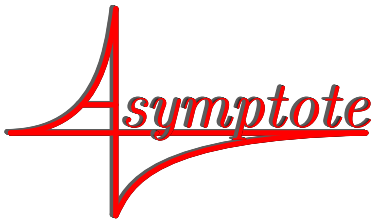
Asymptote: the Vector Graphics Language

For version 3.16



This file documents `Asymptote`, version 3.16.

`https://asymptote.sourceforge.io`

Copyright © 2004-2026 Andy Hammerlindl, John Bowman, and Tom Prince.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Lesser General Public License. On Debian systems, the LGPL can be found at /usr/share/common-licenses/LGPL.

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 3
2.4 Configuring 4
2.5 Search paths 6
2.6 Compiling from UNIX source 6
2.7 Editing modes 7
2.8 Git 7
2.9 Building the documentation 8
2.10 Uninstall 8

3 Tutorial 9

3.1 Drawing in batch mode 9
3.2 Drawing in interactive mode 9
3.3 Figure size 10
3.4 Labels 11
3.5 Paths 11

4 Drawing commands 14

4.1 draw 14
4.2 fill 17
4.3 clip 19
4.4 label 19

5 Bezier curves 23

6 Programming 25

6.1 Data types 25
6.2 Paths and guides 32
6.3 Pens 40
6.4 Transforms 49
6.5 Frames and pictures 50
6.6 Deferred drawing 56
6.7 Files 58
6.8 Variable initializers 61
6.9 Structures 62
6.10 Operators 65
6.10.1 Arithmetic & logical operators 66

6.10.2 Self & prefix operators . . . 67
6.10.3 User-defined operators . . . 67
6.10.4 Bracket operators . . . 68
6.10.5 Cast operators . . . 68
6.11 Implicit scaling . . . 69
6.12 Functions . . . 69
6.12.1 Default arguments . . . 71
6.12.2 Named arguments . . . 71
6.12.3 Rest arguments . . . 72
6.12.4 Mathematical functions . . . 74
6.13 Arrays . . . 76
6.13.1 Slices . . . 83
6.14 Casts . . . 85
6.15 Import . . . 86
6.15.1 Templated imports . . . 88
6.16 Iterators . . . 89
6.16.1 range . . . 91
6.17 Static . . . 91
6.18 Autounravel . . . 93
6.18.1 Where fields are autounraveled . . . 93
6.18.2 Where `autounravel` is legal . . . 94
6.19 Hash functions . . . 94
6.19.1 Hashing user-defined structures . . . 94
6.20 Collections (containers) . . . 95
6.20.1 Pairs . . . 96
6.20.2 Iterators and utilities . . . 96
6.20.2.1 `collections.iter(T)` . . . 96
6.20.2.2 Combining (zipping) iterators . . . 97
6.20.2.3 Attaching a counter to an iterator . . . 99
6.20.3 Maps . . . 99
6.20.3.1 Initializing a `Map` . . . 100
6.20.3.2 The `Map` API . . . 101
6.20.4 Using Maps as Sets . . . 102
6.20.5 Sets . . . 103
6.20.5.1 Initializing a `Set` . . . 103
6.20.5.2 The `Set` API . . . 104
6.20.5.3 The `SortedSet` API . . . 107
6.20.6 Using Sets as Maps . . . 108
6.20.7 Queues . . . 109
6.20.8 Wrapping arrays . . . 110

7 LaTeX usage . . . 112

8 User modules . . . 117

8.1 `graph` . . . 117
8.2 `three` . . . 147

8.3 graph3 161
8.4 palette 168
8.4.1 Surface coloring 173
8.5 colormap 176
8.6 texcolors 177
8.7 x11colors 177
8.8 fontsize 177
8.9 contour 177
8.10 geometry 184
8.11 grid3 184
8.12 interpolate 185
8.13 markers 185
8.14 math 187
8.15 patterns 188
8.16 roundedpath 188
8.17 slopefield 188
8.18 smoothcontour3 189
8.19 solids 190
8.20 stats 191
8.21 tube 191

9 Specialty modules 193

9.1 animation 193
9.2 animate 193
9.3 annotate 193
9.4 babel 193
9.5 binarytree 194
9.6 bsp 195
9.7 CAD 195
9.8 contour3 195
9.9 drawtree 195
9.10 embed 195
9.11 external 195
9.12 feynman 195
9.13 flowchart 196
9.14 labelpath 198
9.15 labelpath3 198
9.16 lmfit 198
9.17 MetaPost 199
9.18 obj 199
9.19 ode 199
9.20 pstoedit 199
9.21 rational 199
9.22 rationalSimplex 199
9.23 simplex 199
9.24 size10 199

9.25 size11 ... 200
9.26 slide ... 200
9.27 syzygy ... 200
9.28 tree ... 200
9.29 trembling ... 200

10 Developer modules ... 201
10.1 plain ... 201
10.2 bezulate ... 201
10.3 simplex2 ... 201
10.4 v3d ... 201

11 Command-line options ... 202

12 Interactive mode ... 208

13 Graphical User Interface ... 210
13.1 GUI installation ... 210
13.2 GUI usage ... 210

14 Command-Line Interface ... 211

15 Language server protocol ... 213

16 PostScript to Asymptote ... 214

17 Help ... 215

18 Debugger ... 216

19 Acknowledgments ... 217

General Index ... 218

1 Description

Asymptote is a powerful descriptive vector graphics language that provides a mathematical coordinate-based framework for technical drawing. Labels and equations are typeset with LaTeX, for overall document consistency, yielding the same high level of typesetting quality that LaTeX provides for scientific text. By default it produces PostScript output, but it can also generate OpenGL, PDF, SVG, WebGL, V3D, and legacy PRC vector graphics, along with any format that the ImageMagick package can produce. You can even try it out in your Web browser without installing it, using the Asymptote Web Application:

https://asymptote.ualberta.ca

It is also possible to send remote commands to this server via the curl utility (see Chapter 14 [Command-Line Interface], page 211).

The Asymptote Wiki at https://asymptote.ualberta.ca/wiki/ is a community gallery where users can share examples and modules. Examples are organized into user-defined categories, compiled with the installed version of Asymptote, and displayed as inline SVG (2D) or interactive WebGL (3D).

A major advantage of Asymptote over other graphics packages is that it is a high-level programming language, as opposed to just a graphics application: 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 14).

Asymptote is mathematically oriented (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;
- the ability to generate and embed 3D vector WebGL graphics within HTML files;
- the ability to generate and embed 3D vector PRC graphics within PDF files;
- a natural coordinate-based framework for technical drawing, 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 `Asymptote` is designed for mathematical typesetting needs, one can write `Asymptote` modules that tailor it to specific applications. For example, a scientific graphing module is available (see Section 8.1 [graph], page 117). Examples of `Asymptote` code and output, including animations, are available at

`https://asymptote.sourceforge.io/gallery/`

Clicking on an example file name in this manual, like `Pythagoras`, will display the PDF output, whereas clicking on its `.asy` extension will show the corresponding `Asymptote` code in a separate window.

