edges. However, smoother Gouraud shading is used for lighting effects when this parameter is omitted. Lighting can be disabled by setting `light=O`, as in this example of a Gaussian surface subsampled 4 times:

```
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)--cycle);

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

picture surface=surface(f,nsub=4,(-1,-1),(1,1),nx=10,light=O);

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);
```

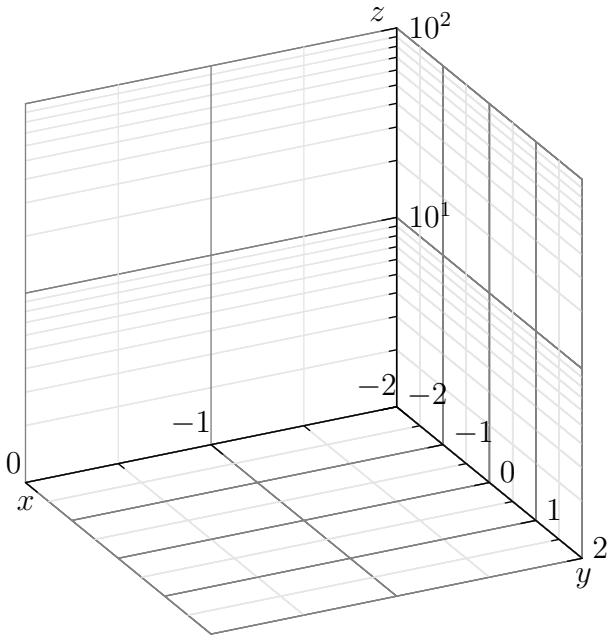


Surface lighting is illustrated in the example files `parametricsurface.asy` and `sinc.asy`. A mesh can be drawn without solid fills by specifying `nullpen` for the surfacepen. A mesh is always drawn in the absence of lighting, unless `meshpen=invisible`. For closed orientable surfaces parametrized such that $df/du \times df/dv$ points in the direction of the outward normal, low-mesh-resolution lighting artifacts near the projected edges can be avoided by specifying `outward=true`, as this forces only the outer surface to be drawn.

7.30 grid3

This module, contributed by Philippe Ivaldi, can be used for drawing 3D grids. Here is an example (further examples can be found in `grid3.asy` and at http://piprim.tuxfamily.org/asymptote/grid3/):

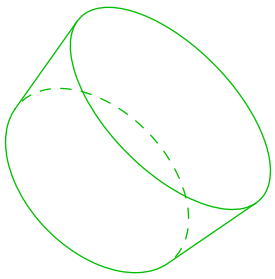
```
import grid3;

size(8cm,0);
currentprojection=orthographic(0.5,1,0.5);

defaultpen(overwrite(SuppressQuiet));
bbox3 b=limits((-2,-2,0),(0,2,2));
aspect(b,1,1,1);

scale(Linear, Linear, Log);

grid3(b,XYZgrid);
xaxis(Label("$x$",position=EndPoint,align=S),b,RightTicks());
yaxis(Label("$y$",position=EndPoint,align=S),b,RightTicks());
zaxis(Label("$z$",position=EndPoint,align=(0,0.5)+W),b,RightTicks());
```



7.31 solids

This solid geometry package defines a structure `revolution` that can be used to fill and draw surfaces of revolution. The following example uses it to display the outline of a circular cylinder of radius 1 with axis `O--1.5unit(Y+Z)` with perspective projection:

```
import solids;

size(0,100);

revolution r=cylinder(O,1,1.5,Y+Z);
r.draw(heavygreen);
```



Further illustrations are provided in the example files `cylinder.asy`, `sphere.asy`, `cones.asy`, `hyperboloid.asy`, and `torus.asy`.

The structure `skeleton` contains the three-dimensional wireframe used to visualize a volume of revolution (here longitudinal refers to those segments that are parallel to the axis of revolution; transverse segments are perpendicular to the axis of revolution):

```
struct skeleton {
  // transverse skeleton
  front3[] front;
  front3[] back;
  // longitudinal skeleton
  front3[] longitudinal;
}
```

7.32 surface

This is the beginnings of a package for drawing Bezier surfaces, the natural two-dimensional generalizations of a Bezier curve. An sample Bezier surface is contained in the example file `BezierSurface.asy`.

7.33 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 package `three`, which fully extends the notion of a path to three dimensions, together with the packages `solids` and `graph3`. The original `featpost3D` package is documented at

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

7.34 flowchart

This package provides routines to assist in drawing flowcharts. The primary structure is a `block`, which represents a single block on the flowchart. The following eight functions return a position on the appropriate edge of the block, given picture transform `t`:

```
pair block.top(transform t=identity());
pair block.left(transform t=identity());
pair block.right(transform t=identity());
pair block.bottom(transform t=identity());
pair block.topleft(transform t=identity());
pair block.topright(transform t=identity());
pair block.bottomleft(transform t=identity());
pair block.bottomright(transform t=identity());
```

To obtain an arbitrary position along the boundary of the block in user coordinates, use:

```
pair block.position(real x, transform t=identity());
```

The center of the block in user coordinates is stored in `block.center` and the block size in `PostScript` coordinates is given by `block.size`.

A frame containing the block is returned by

```
frame block.draw(pen p=currentpen);
```

The following block generation routines accept a Label, string, or frame for their object argument:

rectangular block with an optional header (and padding `dx` *around header and body):*

```
block rectangle(object header, object body,
                pen headerpen=mediumgray, pen bodypen=currentpen,
                pair center=(0,0), real dx=3,
                real minheaderwidth=0, real minheaderheight=0,
                real minbodywidth=0, real minbodyheight=0);
block rectangle(object body, pen bodypen=currentpen,
                pair center=(0,0), real dx=3,
                real minwidth=0, real minheight=0);
```

diamond-shaped flowchart block:

```
block diamond(object body, pair center=(0,0), real ds=5,
              real dw=1, real height=20,
              real minwidth=0, real minheight=0);
```

circular flowchart block:

```
block circle(object body, pair center=(0,0), real dr=3,
             real mindiameter=0);
```

rectangular flowchart block with rounded corners:

```
block roundrectangle(object body, pair center=(0,0),
                     real ds=5, real dw=0,
                     real minwidth=0, real minheight=0);
```

rectangular flowchart block with beveled edges:

```
block bevel(object body, pair center=(0,0), real dh=5, real dw=5,
            real minwidth=0, real minheight=0);
```