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

`https://asymptote.sourceforge.io/links.html`

A quick reference card for `Asymptote` is available at

`https://asymptote.sourceforge.io/asyRefCard.pdf`

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

`https://sourceforge.net/projects/asymptote`

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

`https://sourceforge.net/p/asymptote/discussion/409349`

2.1 UNIX binary distributions

We release both `tgz` and RPM binary distributions of `Asymptote`. The root user can install the `Linux x86_64 tgz` distribution of version `x.xx` of `Asymptote` with the commands:

```
tar -C / -zxf asymptote-x.xx.x86_64.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 users can easily install a recent version of `Asymptote` with the command

```
dnf --enablerepo=rawhide install asymptote
```

To install the latest version of `Asymptote` on a Debian-based distribution (such as Ubuntu) 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 Chathi's prebuilt `Asymptote` binaries from

`https://ftp.debian.org/debian/pool/main/a/asymptote`

2.2 macOS X binary distributions

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

`https://www.macports.org/`

or at

`https://brew.sh/`

Note that many `macOS X` (and FreeBSD) systems lack the GNU `readline` library. For full interactive functionality, GNU `readline` version 4.3 or later must be installed.

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 (we recommend `https://www.tug.org/texlive` or `https://miktex.org`) will be required to typeset labels. You will also need to install `GPL Ghostscript` version 9.56 or later from `https://www.ghostscript.com/`.

To view `PostScript` output, you can install the program `Sumatra PDF` available from `https://www.sumatrapdfreader.org/`.

The ImageMagick package from https://www.imagemagick.org/script/binary-releases.php
is required to support output formats other than HTML, PDF, SVG, and PNG (see [magick], page 205). The Python 3 interpreter from https://www.python.org is only required if you wish to try out the graphical user interface (see Chapter 13 [GUI], page 210).

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 your PostScript viewer (evince under UNIX) to display graphical output. The PostScript viewer should be capable of automatically redrawing whenever the output file is updated. The UNIX PostScript viewer gv supports this (via a SIGHUP signal). Users of ggv will need to enable Watch file under Edit/PostScript Viewer Preferences.

Configuration variables are most easily set as Asymptote variables in an optional configuration file config.asy (see [configuration file], page 205). For example, the setting pdfviewer specifies the location of the PDF viewer. Here are the default values of several important configuration variables under UNIX:

```
import settings;
pdfviewer="xdg-open";
htmlviewer="xdg-open";
psviewer="xdg-open";
display="display";
animate="animate";
gs="gs";
libgs="";
```

Under MSDOS, the viewer settings htmlviewer, pdfviewer, psviewer, display, and animate default to the string cmd, requesting the application normally associated with each file type. The (installation-dependent) default values of gs and libgs are determined automatically from the Microsoft Windows registry. The gs setting specifies the location of the PostScript processor Ghostscript, available from https://www.ghostscript.com/.

The configuration variable htmlviewer specifies the browser to use to display 3D WebGL output. The default setting is google-chrome under UNIX and cmd under Microsoft Windows. Note that Internet Explorer does not support WebGL; Microsoft Windows users should set their default html browser to chrome or microsoft-edge. By default, 2D and 3D HTML images expand to the enclosing canvas; this can be disabled by setting the configuration variable absolute to true.

On UNIX systems, to support automatic document reloading of PDF files in Adobe Reader, we recommend copying the file reload.js from the Asymptote system directory (by default, /usr/share/asymptote under UNIX to ~/.adobe/Acrobat/x.x/JavaScripts/, where x.x represents the appropriate Adobe Reader version number. The automatic document reload feature must then be explicitly enabled by putting

```
import settings;
```

```
pdfreload=true;
pdfreloadOptions="-tempFile";
```

in the Asymptote configuration file. This reload feature is not useful under MSDOS since the document cannot be updated anyway on that operating system until it is first closed by Adobe Reader.

The configuration variable dir can be used to adjust the search path (see Section 2.5 [Search paths], page 6).

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.

These additional configuration variables normally do not require adjustment:

```
config
texpath
texcommand
dvips
dvisvgm
convert
asygl
```

Warnings (such as "unbounded" and "offaxis") may be enabled or disabled with the functions

```
warn(string s);
nowarn(string s);
```

or by directly modifying the string array settings.suppress, which lists all disabled warnings.

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

```
asy -psviewer=evince -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_PAPERTYPE="a4";
```

under Microsoft Windows XP:

1. Click on the Start button;
2. Right-click on My Computer;
3. Choose View system information;
4. Click the Advanced tab;
5. Click the Environment Variables button.

2.5 Search paths

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

1. The current directory;
2. A list of one or more directories specified by the configuration variable dir or environment variable ASYMPTOTE_DIR (separated by : under UNIX and ; under MSDOS);
3. The directory specified by the environment variable ASYMPTOTE_HOME; if this variable is not set, the directory .asy in the user's home directory (%USERPROFILE%\.asy under MSDOS) is used;
4. The Asymptote system directory (by default, /usr/share/asymptote under UNIX and C:\Program Files\Asymptote under MSDOS).
5. The Asymptote examples directory (by default, /usr/share/doc/asymptote/examples under UNIX and C:\Program Files\Asymptote\examples under MSDOS).

2.6 Compiling from UNIX source

To compile and install a UNIX executable from the source release asymptote-x.xx.src.tgz in the subdirectory x.xx under

https://sourceforge.net/projects/asymptote/files/

execute the commands:

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

Then compile Asymptote with the commands

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

We recommend running make all without root privileges (to avoid generating files with incorrect permissions). 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

https://asymptote.sourceforge.io/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
```

One can disable use of the Boehm garbage collector by configuring with ./configure --disable-gc. For a list of other configuration options, say ./configure --help. For example, under macOS X, one can tell configure to use the clang compilers and look for header files and libraries in nonstandard locations:

```
./configure CC=clang CXX=clang++ CPPFLAGS=-I/opt/local/include LDFLAGS=-L/opt/local/lib
```

If you are compiling Asymptote with gcc, you will need a version that supports C++17. For full interactive functionality, you will need version 4.3 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).

2.7 Editing modes

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

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:

```
https://github.com/erlyaws/yaws/blob/master/two-mode-mode.el
```

The hybrid mode lasy-mode can be used to edit a LaTeX file containing embedded Asymptote code (see Chapter 7 [LaTeX usage], page 112). 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 @value{Datadir}/doc/asymptote/examples/asy.vim.gz ~/.vim/syntax/asy.vim.gz
gunzip ~/.vim/syntax/asy.vim.gz
```

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

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

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

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

Syntax highlighting support for the KDE editor Kate can be enabled by running asy-kate.sh in the /usr/share/asymptote directory and putting the generated asymptote.xml file in ~/.local/share/org.kde.syntax-highlighting/syntax/.

2.8 Git

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

```
git clone https://github.com/vectorgraphics/asymptote
```

```
cd asymptote
./autogen.sh
./configure
make all
sudo make install
```

To compile without optimization On `Ubuntu` systems, you may need to first install the required dependencies:

```
apt-get build-dep asymptote
```

2.9 Building the documentation

To build the PDF documentation doc/asymptote.pdf:

```
make man
```

To build the HTML documentation doc/png/index.html:

```
make html
```

The `asy` executable is required for compiling the diagrams in the documentation.

2.10 Uninstall

To uninstall a `Linux x86_64` binary distribution

```
tar -zxvf asymptote-x.xx.x86_64.tgz | xargs --replace=% rm /%
texhash
```

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

```
make uninstall
```

3 Tutorial

A concise introduction to Asymptote is given here. For a more thorough introduction, see the excellent Asymptote tutorial written by Charles Staats:

https://asymptote.sourceforge.io/asymptote_tutorial.pdf

Another Asymptote tutorial is available as a wiki, with images rendered by an online Asymptote engine:

https://www.artofproblemsolving.com/wiki/?title=Asymptote_(Vector_Graphics_Language)

3.1 Drawing in batch mode

To draw a line from coordinate (0,0) to coordinate (100,100), create a text file test.asy containing

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

Then execute the command

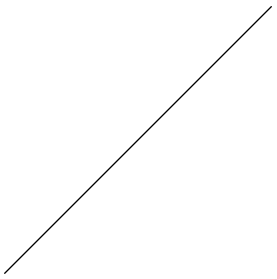
```
asy -V test
```

Alternatively, MSDOS users can drag and drop test.asy onto the Desktop asy icon (or make Asymptote the default application for the extension asy).

This method, known as *batch mode*, outputs a PostScript file test.eps. If you prefer PDF output, use the command line

```
asy -V -f pdf test
```

In either case, the -V option opens up a viewer window so you can immediately view the result:



Here, the -- connector joins the two points (0,0) and (100,100) with a line segment.

3.2 Drawing in interactive mode

Another method is *interactive mode*, where Asymptote reads individual commands as they are entered by the user. To try this out, enter Asymptote's interactive mode by clicking on the Asymptote icon or typing the command asy. Then type

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

followed by Enter, to obtain the above image.

At this point you can type further draw commands, which will be added to the displayed figure, erase to clear the canvas,

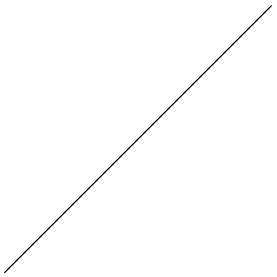
```
input test;
```

to execute all of the commands contained in the file `test.asy`, or `quit` to exit interactive mode. You can use the arrow keys in interactive mode to edit previous lines. 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 12 [Interactive mode], page 208.

3.3 Figure size

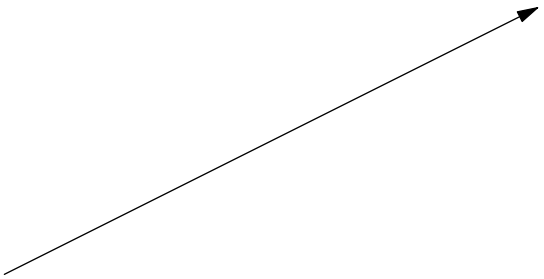
In `Asymptote`, coordinates like `(0,0)` and `(100,100)`, called *pairs*, are expressed in `PostScript` "big points" (1 `bp` = 1/72 `inch`) and the default line width is `0.5bp`. However, it is often inconvenient to work directly in `PostScript` coordinates. The next example produces identical output to the previous example, by scaling the line `(0,0)--(1,1)` to fit a rectangle of width `100.5bp` and height `100.5bp` (the extra `0.5bp` accounts for the line width):

```
size(100.5,100.5);
draw((0,0)--(1,1));
```



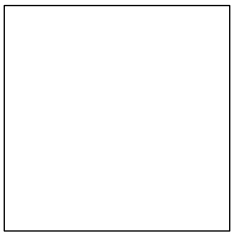
One can also specify the size in `pt` (1 `pt` = 1/72.27 `inch`), `cm`, `mm`, or `inches`. Two nonzero size arguments (or a single size argument) restrict the size in both directions, preserving the aspect ratio. If 0 is given as a size argument, no restriction is made in that direction; the overall scaling will be determined by the other direction (see [size], page 51):

```
size(0,100.5);
draw((0,0)--(2,1),Arrow);
```



To connect several points and create a cyclic path, use the `cycle` keyword:

```
size(3cm);
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))`.

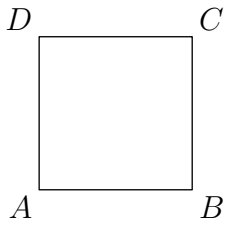
To make the user coordinates represent multiples of exactly `1cm`:

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

3.4 Labels

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

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

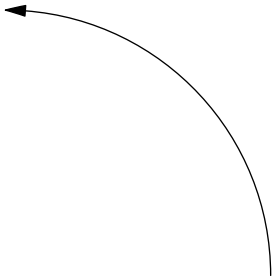


`Asymptote` uses the standard compass directions `E=(1,0)`, `N=(0,1)`, `NE=unit(N+E)`, and `ENE=unit(E+NE)`, etc., which along with the directions `up`, `down`, `right`, and `left` are defined as pairs in the `Asymptote` base module `plain` (a user who has a local variable named `E` may access the compass direction `E` by prefixing it with the name of the module where it is defined: `plain.E`).

3.5 Paths

This example draws a path that approximates a quarter circle, terminated with an arrowhead:

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



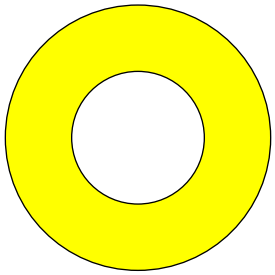
Here the directions `up` and `left` in braces specify the outgoing and incoming directions at the points `(1,0)` and `(0,1)`, respectively.

In general, a path is specified as a list of points (or other paths) interconnected with `--`, which denotes a straight line segment, or `..`, which denotes a cubic spline (see Chapter 5 [Bezier curves], page 23). Specifying a final `..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;
```

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 43). 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 `box(triple, triple)` function in the module `three` to construct the edges of a cube `unitbox` without retracing steps (see Section 8.2 [three], page 147):

```
import three;
```

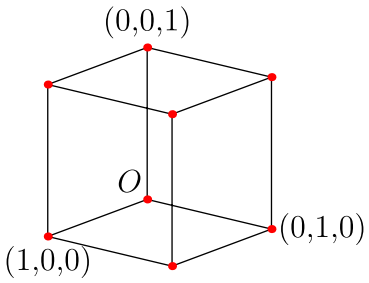
```
currentprojection=orthographic(5,4,2,center=true);

size(5cm);
size3(3cm,5cm,8cm);

draw(unitbox);

dot(unitbox,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),Z);
```



See section Section 8.1 [graph], page 117, (or the online `Asymptote` gallery and external links posted at `https://asymptote.sourceforge.io`) for further examples, including two-dimensional and interactive three-dimensional scientific graphs. Additional examples have been posted by Philippe Ivaldi at `https://blog.piprime.fr/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 52).

If you wish to draw PostScript objects on top of labels (or verbatim tex commands; see [tex], page 56), 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.

A page break can be generated with the command

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

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 bar 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(pen).

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).

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.

For example:

```
draw((0,0)--(1,1),arrow=Arrows);
```

All of the arrow specifiers except for `None` and `Blank` may be given optional arguments, for example:

```
draw((0,0)--(1,1),arrow=Arrow(
     arrowhead=HookHead,size=3mm,angle=20,filltype=Draw,position=0.9));
```

The function `Arrow` has the signature

```
arrowbar Arrow(arrowhead arrowhead=DefaultHead,
               real size=0, real angle=arrowangle,
               filltype filltype=null, position position=EndPoint)
```

Calling `Arrow()` returns `Arrow`, which is an `arrowbar` object. The parameters are:

- `arrowhead` can be one of the predefined arrowhead styles `DefaultHead`, `SimpleHead`, `HookHead`, `TeXHead`.
- real `size` is the arrowhead size in `PostScript` coordinates.

  The default arrowhead size when drawn with a pen `p` is `arrowsize(p)`.
- real `angle` is the arrowhead angle in degrees.
- filltype `filltype` (see [filltype], page 52),
- (except for `MidArrow` and `Arrows`) real `position` (in the sense of `point(path p, real t)`) along the path where the tip of the arrow should be placed.

Margins `margin` 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`
: (equivalently `Margin`)

`Margins`
: leaves a margin at both ends of the path.

`Margin(real begin, real end=begin)`
: specify the size of the beginning and ending margin, respectively, in multiples of the units `labelmargin(p)` used for aligning labels.

`BeginPenMargin`
`EndPenMargin`
: (equivalently `PenMargin`)

`PenMargins`
`PenMargin(real begin, real end=begin)`
: specify a margin in units of the pen line width, taking account of the pen line width when drawing the path or arrow.

`DotMargin`
: an abbreviation for `PenMargin(-0.5*dotfactor,0.5*dotfactor)`, used to draw from the usual beginning point just up to the boundary of an end dot of width `dotfactor*linewidth(p)`.

`BeginDotMargin`
`DotMargins`
: work similarly.

`TrueMargin(real begin, real end=begin)`
: specify a margin directly in `PostScript` units, independent of the pen line width.

The use of arrows, bars, and margins is illustrated by the examples `Pythagoras.asy` and `sqrtx01.asy`. Since bars are intended for indicating precise dimensions, they should only be drawn with NoMargin (the default).

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

```
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, real maxheight=0,
             bool hstretch=false, bool vstretch=false, 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 the legend entry size), `maxwidth` and `maxheight` specify optional upper limits on the width and height of the resulting legend (0 means unlimited), `hstretch` and `vstretch` allow the legend to stretch horizontally or vertically, 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 54).