To draw paths joining the pairs in `point` with right-angled lines, use the routine:

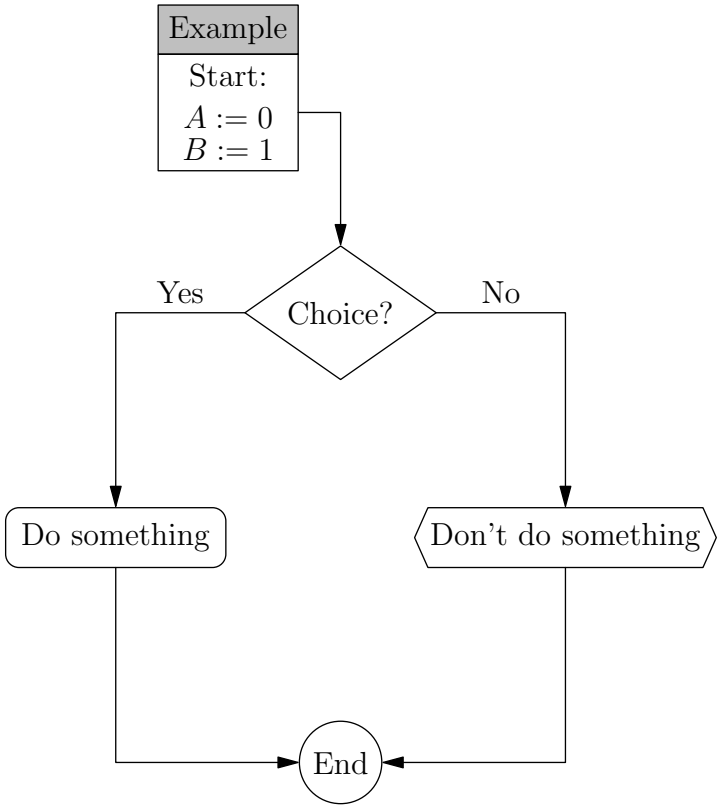
```
path path(pair point[] ... bool horizontal[]);
```

The entries in `horizontal` identify whether successive segments between the pairs specified by `point` should be drawn horizontal (`true`) or vertical (`false`).

Here is a simple flowchart example:

```
size(0,300);

import flowchart;

block block1=rectangle("Example",pack("Start:","","$A:=0$","$B:=1$"),(-0.5,3));
block block2=diamond("Choice?",(0,2));
block block3=roundrectangle("Do something",(-1,1));
block block4=bevel("Don't do something",(1,1));
block block5=circle("End",(0,0));

draw(block1);
draw(block2);
draw(block3);
draw(block4);
draw(block5);
```

```
add(new void(frame f, transform t) {
    picture pic;
    draw(pic,path(new pair[]{block1.right(t),block2.top(t)},Horizontal),
         Arrow,PenMargin);
    draw(pic,Label("Yes",0.5),path(new pair[]{block2.left(t),block3.top(t)},
                                   Horizontal),Arrow,PenMargin);
    draw(pic,Label("No",0.5,N),path(new pair[]{block2.right(t),block4.top(t)},
                                    Horizontal),Arrow,PenMargin);
    draw(pic,path(new pair[]{block3.bottom(t),block5.left(t)},Vertical),
         Arrow,PenMargin);
    draw(pic,path(new pair[]{block4.bottom(t),block5.right(t)},Vertical),
         Arrow,PenMargin);

    add(f,pic.fit());
  });
```



7.35 contour

This package draws contour lines. To construct contours for a function `f`, use

```
guide[][] contour(real f(real, real), pair a, pair b,
                  real[] c, int nx=ngraph, int ny=nx,
                  interpolate join=operator --);
```

The contour lines `c` for the function `f` are drawn on the rectangle defined by the bottom-left and top-right points `a` and `b`. The integers `nx` and `ny` define the resolution. The default resolution, `ngraph` x `ngraph` (here `ngraph` defaults to `100`), can be increased for greater accuracy. The default interpolation operator is `operator --` (linear). Spline interpolation (`operator ..`) generally produces more accurate pictures, but as usual, can overshoot in certain cases.

To construct contours for an array of data values on a uniform two-dimensional lattice, use

```
guide[][] contour(real[][] f, real[][] midpoint=new real[][],
                  pair a, pair b, real[] c,
                  interpolate join=operator --);
```

To construct contours for an array of irregularly spaced points and an array of values `f` at these points, use one of the routines

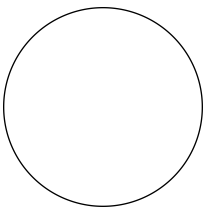
```
guide[][] contour(pair[] z, real[] f, real[] c,
                  interpolate join=operator --);
guide[][] contour(real[] x, real[] y, real[] f, real[] c,
                  interpolate join=operator --);
```

The contours themselves can be drawn with one of the routines

```
void draw(picture pic=currentpicture, Label[] L=new Label[],
          guide[][] g, pen p=currentpen)

void draw(picture pic=currentpicture, Label[] L=new Label[],
          guide[][] g, pen[] p)
```

The following simple example draws the contour at value `1` for the function $z = x^2 + y^2$, which is a unit circle:

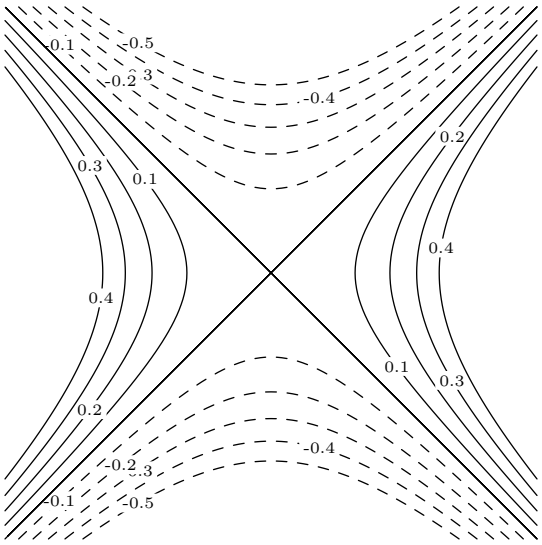
```
import contour;
size(75);

real f(real a, real b) {return a^2+b^2;}
draw(contour(f,(-1,-1),(1,1),new real[] {1}));
```