To draw a dot, simply draw a path containing a single point. The `dot` command defined in the module `plain` draws a dot using the specified filltype (see [filltype], page 52) or `dotfilltype` (`Fill` by default). With `filltype=Fill`, the dot has a diameter equal to an explicit pen line width or the default line width magnified by `dotfactor` (6 by default). With other filltypes, the dot is drawn as a ring with an outer diameter of `dotfactor` times the pen line width, regardless of whether the line width is explicitly specified:

```
void dot(frame f, pair z, pen p=currentpen, filltype filltype=dotfilltype);
void dot(picture pic=currentpicture, pair z, pen p=currentpen,
         filltype filltype=dotfilltype);
void dot(picture pic=currentpicture, Label L, pair z, align align=NoAlign,
         string format=defaultformat, pen p=currentpen, filltype filltype=dotfilltype);
void dot(picture pic=currentpicture, Label[] L=new Label[], pair[] z,
         align align=NoAlign, string format=defaultformat, pen p=currentpen,
         filltype filltype=dotfilltype);
void dot(picture pic=currentpicture, path[] g, pen p=currentpen,
         filltype filltype=dotfilltype);
void dot(picture pic=currentpicture, Label L, pen p=currentpen,
         filltype filltype=dotfilltype);
```

If the variable `Label` is given as the `Label` argument to the third routine, the `format` argument will be used to format a string based on the dot location (here `defaultformat` is

"$%.4g$"). The fourth routine draws a dot at every point of a pair array z. One can also draw a dot at every node of a path:

```
void dot(picture pic=currentpicture, Label[] L=new Label[],
         explicit path g, align align=RightSide, string format=defaultformat,
         pen p=currentpen, filltype filltype=dotfilltype);
```

See [pathmarkers], page 127, and Section 8.13 [markers], page 185, 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 fill rule fillrule, can be produced with

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

If stroke=true, the region filled is the same as the region that would be drawn by draw(pic,g,zerowinding); in this case the path g need not be cyclic. 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, bool stroke=false,
```

```
                pen pena, pair a, bool extenda=true,
                pen penb, pair b, bool extendb=true);
```

The boolean parameters `extenda` and `extendb` indicate whether the shading should extend beyond the axis endpoints `a` and `b`. An example of axial shading is provided in the example file `axialshade.asy`.

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, bool stroke=false,
                 pen pena, pair a, real ra, bool extenda=true,
                 pen penb, pair b, real rb, bool extendb=true);
```

The boolean parameters `extenda` and `extendb` indicate whether the shading should extend beyond the radii `a` and `b`. Illustrations of radial shading are provided in the example files `shade.asy`, `ring.asy`, and `shadestroke.asy`.

Gouraud shading using fill rule `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, bool stroke=false,
                  pen fillrule=currentpen, pen[] p, pair[] z,
                  int[] edges);
void gouraudshade(picture pic=currentpicture, path g, bool stroke=false,
                  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`. The edge flags used in Gouraud shading are documented on pages 270–274 of the PostScript Language Reference (3rd edition):

`https://www.adobe.com/jp/print/postscript/pdfs/PLRM.pdf`

Tensor product shading using clipping path `g`, fill rule `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, bool stroke=false,
                 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, bool stroke=false,
                 pen fillrule=currentpen, pen[] p, path b=g,
                 pair[] z=new pair[]);
```

One can also smoothly shade the regions between consecutive paths of a sequence using a given array of pens:

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

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

More general shading possibilities are available using TEX engines that produce PDF output (see [texengines], page 205): the routine

```
void functionshade(picture pic=currentpicture, path[] g, bool stroke=false,
                   pen fillrule=currentpen, string shader);
```

shades on picture `pic` the interior of path `g` according to fill rule `fillrule` using the `PostScript` calculator routine specified by the string `shader`; this routine takes 2 arguments, each in [0,1], and returns `colors(fillrule).length` color components. Function shading is illustrated in the example `functionshading.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, stroke=false,
          pen fillrule=currentpen);
```

Clip the current contents of picture `pic` to the region bounded by the path `g`, using fill rule `fillrule` (see [fillrule], page 43). If `stroke=true`, the clipped portion is the same as the region that would be drawn with `draw(pic,g,zerowinding)`; in this case the path `g` need not be cyclic. While clipping has no notion of depth (it transcends layers and even pages), one can localize clipping to a temporary picture, which can then be added to `pic`. For an illustration of picture clipping, see the first example in Chapter 7 [LaTeX usage], page 112.

4.4 label

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

Here, `real labelmargin(pen p=currentpen)` is a quantity used to align labels. In the code below,

```
label("abcdefg",(0,0),align=up,basealign);
```

the baseline of the label will be exactly `labelmargin(currentpen)` `PostScript` units above the center.

The constant `Align` can be used to align the bottom-left corner of the label at `position`. 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 6.4 [Transforms], page 49). 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`. An explicit pen specified within the Label overrides other pen arguments. 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=currentpen, filltype filltype=NoFill);
```

By default the label will be positioned at the midpoint of the path. An alternative label position (in the sense of `point(path p, real t)`) may be specified as a 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` and `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, string template="\strut")` function can be used to enlarge the bounding box of a label to match a given template, so that their baselines will be typeset on a horizontal line. See `Pythagoras.asy` for an example. Alternatively, the pen `basealign` may be used to force labels to respect the TeX baseline (see [basealign], page 43).

One can prevent labels from overwriting one another with the `overwrite` pen attribute (see [overwrite], page 49).

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[]);
```

To draw or fill a box (or ellipse or other path) around a `Label` and return the bounding object, use one of the routines

```
object draw(picture pic=currentpicture, Label L, envelope e,
            real xmargin=0, real ymargin=xmargin, pen p=currentpen,
            filltype filltype=NoFill, bool above=true);
object draw(picture pic=currentpicture, Label L, envelope e, pair position,
            real xmargin=0, real ymargin=xmargin, pen p=currentpen,
            filltype filltype=NoFill, bool above=true);
```

Here `envelope` is a boundary-drawing routine such as `box`, `roundbox`, or `ellipse` defined in `plain_boxes.asy`.

The function `path[] texpath(Label L)` returns the path array that TEX would fill to draw the Label `L`.