The next example draws and labels multiple contours for the function $z = x^2 - y^2$ with the resolution `100` x `100`, using a dashed pen for negative contours and a solid pen for positive (and zero) contours:

```
import contour;
import stats;
size(200);
```

```
real f(real x, real y) {return x^2-y^2;}
int n=10;
real[] c = new real[n];
for(int i=0; i < n; ++i) c[i]=(i-n/2)/n;

pen[] p=sequence(new pen(int i) {
    return (c[i] >= 0 ? solid : dashed)+fontsize(6);
  },n);

Label[] Labels=sequence(new Label(int i) {
    return Label(c[i] != 0 ? (string) c[i] : "",Relative(unitrand()),(0,0),
                 UnFill(1bp));
  },n);

draw(Labels,contour(f,(-1,-1),(1,1),c),p);
```

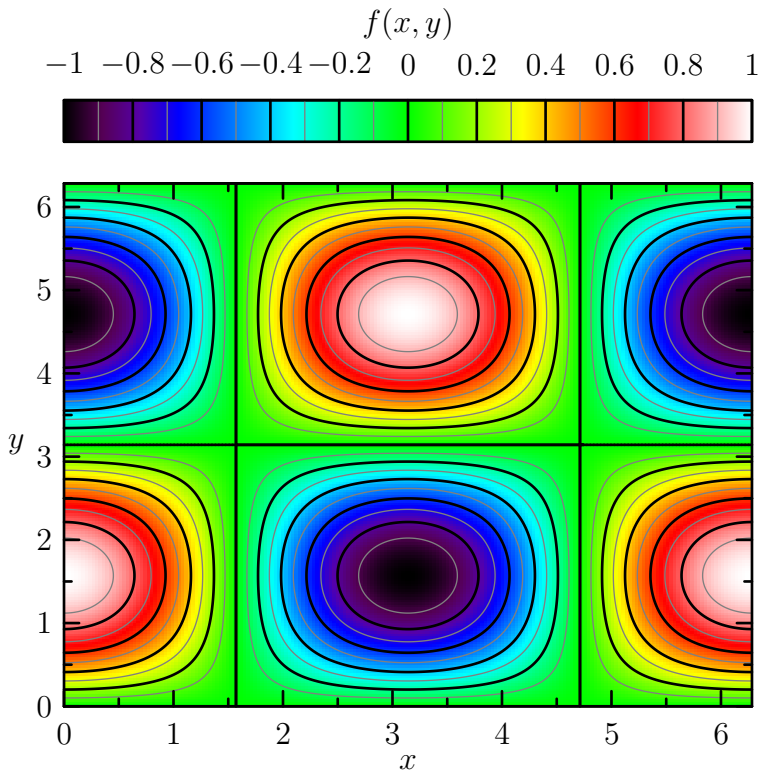


The next example illustrates how contour lines can be drawn on color density images:

```
import graph;
import palette;
import contour;

size(10cm,10cm,IgnoreAspect);

pair a=(0,0);
pair b=(2pi,2pi);

real f(real x, real y) {return cos(x)*sin(y);}

int N=200;
```

```
int Divs=10;
int divs=2;

defaultpen(1bp);
pen Tickpen=black;
pen tickpen=gray+0.5*linewidth(currentpen);
pen[] Palette=BWRainbow();

scale(false);

bounds range=image(f,Automatic,a,b,N,Palette);

// Major contours
real[] Cvals;
Cvals=sequence(Divs+1)/Divs*(range.max-range.min)+range.min;
draw(contour(f,a,b,Cvals,N,operator --),Tickpen);

// Minor contours
real[] cvals;
real[] sumarr=sequence(1,divs-1)/divs*(range.max-range.min)/Divs;
for (int ival=0; ival < Cvals.length-1; ++ival)
  cvals.append(Cvals[ival]+sumarr);
draw(contour(f,a,b,cvals,N,operator --),tickpen);

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

palette("$f(x,y)$",range,point(NW)+(0,0.5),point(NE)+(0,1),Top,Palette,
```

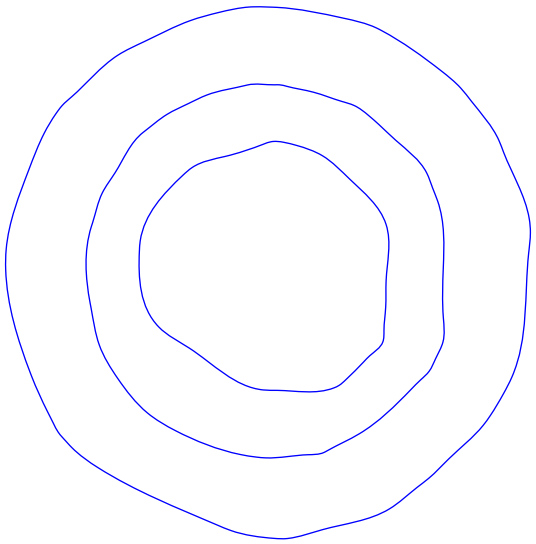
```
PaletteTicks(N=Divs,n=divs,Tickpen,tickpen));
```



Finally, here is an example that illustrates the construction of contours from irregularly spaced data:

```
import contour;

size(200);

int n=100;
pair[] points=new pair[n];
real[] values=new real[n];

real r() {return 1.1*(rand()/randMax*2-1);}

for(int i=0; i < n; ++i)
  points[i]=(r(),r());

real f(real a, real b) {return a^2+b^2;}

for(int i=0; i < n; ++i)
  values[i]=f(points[i].x,points[i].y);
```

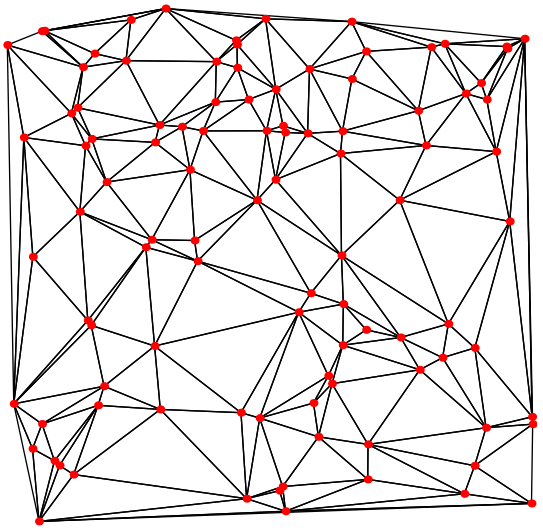
```
draw(contour(points,values,new real[]{0.25,0.5,1},operator ..),blue);
```



In the above example, the contours of irregularly spaced data are constructed by first creating a triangular mesh from an array `z` of pairs, using Gilles Dumoulin's C++ port of Paul Bourke's triangulation code:

```
int[][] triangulate(pair[] z);

size(200);
int np=100;
pair[] points;

real r() {return 1.2*(rand()/randMax*2-1);}

for(int i=0; i < np; ++i)
  points.push((r(),r()));

int[][] trn=triangulate(points);

for(int i=0; i < trn.length; ++i) {
  draw((points[trn[i][0]])--(points[trn[i][1]]));
  draw((points[trn[i][1]])--(points[trn[i][2]]));
  draw((points[trn[i][2]])--(points[trn[i][0]]));
}

for(int i=0; i < np; ++i)
```

```
dot(points[i],red);
```



The example `Gouraudcontour` illustrates how to produce color density images over such irregular triangular meshes.

7.36 contour3

This package draws surfaces described as the null space of real-valued functions of (x, y, z) or real[][][] matrices. Its usage is illustrated in the example file `magnetic.asy`.

7.37 slopefield

To draw a slope field for the differential equation $dy/dx = f(x, y)$ (or $dy/dx = f(x)$), use:

```
picture slopefield(real f(real,real), pair a, pair b,
                   int nx=nmesh, int ny=nx,
                   real tickfactor=0.5, pen p=currentpen);
```

Here, the points `a` and `b` are the lower left and upper right corners of the rectangle in which the slope field is to be drawn, `nx` and `ny` are the respective number of ticks in the x and y directions, `tickfactor` is the fraction of the minimum cell dimension to use for drawing ticks, and `p` is the pen to use for drawing the slope fields. The return value is a picture that can be added to `currentpicture` via the `add(picture)` command.

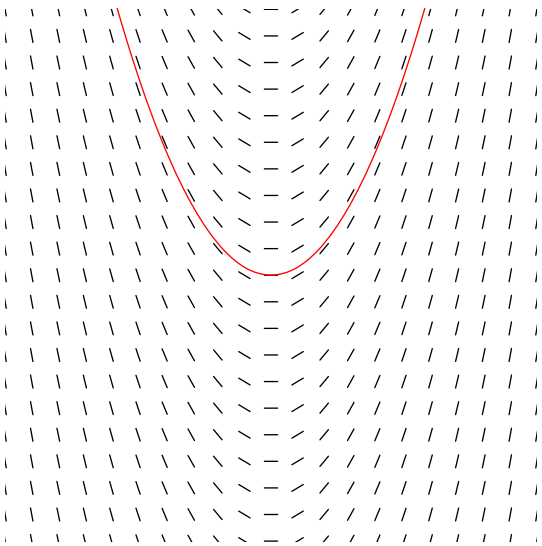
The function

```
path curve(pair c, real f(real,real), pair a, pair b);
```

takes a point (`c`) and a slope field-defining function `f` (`real f(real)` is also accepted), and returns, as a path, the curve passing through that point. The points `a` and `b` represent the rectangular boundaries over which the curve is interpolated. Here is an example of a slope field:

```
import slopefield;

size(200);
```

```
real func(real x) {return 2x;}
add(slopefield(func,(-3,-3),(3,3),20));
draw(curve((0,0),func,(-3,-3),(3,3)),red);
```



8 Options

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

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

Options (negate by replacing - with -no):

-V,-View               View output; command-line only
-a,-align C|B|T|Z      Center, Bottom, Top, or Zero page alignment [Center]
-autoimport string     Module to automatically import [""]
-autoplain             Enable automatic importing of plain [true]
-autorotate            Enable automatic PDF page rotation [false]
-batchMask             Mask fpu exceptions in batch mode [false]
-batchView             View output in batch mode [false]
-bw                    Convert all colors to black and white [false]
-cd directory          Set current directory; command-line only
-c,-clearGUI           Clear GUI operations [false]
-cmyk                  Convert rgb colors to cmyk [false]
-compact               Conserve memory at the expense of speed [false]
-d,-debug              Enable debugging messages [false]
-x,-deconstruct X      Deconstruct into transparent objects magnified by X
-divisor n             Free space divisor for garbage collection [2]
-globalwrite           Allow write to other directory; command-line only [false]
-gray                  Convert all colors to grayscale [false]
-h,-help               Show summary of options; command-line only
-historylines n        Retain n lines of history [1000]
-i,-ignoreGUI          Ignore GUI operations [false]
-inlinetex             Generate inline tex code [false]
-interactiveMask       Mask fpu exceptions in interactive mode [true]
-interactiveView       View output in interactive mode [true]
-k,-keep               Keep intermediate files [false]
-keepaux               Keep intermediate LaTeX .aux files [false]
-l,-listvariables      List available global functions and variables [false]
-localhistory          Use a local interactive history file [false]
-m,-mask               Mask fpu exceptions; command-line only
-multiline             Input code over multiple lines at the prompt [false]
-multipleView          View output from multiple batch-mode files [false]
-O,-offset pair        PostScript offset [(0,0)]
-f,-outformat format   Convert each output file to specified format
-o,-outname name       Alternative output name for first file
-p,-parseonly          Parse file [false]
-prompt string         Prompt ["> "]
-prompt2 string        Continuation prompt for multiline input [".."]
-q,-quiet              Suppress welcome message [false]
-rgb                   Convert cmyk colors to rgb [false]
-safe                  Disable system call; command-line only [true]
-scroll n              Scroll standard output n lines at a time [0]
```