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:




```
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{\LARGE {\tt Asymptote}\\
\smallskip
\small The Vector Graphics Language}\\
\smallskip
\textsc{Andy Hammerlindl, John Bowman, and Tom Prince}
https://asymptote.sourceforge.io\\
",8cm),(0,0.6));
label(graphic("logo","height=7cm"),(0,-0.22));
clip(unitcircle^^(scale(2/11.7)*unitcircle),evenodd);
```

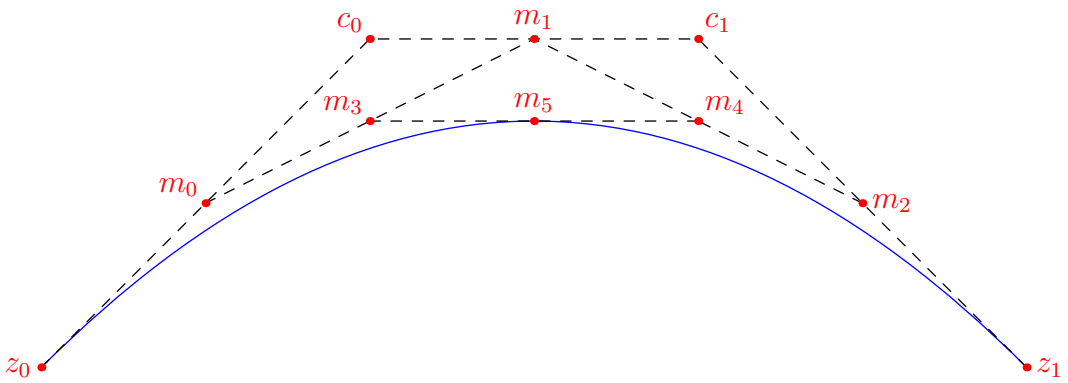
5 Bezier curves

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 midpoint 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 [see John D. Hobby, Discrete and Computational Geometry 1, 123–140 (1986)]:

```
draw((100,0)..tension 2 ..(100,100)..(0,100));
draw((100,0)..tension 3 and 2 ..(100,100)..(0,100));
draw((100,0)..tension atleast 2 ..(100,100)..(0,100));
```

In these examples there is a space between `2` and `...` This is needed as `2.` is interpreted as a numerical constant.

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 6.12.3 [Rest arguments], page 72). 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, after first stripping off the last node of the first path (which normally should coincide with the first node of the second path).

6 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 supports `while`, `do`, `break`, and `continue` statements just as in C/C++. It also supports range-based `for` loops, which are convenient for iterating over all elements of an array:

```
// Iterate over an array
int[] array={1,1,2,3,5};
for(int k : array) {
  write(k);
}
```

In addition, it supports many features beyond the ones found in those languages.

6.1 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` or `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.

`bool3` an extended boolean type that can take on the values `true`, `default`, or `false`. A bool3 type can be cast to or from a bool. The default initializer for bool3 is `default`.

`int` an integer type; if no initializer is given, the implicit value `0` is assumed. The minimum allowed value of an integer is `intMin` and the maximum value 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`. The variables `inf` and `nan`, along with the function `bool isnan(real x)` are useful when floating-point exceptions are masked with the `-mask` command-line option (the default in interactive mode).

`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)`. The function `abs2(pair z)` returns |`z`|^2;

`real angle(pair z, bool warn=true)`
: returns the angle of `z` in radians in the interval [`-pi`,`pi`] or `0` if `warn` is `false` and `z=(0,0)` (rather than producing an error);

`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=(0,0)` (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 degrees)`
: returns a unit vector in the direction `degrees` 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(explicit pair z, explicit pair w)`
: returns the dot product `z.x*w.x+z.y*w.y`;

`real cross(explicit pair z, explicit pair w)`
: returns the 2D scalar product `z.x*w.y-z.y*w.x`;

`real orient(pair a, pair b, pair c);`
: returns a positive (negative) value if `a--b--c--cycle` is oriented counterclockwise (clockwise) or zero if all three points are colinear. Equivalently, a positive (negative) value is returned if `c` lies to the left (right) of the line through `a` and `b` or zero if `c` lies on this line. The value returned can be expressed in terms of the 2D scalar cross product as `cross(a-c,b-c)`, which is the determinant

```
|a.x a.y 1|
|b.x b.y 1|
|c.x c.y 1|
```

`real incircle(pair a, pair b, pair c, pair d);`
: returns a positive (negative) value if `d` lies inside (outside) the circle passing through the counterclockwise-oriented points `a,b,c` or zero if `d` lies on the this circle. The value returned is the determinant

```
|a.x a.y a.x^2+a.y^2 1|
|b.x b.y b.x^2+b.y^2 1|
|c.x c.y c.x^2+c.y^2 1|
|d.x d.y d.x^2+d.y^2 1|
```

`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 its argument `v`. A synonym for `length(triple)` is `abs(triple)`. The function `abs2(triple v)` returns $|v|^2$;

`real polar(triple v, bool warn=true)`
: returns the colatitude of `v` measured from the z axis in radians or `0` if `warn` is `false` and `v=0` (rather than producing an error);

`real azimuth(triple v, bool warn=true)`
: returns the longitude of `v` measured from the x axis in radians or `0` if `warn` is `false` and `v.x=v.y=0` (rather than producing an error);

`real colatitude(triple v, bool warn=true)`
: returns the colatitude of `v` measured from the z axis in degrees or `0` if `warn` is `false` and `v=0` (rather than producing an error);

`real latitude(triple v, bool warn=true)`
: returns the latitude of `v` measured from the xy plane in degrees or `0` if `warn` is `false` and `v=0` (rather than producing an error);

`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 polar, real azimuth)`
: returns a unit triple in the direction `(polar,azimuth)` 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 mappings to allow the use of double quotes in TEX (for using the `babel` package, see Section 9.4 [babel], page 193):

- \" maps to "
- \\ 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.

A single-character substring of a string `s` can be accessed with `s[pos]` for an integer index `pos` in the range `[-n, n-1]`, where `n` is the length of the string `s`. Negative indices count from the end of the string, so `s[-1]` is the last character of `s`. The start of the string is `s[0]`. If `pos` is out of range, the empty string `""` is returned.

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)`
: returns 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)`
: returns 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[] split(string s, string delimiter="")`
: returns an array of strings obtained by splitting `s` into substrings delimited by `delimiter` (an empty delimiter signifies a space, but with duplicate delimiters discarded);

`string[] array(string s)`
: returns an array of strings obtained by splitting `s` into individual characters. The inverse operation is provided by `operator +(...string[] a)`.

`string format(string s, int n, string locale="")`
: returns a string containing `n` formatted according to the C-style format string `s` using locale `locale` (or the current locale if an empty string is specified), following the behavior of the C function `fprintf`), except that only one data field is allowed.

`string format(string s=defaultformat, bool forcemath=false, string s=defaultseparator, real x, string locale="")`
: returns a string containing `x` formatted according to the C-style format string `s` using locale `locale` (or the current locale if an empty string is specified), following the behavior of the C function `fprintf`), except that only one data field is allowed, trailing zeros are removed by default (unless `#` is specified), and if `s` specifies math mode or `forcemath=true`, TEX is used to typeset scientific notation using the `defaultseparator="\!\times\!";`;

`int hex(string s);`
: casts a hexadecimal string `s` to an integer;

`int ascii(string s);`
: returns the ASCII code for the first character of string `s`;

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

`string locale(string s="")`
: sets the locale to the given string, if nonempty, and returns the current locale;

`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);
```

`int system(string s)`
`int system(string[] s)`
: if the setting `safe` is false, call the arbitrary system command `s`;

`void asy(string format, bool overwrite=false, ... string[] s)`
: conditionally process each file name in array `s` in a new environment, using format `format`, overwriting the output file only if `overwrite` is true;

`void abort(string s="")`
: aborts execution (with a non-zero return code in batch mode); if string `s` is nonempty, a diagnostic message constructed from the source file, line number, and `s` is printed;

`void assert(bool b, string s="")`
: aborts execution with an error message constructed from `s` if `b=false`;

`void exit()`
: exits (with a zero error return code in batch mode);

`void sleep(int seconds)`
: pauses for the given number of seconds;

`void usleep(int microseconds)`
: pauses for the given number of microseconds;

`void beep()`
: produces a beep on the console;

As in C/C++, complicated types may be abbreviated with `typedef` or `using`. For instance, the line

```
using multipath = path[];
```

will make `multipath` a type equivalent `path[]`, as will the equivalent line