```
-tabcompletion        Interactive prompt auto-completion [true]
-tex engine           TeX engine ("latex|pdflatex|tex|pdftex|none") ["latex"]
-s,-translate         Show translated virtual machine code [false]
-twice                Run LaTeX twice (to resolve references) [false]
-unsafe               Enable system call (=> global); command-line only [false]
-u,-user string       General purpose user string [""]
-v,-verbose           Increase verbosity level [0]
-version              Show version; command-line only
-wait                 Wait for child processes to finish before exiting [false]
-where                Show where listed variables are declared [false]
-xformat format       GUI deconstruction format ["gif"]
```

All boolean options, except for `-unsafe`, `-safe`, and `-help`, can be negated by prepending `no` to the option name.

If no arguments are given, `Asymptote` runs in interactive mode (see Chapter 9 [Interactive mode], page 141). 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.

If the string `autoimport` is nonempty, a module with this name is automatically imported for each run as the final step in loading module `plain`.

Default option values may be entered as `Asymptote` code in a configuration file named `config.asy` (or the file specified by the environment variable `ASYMPTOTE_CONFIG` or `-config` option). `Asymptote` will look for this file in its usual search path. Typically the configuration file is placed in the `.asy` directory in the user's home directory (`%USERPROFILE%\.asy` under `MSDOS`). Configuration variables are accessed using the long form of the option names:

```
import settings;
outformat="pdf";
batchView=false;
interactiveView=true;
batchMask=false;
interactiveMask=true;
```

Command-line options override these defaults. Most configuration variables may also be changed at runtime. The advanced configuration variables `dvipsOptions` and `gsOptions` allow specialized options to be passed as a string to `dvips` and `gs`, respectively.

If you insert

```
import plain;
settings.autoplain=true;
```

at the beginning of the configuration file, it can contain arbitrary `Asymptote` code.

The default output format is EPS for the (default) `latex` tex engine and PDF for the `pdflatex` tex engine. An alternative output format (without antialiasing) 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; an `Invalid Parameter` error message indicates that the `MSDOS` utility `convert` is being used instead of the one that comes with `ImageMagick`). To give specific options to the `convert` program, call convert manually. This example enables antialiasing and produces a `tiff` format at double the usual size:

```
asy -o - venn | convert -density 288x288 -geometry 100%x eps:- venn.tiff
```

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

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 default value of the page alignment setting `align` is `Center`.

The `-u` option may be used to specify arbitrary `Asymptote` code on the command line as a string. It is not necessary to terminate the string with a semicolon. Multiple `-u` options are executed in the order they are given. Command-line code like `-u x=sqrt(2)` can be executed within a module like this:

```
real x;
usersetting();
write(x);
```

When the `-l` option is used with file arguments, it only lists global functions and variables defined in the specified file(s).

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

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

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

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

`-vvvv` Also output knot guide solver diagnostics.

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

9 Interactive mode

Interactive mode is entered by executing the command `asy` with no file arguments. When the `-multiline` option is disabled (the default), each line must be a complete `Asymptote` statement (unless explicitly continued by a final backslash character \); it is not necessary to terminate input lines with a semicolon. If one assigns `settings.multiline=true`, interactive code can be entered over multiple lines; in this mode, the automatic termination of interactive input lines by a semicolon is inhibited. Multiline mode is useful for cutting and pasting `Asymptote` code directly into the interactive input buffer.

Interactive mode can be conveniently used as a calculator: expressions entered at the interactive prompt (for which a corresponding `write` function exists) are automatically evaluated and written to `stdout`.

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

`help` view the manual;

`reset` reset the `Asymptote` environment to its initial state, except for changes to the settings module (see [settings], page 139), the current path (see [cd], page 48), and breakpoints (see Chapter 13 [Debugger], page 147);

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

`quit` exit interactive mode (`exit` is a synonym; the abbreviation `q` is also accepted unless there exists a top-level variable named `q`). A history of the most recent 1000 (this number can be changed with the `historylines` configuration variable) 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, with command history and auto-completion. To customize the key bindings, see: http://cnswww.cns.cwru.edu/php/chet/readline/readline.html

The file `asymptote.py` in the `Asymptote` system directory provides an alternative way of entering `Asymptote` commands interactively, coupled with the full power of `Python`. Copy this file to your `Python path` and then execute from within `Python` the commands

```
from asymptote import *
g=asy()
g.size(200)
g.draw("unitcircle")
g.send("draw(unitsquare)")
```

```
g.fill("unitsquare, blue")
g.clip("unitcircle")
g.label("\"$O$\", (0,0), SW")
```

10 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`.

10.1 GUI Installation

As `xasy` is written in the interactive scripting language `Python/TK`, it requires that both `Python` and the `tkinter` package (included with `Python` under `MSDOS`) be installed. `Fedora Core` users can either install `tkinter` with the command

```
yum install tkinter
```

or manually install the `tkinter`, `tix`, and `tk` packages.

By default, `xasy` uses the GIF image format with white as the transparent color. To overcome certain side-effects, support for an alternative transparent image format, PNG, is also available.

To enable full alpha channel support for PNG images in the Python Imaging Library (PIL) used by `xasy`, on `UNIX` systems place http://effbot.org/downloads/Imaging-1.1.6.tar.gz in the `Asymptote` source directory, and type (as the root user):

```
tar -zxf Imaging-1.1.6.tar.gz
cd Imaging-1.1.6
patch -p1 < ../patches/TkAlpha-Imaging-1.1.6.patch
python setup.py install
```

To configure `Asymptote` to use PNG files for picture deconstruction, these lines need to be in the `Asymptote` configuration file (see [configuration file], page 139):

```
import settings;
xformat="png";
```

10.2 GUI Usage

To use `xasy`, one must first deconstruct `Asymptote` pictures into transparent GIF (or PNG) 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)`. Deconstruction using transparent GIF files requires that `ImageMagick` (see [convert], page 139) be installed.

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.

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 45).

11 PostScript to Asymptote

The excellent PostScript editor pstoedit (version 3.44 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, PostScript subpath, and full image 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.

12 Help

A list of frequently asked questions (FAQ) is maintained at

http://asymptote.sourceforge.net/FAQ

Questions on installing and using Asymptote that are not addressed in the FAQ should be sent to the Asymptote forum:

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

Including an example that illustrates what you are trying to do will help you get useful feedback. LaTeX problems can often be diagnosed with the -vv or -vvv command-line options. 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 Subversion developmental code (see Section 2.8 [Subversion], page 7) before submitting a bug report. New bugs can be submitted using the Bug Tracking System at

http://sourceforge.net/projects/asymptote

To see if the bug has already been fixed, check bugs with Status Closed and recent lines in

http://asymptote.sourceforge.net/ChangeLog

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 59) 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).

13 Debugger

Asymptote now includes a line-based (as opposed to code-based) debugger that can assist the user in following flow control. To set a break point in file `file` at line `line`, use the command