```
typedef path[] multipath;
```

For the most part such type aliasing is a convenience. However, it is required when declaring a function whose return type is a function (see the example in Section 6.12 [Functions], page 69).

6.2 Paths and guides

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

  For example, 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` (see [unitcircle], page 12):

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

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

```
import graph;
path Circle(pair c, real r, int n=nCircle);
```

  A circular arc consistent with `circle` 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);
```

  One may also specify the direction explicitly:

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

  Here the direction can be specified as CCW (counter-clockwise) or CW (clockwise). For convenience, an arc centered at `c` from pair `z1` to `z2` (assuming `|z2-c|=|z1-c|`) in the may also be constructed with

```
path arc(pair c, explicit pair z1, explicit pair z2,
```

```
        bool direction=CCW)
```

If high accuracy is needed, true arcs may be produced with routines in the module `graph` that produce Bezier curves with `n` control points:

```
import graph;
path Arc(pair c, real r, real angle1, real angle2, bool direction,
        int n=nCircle);
path Arc(pair c, real r, real angle1, real angle2, int n=nCircle);
path Arc(pair c, explicit pair z1, explicit pair z2,
        bool direction=CCW, int n=nCircle);
```

An ellipse can be drawn with the routine

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

A brace can be constructed between pairs `a` and `b` with

```
path brace(pair a, pair b, real amplitude=bracedefaultratio*length(b-a));
```

This example illustrates the use of all five guide connectors discussed in Chapter 3 [Tutorial], page 9, and Chapter 5 [Bezier curves], page 23:

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

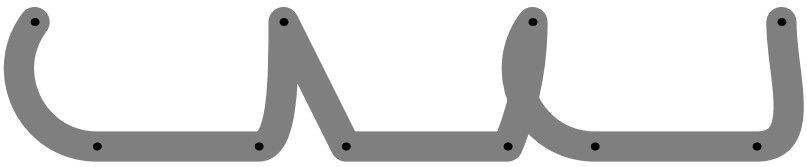
z[0]=(0,100); z[1]=(50,0); z[2]=(180,0);

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.

`bool piecewisestraight(path p)`
: returns `true` iff the path `p` is piecewise 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 Chapter 5 [Bezier curves], page 23). 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, bool normalize=true);`
: If `sign < 0`, return the direction (as a pair) of the incoming tangent to path `p` at node `t`; if `sign > 0`, return the direction of the outgoing tangent. If `sign=0`, the mean of these two directions is returned.

`pair dir(path p, real t, bool normalize=true);`
: 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 Chapter 5 [Bezier curves], page 23).

`pair dir(path p)`
: returns dir(p,length(p)).

`pair dir(path p, path q)`
: returns unit(dir(p)+dir(q)).

`pair accel(path p, int t, int sign=0);`
: If `sign < 0`, return the acceleration of the incoming path `p` at node `t`; if `sign > 0`, return the acceleration of the outgoing path. If `sign=0`, the mean of these two accelerations is returned.

`pair accel(path p, real t);`
: returns the acceleration of the path `p` at the point `t`.

`real radius(path p, real t);`
: returns the radius of curvature of the path `p` at the point `t`.

`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`.

`pair arcpoint(path p, real L);`
: returns `point(p,arctime(p,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=-1);`
: 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). The computations are performed to the absolute error specified by fuzz, or if fuzz < 0, to machine precision. If the paths do not intersect, return a real array of length 0.

`real[][] intersections(path p, path q, real fuzz=-1);`
returns all (unless there are infinitely many) intersection times of paths p and q as a sorted array of real arrays of length 2 (see [sort], page 79). The computations are performed to the absolute error specified by fuzz, or if fuzz < 0, to machine precision.

`real[] intersections(path p, explicit pair a, explicit pair b, real fuzz=-1);`
returns all (unless there are infinitely many) intersection times of path p with the (infinite) line through points a and b as a sorted array. The intersections returned are guaranteed to be correct to within the absolute error specified by fuzz, or if fuzz < 0, to machine precision.

`real[] times(path p, real x)`
returns all intersection times of path p with the vertical line through (x,0).

`real[] times(path p, explicit pair z)`
returns all intersection times of path p with the horizontal line through (0,z.y).

`real[] mintimes(path p)`
returns an array of length 2 containing times at which path p reaches its minimal horizontal and vertical extents, respectively.

`real[] maxtimes(path p)`
returns an array of length 2 containing times at which path p reaches its maximal horizontal and vertical extents, respectively.

`pair intersectionpoint(path p, path q, real fuzz=-1);`
returns the intersection point point(p,intersect(p,q,fuzz)[0]).

`pair[] intersectionpoints(path p, path q, real fuzz=-1);`
returns an array containing all intersection points of the paths p and q.

`pair extension(pair P, pair Q, pair p, pair q);`
returns the intersection point of the extensions of the line segments P--Q and p--q, or if the lines are parallel, (infinity,infinity).

`slice cut(path p, path knife, int n);`
returns the portions of path p before and after the nth intersection of p with path knife as a structure slice (if no intersection exist is found, the entire path is considered to be 'before' the intersection):

```
struct slice {
```

```
  path before,after;
}
```

The argument `n` is treated as modulo the number of intersections.

`slice firstcut(path p, path knife);`
: equivalent to `cut(p,knife,0);` Note that `firstcut.after` plays the role of the `MetaPost cutbefore` command.

`slice lastcut(path p, path knife);`
: equivalent to `cut(p,knife,-1);` 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 two or more consecutively intersecting paths, following the behavior 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 `p` relative to the point `z`. The winding number is positive if the path encircles `z` in the counterclockwise direction. If `z` lies on `p` the constant `undefined` (defined to be the largest odd integer) is returned.

`bool interior(int windingnumber, pen fillrule)`
: returns true if `windingnumber` corresponds to an interior point according to `fillrule`.

`bool inside(path p, pair z, pen fillrule=currentpen);`
: returns `true` iff the point `z` lies inside or on the edge of the region bounded by the cyclic path `p` according to the fill rule `fillrule` (see [fillrule], page 43).

`int inside(path p, path q, pen fillrule=currentpen);`
: returns `1` if the cyclic path `p` strictly contains `q` according to the fill rule `fillrule` (see [fillrule], page 43), `-1` if the cyclic path `q` strictly contains `p`, and `0` otherwise.

`pair inside(path p, pen fillrule=currentpen);`
: returns an arbitrary point strictly inside a nondegenerate cyclic path `p` according to the fill rule `fillrule` (see [fillrule], page 43).

`path[] strokepath(path g, pen p=currentpen);`
: returns the path array that `PostScript` would fill in drawing path `g` with pen `p`.

`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);

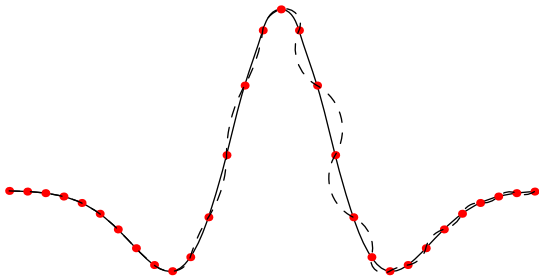
real mexican(real x) {return (1-8x^2)*exp(-(4x^2));}

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 p)`.

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

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

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

`guide reverse(guide g);`
: Analogous to `reverse(path p)`. 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;
```

6.3 Pens

In `Asymptote`, pens provide a context for the four basic drawing commands (see Chapter 4 [Drawing commands], page 14). 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 nonassociative binary operator `+`. This will add 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 RGB(int r, int g, int b);`
  : This produces an RGB color, where each of the red, green, and blue intensities `r`, `g`, `b`, lies in the interval [0,255].

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

  `pen invisible;`
  : This special pen writes in invisible ink, but adjusts the bounding box as if something had been drawn (like the `\phantom` command in TEX). The function `bool invisible(pen)` can be used to test whether a pen is invisible.

  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. The function `colorless(pen=currentpen)` returns a copy of its argument with the color attributes stripped (to avoid color mixing).

A 6-character RGB hexadecimal string can be converted to a pen with the routine

```
pen rgb(string s);
```

- A pen can be converted to a hexadecimal string with `string hex(pen p);`

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 (such as `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 9.26 [slide], page 200) it may be desirable to specify the color directly to LaTeX. This file can be passed to LaTeX with the `Asymptote` command

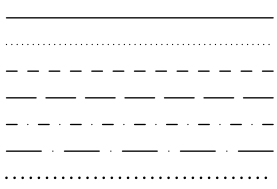
```
usepackage("asycolors");
```

The structure `hsv` defined in `plain_pens.asy` may be used to convert between HSV and RGB spaces, where the hue `h` is an angle in $[0, 360)$ and the saturation `s` and value `v` lie in `[0,1]`:

```
pen p=hsv(180,0.5,0.75);
write(p);           // ([default], red=0.375, green=0.75, blue=0.75)
hsv q=p;
write(q.h,q.s,q.v); // 180     0.5     0.75
```

- Line types are specified with the function `pen linetype(real[] a, real offset=0, bool scale=true, bool adjust=true)`, where `a` is an array of real array numbers. 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 line width; 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(new real[]);
pen dotted=linetype(new real[] {0,4});
pen dashed=linetype(new real[] {8,8});
pen longdashed=linetype(new real[] {24,8});
pen dashdotted=linetype(new real[] {8,8,0,8});
pen longdashdotted=linetype(new real[] {24,8,0,8});
pen Dotted(pen p=currentpen) {return linetype(new real[] {0,3})+2*linewidth(p);}
pen Dotted=Dotted();
```



The default line type is `solid`; this may be changed with `defaultpen(pen)`. The line type of a pen can be determined with the functions `real[] linetype(pen p=currentpen)`, `real offset(pen p)`, `bool scale(pen p)`, and `bool adjust(pen p)`.

- 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_pens` we define

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

so that one may set the line width 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)`. The line cap of a pen is returned by `int linecap(pen p=currentpen)`.

- 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)`.The join style of a pen is returned by `int linejoin(pen p=currentpen)`.

- A pen with a specific PostScript miter limit is returned by calling `miterlimit(real)`. The default miterlimit, `10.0`, may be changed with `defaultpen(pen)`. The miter limit of a pen is returned by `real miterlimit(pen p=currentpen)`.

- 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)`. The fill rule of a pen is returned by `int fillrule(pen p=currentpen)`.

- 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 base align setting of a pen is returned by `int basealign(pen p=currentpen)`.

  For example, in the following image, the baselines of green π and γ are aligned, while the bottom border of red $-\pi$ and $-\gamma$ are aligned.

  ```
  import fontsize;
  ```

```
import three;

settings.autobillboard=false;
settings.embed=false;
currentprojection=orthographic(Z);

defaultpen(fontsize(100pt));

dot(O);

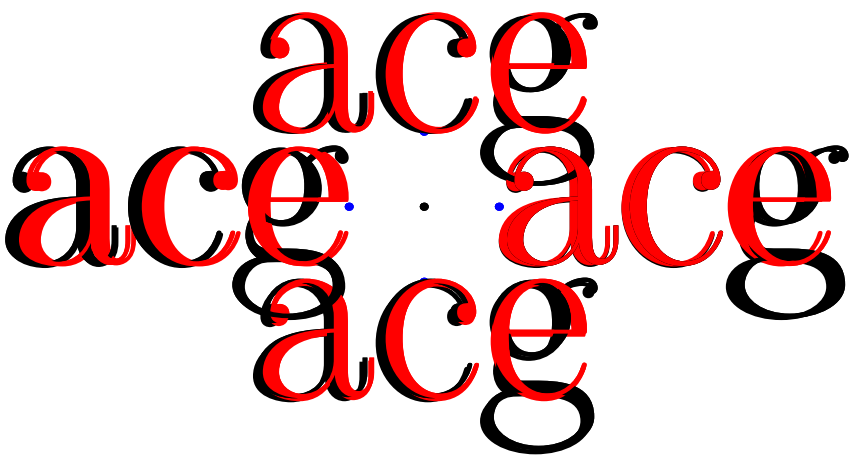
label("acg",O,align=N,basealign);
label("ace",O,align=N,red);
label("acg",O,align=S,basealign);
label("ace",O,align=S,red);
label("acg",O,align=E,basealign);
label("ace",O,align=E,red);
label("acg",O,align=W,basealign);
label("ace",O,align=W,red);

picture pic;
dot(pic,(labelmargin(),0,0),blue);
dot(pic,(-labelmargin(),0,0),blue);
dot(pic,(0,labelmargin(),0),blue);
dot(pic,(0,-labelmargin(),0),blue);
add(pic,O);

dot((0,0));

label("acg",(0,0),align=N,basealign);
label("ace",(0,0),align=N,red);
label("acg",(0,0),align=S,basealign);
label("ace",(0,0),align=S,red);
label("acg",(0,0),align=E,basealign);
label("ace",(0,0),align=E,red);
label("acg",(0,0),align=W,basealign);
label("ace",(0,0),align=W,red);

picture pic;
dot(pic,(labelmargin(),0),blue);
dot(pic,(-labelmargin(),0),blue);
dot(pic,(0,labelmargin()),blue);
dot(pic,(0,-labelmargin()),blue);
add(pic,(0,0));
```



Another method for aligning baselines is provided by the `baseline` function (see [baseline], page 21).

- 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 `type1cm` package available from

  `https://mirror.ctan.org/macros/latex/contrib/type1cm/`

  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, string shape)`. The default setting, `font("OT1","cmr","m","n")`, corresponds to 12pt Computer Modern Roman; this may be changed with `defaultpen(pen)`. The font setting of a pen is returned by `string font(pen p=currentpen)`.

  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");
```

- Starting with the 2018/04/01 release, LaTeX takes UTF-8 as the new default input encoding. However, you can still set different input encoding (so as the font, font encoding or even language context). Here is an example for `cp1251` and Russian language in Cyrillic script (font encoding `T2A`):

  ```
  texpreamble("\usepackage[math]{anttor}");
  texpreamble("\usepackage[T2A]{fontenc}");
  texpreamble("\usepackage[cp1251]{inputenc}");
  texpreamble("\usepackage[russian]{babel}");
  ```

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

  `https://ctan.org/pkg/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*}}");
  ```

- 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 Tables 136 and 137 of `https://opensource.adobe.com/dc-acrobat-sdk-docs/standards/pdfstandards/pdf/PDF32000_2008.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 magick` 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 global `PostScript` frame `currentpatterns`, with optional left-bottom margin `lb` and right-top margin `rt`.