```
void stop(string file, int line, code s=quote{});
```

The optional argument `s` may be used to conditionally set the variable `ignore` in `plain_debugger.asy` to `true`. For example, the first 10 instances of this breakpoint will be ignored (the variable `int count=0` is defined in `plain_debugger.asy`):

```
stop("test",2,quote{ignore=(++count <= 10);});
```

To set a break point in file `file` at the first line containing the string `text`, use

```
void stop(string file, string text, code s=quote{});
```

To list all breakpoints, use:

```
void breakpoints();
```

To clear a breakpoint, use:

```
void clear(string file, int line);
```

To clear all breakpoints, use:

```
void clear();
```

The following commands may be entered at the debugging prompt:

`h`	help;
`c`	continue execution;
`i`	step to the next instruction;
`s`	step to the next executable line;
`n`	step to the next executable line in the current file;
`f`	step to the next file;
`r`	return to the file associated with the most recent breakpoint;
`t`	toggle tracing (`-vvvvv`) mode;
`q`	quit debugging and end execution;
`x`	exit the debugger and run to completion.

Arbitrary `Asymptote` code may also be entered at the debugging prompt; however, since the debugger is implemented with `eval`, currently only top-level (global) variables can be displayed or modified.

The debugging prompt may be entered manually with the call

```
void breakpoint(code s=quote{});
```

14 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. Other contributors include Radoslav Marinov, Chris Savage, Philippe Ivaldi, Jacques Pienaar, Mark Henning, Steve Melenchuk, Martin Wiebusch, and Stefan Knorr.

Index

!

! 55
!= 51, 55

%

% 55
%= 56

&

& 13, 55
&& 55

*

* 34, 55
** 55
*= 56

+

+ 34, 55
++ 56
+= 56

-

- 55
-- 11, 56
--- 13
-= 56
-l 140
-u 140
-V 4, 9

.

.. .. 11
.asy 5

/

/ 55
/= 56

:

: 55
:: 12

<

< 55
<= 55

=

== 51, 55

>

> 55
>= 55

?

? 55

^

^ 55
^= 56
^^ 13

|

| 55
|| 55

2

2D graphs 86

3

3D graphs 121
3D grids 126

A

a4 5
abort 28
abs 24, 62
access 69
acknowledgments 148
acos 62
aCos 62
acosh 62
add 45
Ai 62
Ai_deriv 62
Airy 62
alias 51
alias 65
align 140
all 67
Allow 41

and ... 12
AND ... 55
angle ... 24
animate ... 5, 83
animation ... 83
annotate ... 86
antialiasing ... 139
append ... 48
append ... 63
ArcArrow ... 15
ArcArrows ... 15
arclength ... 30, 119
arctime ... 30, 119
arguments ... 59
arithmetic operators ... 55
arrays ... 62
arrow ... 15, 19
Arrow ... 15
arrow keys ... 9
Arrows ... 15
as ... 70
asin ... 62
aSin ... 62
asinh ... 62
aspect ... 121
Aspect ... 42
assignment ... 23
asy ... 71
asy-mode ... 7
asy.vim ... 7
asymptote.sty ... 74
ASYMPTOTE_CONFIG ... 139
atan ... 62
aTan ... 62
atan2 ... 62
atanh ... 62
atleast ... 12
attach ... 93
autoimport ... 139
autolimits ... 121
automatic scaling ... 99
axial shading ... 17
axis ... 106, 108, 123, 124
azimuth ... 25

B

babel ... 85
background color ... 44
Bar ... 15
Bars ... 15
base modules ... 79
basealign ... 37
baseline ... 20
batch mode ... 9
bbox3 ... 121
BeginArcArrow ... 15
BeginArrow ... 15
BeginBar ... 15
BeginDotMargin ... 15
BeginMargin ... 15
BeginPenMargin ... 15
BeginPoint ... 19
Bessel ... 62
bevel ... 129
beveljoin ... 37
Bi ... 62
Bi_deriv ... 62
binary format ... 49
binary operators ... 55
binarytree ... 82
binput ... 49
Blank ... 15
block.bottom ... 128
block.bottomleft ... 128
block.bottomright ... 128
block.center ... 128
block.draw ... 128
block.left ... 128
block.position ... 128
block.right ... 128
block.top ... 128
block.topleft ... 128
block.topright ... 128
bool ... 23
boolean operators ... 55
Bottom ... 88
BottomTop ... 88
bounding box ... 44
boutput ... 49
box ... 44
bp ... 9
break ... 23
breakpoints ... 147
brick ... 38
broken axis ... 103
bug reports ... 146
buildcycle ... 31
Button-1 ... 143
Button-2 ... 143
BWRainbow ... 112
BWRainbow2 ... 112

C

C string ... 26
CAD ... 86
calculateTransform ... 43
casts ... 68
cbrt ... 62
cd ... 48
ceil ... 62
Center ... 19
checker ... 38
Chinese ... 85
Ci ... 62

circle ... 28, 129
circlebarframe ... 80
CJK ... 85
clear ... 49, 147
clip ... 18
cm ... 10
cmyk ... 35
colatitude ... 25
color ... 34
colorless ... 35
colors ... 35
comma-separated-value mode ... 68
command-line options ... 5, 138
comment character ... 48
compass directions ... 11
Compiling from UNIX source ... 6
complement ... 64
concat ... 64
conditional ... 23, 55
config ... 139
configuration file ... 4
configuring ... 4
conj ... 24
constructors ... 52
continue ... 23, 147
contour ... 130
contour3 ... 136
controls ... 12, 116
controlSpecifier ... 33
convert ... 5, 83, 139
Coons shading ... 18
copy ... 64
cos ... 62
Cos ... 62
cosh ... 62
cputime ... 54
crop ... 98
cropping graphs ... 98
cross ... 25, 96
crossframe ... 80
crosshatch ... 39
csv ... 68
cubicroots ... 67
curl ... 12, 116
curlSpecifier ... 34
currentpen ... 34
currentprojection ... 118
custom axis types ... 88
custom tick locations ... 90
cycle ... 9, 11, 116
cyclic ... 29, 33, 63, 119
cyclicflag ... 63
Cyrillic ... 85

D

dashdotted ... 36
dashed ... 36
data types ... 23
date ... 27
Debian ... 3
debugger ... 147
declaration ... 23
default arguments ... 59
defaultpen ... 35, 36, 37, 41
degrees ... 24, 62
Degrees ... 62
delete ... 63
description ... 1
diagonal ... 79
diamond ... 129
dimension ... 68
dir ... 5, 24, 25, 30, 119
direction specifier ... 11
directory ... 48
dirSpecifier ... 33
dirtime ... 30
display ... 5
do ... 23
dot ... 16, 24, 25
DotMargin ... 15
DotMargins ... 15
dotted ... 36
double ... 49
draw ... 15, 17
Draw ... 15, 44
drawing commands ... 15
drawline ... 79
drawtree ... 83
dvips ... 5
dvipsOptions ... 139

E

E ... 11, 62
Editing modes ... 7
Ei ... 62
elliptic functions ... 62
else ... 23
emacs ... 7
embed ... 84
EndArcArrow ... 15
EndArrow ... 15
EndBar ... 15
EndDotMargin ... 15
endl ... 49
EndMargin ... 15
EndPenMargin ... 15
EndPoint ... 19
environment variables ... 5
eof ... 49, 67
eol ... 49, 67
EPS ... 20
erase ... 26, 47
erf ... 62
erfc ... 62

error ... 48, 49
errorbars ... 96
eval ... 71
evenodd ... 13
evenodd ... 37
exit ... 141, 147
exp ... 62
expi ... 24, 25
explicit ... 68
explicit casts ... 68
exponential integral ... 62
extendcap ... 37
extension ... 79
external ... 84

F

F ... 62
fabs ... 62
face ... 119
featpost3D ... 128
Fedora Core ... 3
feynman ... 83
fft ... 65
FFTW ... 6
file ... 48, 147
fill ... 17
Fill ... 15, 44
filldraw ... 17
FillDraw ... 15, 44
filloutside ... 17
fillrule ... 37
find ... 26, 64
firstcut ... 31
fit ... 43
fix-cm ... 37
fixedscaling ... 43
floor ... 62
flowchart ... 128
flush ... 49
fmod ... 62
font ... 37
font command ... 37
fontcommand ... 38
fontsize ... 37
for ... 23
format ... 27, 139
forum ... 146
frame ... 42
from ... 69
function declarations ... 58
functions ... 57, 62

G

gamma ... 62
Gaussrand ... 62
geometry ... 80
getc ... 48
getreal ... 49
getstring ... 49
GNU Scientific Library ... 62
Gouraud shading ... 17
gradient shading ... 17
graph ... 86
graph3 ... 121
graphic ... 20
graphical user interface ... 143
gray ... 34
grayscale ... 34
Grayscale ... 112
grid ... 38, 101
grid3 ... 126
gs ... 4
gsl ... 62
GSL ... 6
gsOptions ... 139
GUI ... 143
GUI installation ... 143
GUI usage ... 143
guide ... 32
guide3 ... 116

H

hatch ... 39
help ... 141
help ... 146
help ... 147
Hermite ... 86
Hermite(splinetype splinetype ... 86
hidden surface removal ... 119
histogram ... 62
historylines ... 141
hypot ... 62

I

I ... 62
i_scaled ... 62
identity ... 41, 62, 66
if ... 23
IgnoreAspect ... 43
image ... 112
ImageMagick ... 5, 83, 139
implicit casts ... 68
implicit linear solver ... 79
implicit scaling ... 57
import ... 70
inches ... 10
including images ... 20
inheritance ... 54
initialized ... 63
initializers ... 50
input ... 48, 141
insert ... 26, 63

inside ... 32, 119
inst ... 147
installation ... 3
int ... 23
integer division ... 55
interactive mode ... 141
international characters ... 85
interpolate ... 80
intersect ... 31, 79, 119
intersectionpoint ... 31, 79
intersectionpoints ... 31
intersectpoint ... 119
intMax ... 23
inverse ... 41, 66
invert ... 118
invisible ... 35

J

J ... 62
Japanese ... 85

K

K ... 62
k_scaled ... 62
keywords ... 59
Korean ... 85

L

label ... 18
Label ... 16
Label ... 91
labelpath ... 85
labelx ... 91
labely ... 91
Landscape ... 44
lastcut ... 31
lasy-mode ... 7
LaTeX fonts ... 37
LaTeX usage ... 74
latin1 ... 85
latitude ... 25
lattice shading ... 17
layer ... 15
leastsquares ... 80, 107
Left ... 90
LeftRight ... 90
LeftSide ... 19
LeftTicks ... 88, 90
legend ... 15
legend ... 94
Legendre ... 62
length ... 24, 25, 26, 29, 33, 63, 119
letter ... 5
libm routines ... 62
libsigsegv ... 59, 146
light ... 121
limits ... 98, 121
line ... 67
line mode ... 67
Linear ... 99
linecap ... 37
linejoin ... 37
lineskip ... 37
linewidth ... 36
log ... 62
Log ... 99
log-log graph ... 100
log10 ... 62
log2 graph ... 102
logarithmic graph ... 100
logical operators ... 55
longdashdotted ... 36
longdashed ... 36
longitude ... 25
loop ... 23

M

MacOS X binary distributions ... 3
makepen ... 40
map ... 65
Margin ... 15
Margins ... 15
markangle ... 81
marker ... 96
markers ... 80
marknodes ... 96
markuniform ... 96
math ... 79
mathematical functions ... 62
max ... 32, 65, 119
maxbound ... 24, 26
merge ... 83
MetaPost ... 85
MetaPost 12
MetaPost cutafter ... 31
MetaPost cutbefore ... 31
MetaPost pickup ... 34
MetaPost whatever ... 79
Microsoft Windows ... 4
MidArrow ... 15
midpoint ... 30
MidPoint ... 19
min ... 31, 65, 119
minbound ... 24, 25
minipage ... 20
miterjoin ... 37
mm ... 10
mouse ... 143
Move ... 41
MoveQuiet ... 41
multdiagonal ... 66

N

N ... 11
named arguments ... 59
new ... 51, 64
newframe ... 42
newton ... 67
next ... 147
NFSS ... 37
nobasealign ... 37
NoFill ... 15, 44
NoMargin ... 15
none ... 49
None ... 15
NOT ... 55
NoTicks ... 88
null ... 51
nullpen ... 18, 44

O

oblique ... 117
obliqueX ... 118
obliqueY ... 118
obliqueZ ... 117
offset ... 140
opacity ... 38
open ... 48
operator ... 56
operator -- ... 86
operator * ... 86
operator cast ... 69
operator ecast ... 69
operator init ... 50, 53
operators ... 55
options ... 138
OR ... 55
orientation ... 44
orthographic ... 118
output ... 48, 139
overloading functions ... 58
overwrite ... 41

P

P ... 62
pack ... 20
packing ... 61
pair ... 9, 23
paperheight ... 5
papertype ... 5
paperwidth ... 5
parametrized curve ... 98
path ... 28, 129
path[] ... 13
path3 ... 116
patterns ... 38
patterns ... 80
pdfviewer ... 4
pen ... 34
PenMargin ... 15
PenMargins ... 15
perpendicular ... 80
perspective ... 118
picture ... 42
picture alignment ... 45
Pl ... 62
plain ... 79
point ... 29, 33, 119
polar ... 25
polargraph ... 87
polygon ... 96
pop ... 63
Portrait ... 44
postcontrol ... 30, 119
postfix operators ... 56
postscript ... 47
PostScript fonts ... 38
PostScript subpath ... 13
pow10 ... 62
precision ... 49
precontrol ... 30, 119
prefix operators ... 56
private ... 51
programming ... 23
psviewer ... 4
pt ... 10
public ... 51
push ... 63
python ... 4
Python usage ... 141

Q

quadraticroots ... 66
quit ... 141, 147
quote ... 71
quotient ... 55

R

radial shading ... 17
RadialShade ... 44
radians ... 62
Rainbow ... 112
rand ... 62
randMax ... 62
read1 ... 68
read2 ... 68
read3 ... 68
reading ... 48
reading string arrays ... 67
readline ... 50
real ... 23
realDigits ... 23
realEpsilon ... 23
realMax ... 23

realMin ... 23
realmult ... 24
rectangle ... 129
recursion ... 59
reflect ... 42
Relative ... 19
relpoint ... 30
reltime ... 30
remainder ... 62
replace ... 27
resetdefaultpen ... 41
rest arguments ... 60
restore ... 47
restricted ... 51
return ... 147
reverse ... 27, 31, 33, 64, 119
rewind ... 49
rfind ... 26
rgb ... 34
Riemann zeta function ... 62
Right ... 90
RightSide ... 19
RightTicks ... 88, 90
Rotate ... 19
Rotate(pair z) ... 19
round ... 62
roundcap ... 37
roundedpath ... 83
roundjoin ... 37
roundrectangle ... 129
RPM ... 3
runtime imports ... 71
Russian ... 85

S

S ... 11
save ... 47
scale ... 42, 99
Scale ... 19, 100
scale3 ... 118
scientific graph ... 92
scroll ... 49
search ... 64
search paths ... 5
Seascape ... 44
secondary axis ... 104
secondaryX ... 104
secondaryY ... 104
seconds ... 27
seek ... 49
seekeof ... 49
segmentation fault ... 146
self operators ... 56
sequence ... 64
settings ... 4, 139
sgn ... 62
shading ... 17
shift ... 41, 42, 118
Shift ... 19
shiftless ... 42
shipout ... 43
Si ... 62
simplex ... 79
sin ... 62
Sin ... 62
single ... 49
sinh ... 62
size ... 29, 33, 42, 43, 119, 139
slant ... 42
Slant ... 19
slice ... 31
slide ... 84
slopefield ... 136
solid ... 36
solids ... 127
solve ... 66
sort ... 65
Spline ... 86
sqrt ... 62
squarecap ... 37
srand ... 62
stack overflow ... 59, 146
static ... 72
stats ... 80
stdin ... 48
stdout ... 48
step ... 147
stickframe ... 80
stop ... 147
straight ... 29, 79, 119
Straight ... 86
strftime ... 27
string ... 26, 28
strptime ... 27
struct ... 51
structures ... 51
subpath ... 31, 119
subpictures ... 43
substr ... 26
Subversion ... 7
sum ... 65
superpath ... 13
Suppress ... 41
SuppressQuiet ... 41
surface ... 128
SVN ... 7
syzygy ... 83

T

tab ... 49
tab completion ... 9
tan ... 62
Tan ... 62
tanh ... 62

tell ... 49
tension ... 12, 116
tensionSpecifier ... 34
tensor product shading ... 18
tex ... 47
TeX fonts ... 38
TEX string ... 26
texcommand ... 5
texpath ... 5
texpreamble ... 47
texreset ... 48
textbook graph ... 92
tgz ... 3
this ... 51
three ... 116
tick ... 91
ticks ... 88
Ticks ... 88, 90
tildeframe ... 80
tile ... 38
tilings ... 38
time ... 27
Top ... 88
trace ... 147
transform ... 41, 118
transparency ... 38
transpose ... 65
tree ... 82
triangle ... 80
tridiagonal ... 66
trigonometric integrals ... 62
triple ... 25
TrueMargin ... 15
tutorial ... 9
typedef ... 28, 58

U

unfill ... 18
UnFill ... 44
unicode ... 85
Uninstall ... 8
unit ... 24, 25
unitcircle ... 11
unitsize ... 10, 43
UNIX binary distributions ... 3
unpacking ... 61
unravel ... 70
update ... 48
UpsideDown ... 44
usepackage ... 48
user coordinates ... 10
user-defined operators ... 56

V

variable initializers ... 50
vector field ... 109
vectorization ... 67
verbatim ... 47
vim ... 7
virtual functions ... 54
void ... 23

W

W ... 11
wheel mouse ... 143
while ... 23
white-space string delimiter mode ... 67
windingnumber ... 32
word ... 67
write ... 48, 68

X

xasy ... 5, 143
xaxis ... 121
xequals ... 90
XEquals ... 90
xinput ... 49
xlimits ... 98, 121
XOR ... 55
xoutput ... 49
xpart ... 24, 25
xscale ... 41
xscale3 ... 118
xtick ... 91
XZero ... 90

Y

Y ... 62
yaxis ... 121
yequals ... 90
YEquals ... 88
ylimits ... 98, 121
ypart ... 24, 25
yscale ... 41
yscale3 ... 118
ytick ... 91
YZero ... 88

Z

zaxis ... 121
zero_Ai ... 62
zero_Ai_deriv ... 62
zero_Bi ... 62
zero_Bi_deriv ... 62
zero_J ... 62
zerowinding ... 37
zeta ... 62
zlimits ... 121
zpart ... 25
zscale3 ... 118