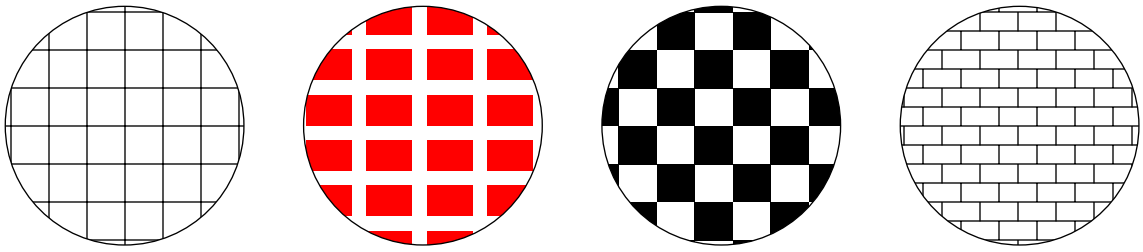
  ```
  import patterns;
  void add(string name, picture pic, pair lb=0, pair rt=0);
  ```

  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 module `patterns`:

  ```
  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"));
```

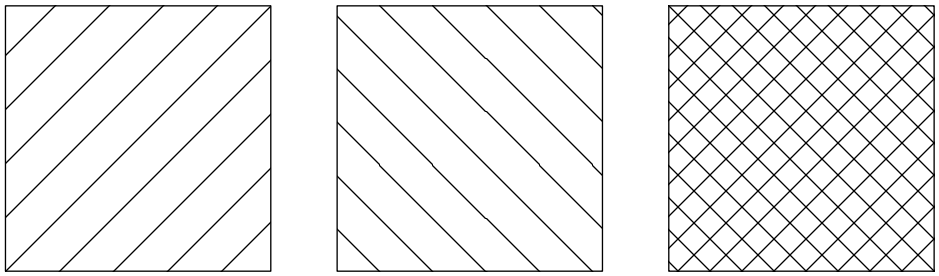


Hatch patterns can be generated with the routines `picture hatch(real H=5mm, pair dir=NE, pen p=currentpen)`, `picture crosshatch(real H=5mm, pen p=currentpen)`:

```
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 module `patterns`. 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;

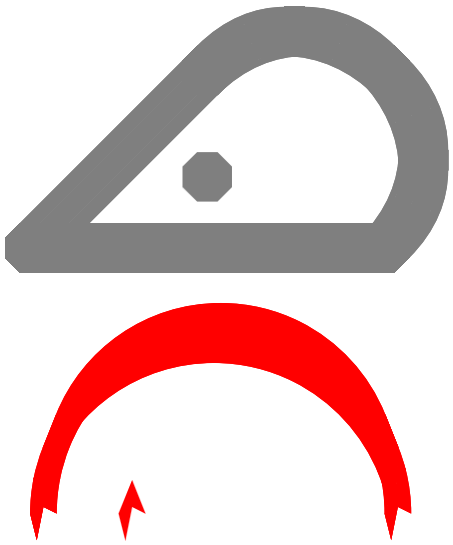
real d=4mm;
picture tiling;
path square=scale(d)*unitsquare;
axialshade(tiling,square,white,(0,0),black,(d,d));
fill(tiling,shift(d,d)*square,blue);
add("shadedtiling",tiling);

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 behavior (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.

6.4 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, strings, transforms, frames, and pictures by multiplication (via the binary operator `*`) on the left (see [circle], page 32, 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 scale(real x, real y);`
: scale by `x` in the x direction and by `y` in the y direction;

`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`.

`transform zeroTransform;`
: the zero transform;

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. The function `bool isometry(transform t)` can be used to test if `t` is an isometry (preserves distance).

6.5 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 The implicit initializer for frames is `newframe`. The function `bool empty(frame f)` returns `true` only if the frame `f` is empty. A frame may be erased with the `erase(frame)` routine. The functions `pair min(frame)` and `pair max(frame)` 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 19), is returned by

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

`picture` Pictures are high-level structures (see Section 6.9 [Structures], page 62) 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=pic.keepAspect);
```

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 default), 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, bool warn=false);
```

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 pic can be fit to a frame and output to a file prefix.format using image format format by calling the shipout function:

```
void shipout(string prefix=defaultfilename, picture pic=currentpicture,
             orientation orientation=orientation,
             string format="", bool wait=false, bool view=true,
             string options="", string script="",
             light light=currentlight, projection P=currentprojection)
```

The default output format, PostScript, may be changed with the -f or -tex command-line options. The options, script, and projection parameters are only relevant for 3D pictures. If defaultfilename is an empty string, the prefix outprefix() will be used.

A shipout() command is added implicitly at file exit. Explicit shipout() commands to the same file as the final implicit shipout are ignored. The default page orientation is Portrait; this may be modified by changing the variable orientation. To output in landscape mode, simply set the variable orientation=Landscape or issue the command

```
shipout(Landscape);
```

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

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 (such as 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.

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 the interior and draw the boundary.

FillDraw(real xmargin=0, real ymargin=xmargin, pen fillpen=nullpen, pen drawpen=nullpen) If fillpen is nullpen, fill with the drawing pen; otherwise fill with pen fillpen. If drawpen is nullpen, draw

the boundary with `fillpen`; otherwise with `drawpen`. An optional margin of `xmargin` and `ymargin` can be specified.

`Fill` Fill the interior.

`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.

`RadialShadeDraw(real xmargin=0, real ymargin=xmargin, pen penc, pen penr, pen drawpen=nullpen)` Fill with RadialShade and draw the boundary.

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 pen `p`, using the function `bbox(p,Fill)`.

To pad a picture to a precise size in both directions, fit the picture to a frame using the function

```
frame pad(picture pic=currentpicture, real xsize=pic.xsize,
          real ysize=pic.ysize, filltype filltype=NoFill);
```

The functions

```
pair min(picture pic, user=false);
pair max(picture pic, user=false);
pair size(picture pic, user=false);
```

calculate the bounds that picture `pic` would have if it were currently fit to a frame using its default size specification. If `user` is `false` the returned value is in `PostScript` coordinates, otherwise it is in user coordinates.

The function

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

is a convenient way of determining the point on the bounding box of `pic` in the direction `dir` relative to its center, ignoring the contributions from fixed-size

objects (such as labels and arrowheads). If `user` is `true` the returned value is in user coordinates, otherwise it is in `PostScript` coordinates.

The function

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

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. If `user` is `true` the returned value is in user coordinates, otherwise it is in `PostScript` coordinates.

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

The first example adds `src` to `currentpicture`; the second one adds `src` to `dest`. The `group` option specifies whether or not the graphical user interface should treat all of the elements of `src` as a single entity (see Chapter 13 [GUI], page 210), `filltype` requests optional background filling or clipping, and `above` specifies whether to add `src` above or below existing objects.

There are also routines to add a fixed-size picture or frame `src` to another picture `dest` (or `currentpicture`) about the user coordinate `position`:

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

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 19). 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 128, and [image], page 169. If you want to align three or more subpictures, group them two at a time:

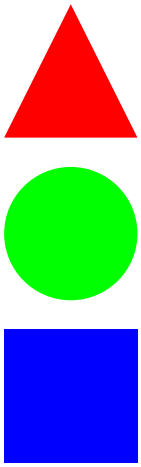
```
picture pic1;
real size=50;
size(pic1,size);
fill(pic1,(0,0)--(50,100)--(100,0)--cycle,red);

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

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

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

To save a snapshot of `currentpicture`, `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.

6.6 Deferred drawing

It is sometimes desirable to have elements of a fixed absolute size, independent of the picture scaling from user to PostScript coordinates. For example, normally one would want the size of a dot created with dot, the size of the arrowheads created with arrow (see [arrows], page 14), and labels to be drawn independent of the scaling.

However, because of Asymptote's automatic scaling feature (see Section 3.3 [Figure size], page 10), the translation between user coordinate and PostScript coordinate is not determined until shipout time:

```
size(1cm);
dot((0,0));
dot((1,1));
shipout("x"); // at this point, 1 unit coordinate = 1cm
dot((2,2));
shipout("y"); // at this point, 1 unit coordinate = 0.5cm
```

It is therefore necessary to defer the drawing of these elements until shipout time.

For example, a frame can be added at a specified user coordinate of a picture with the deferred drawing routine add(picture dest=currentpicture, frame src, pair position):

```
frame f;
```

```
fill(f,circle((0,0),1.5pt));
add(f,position=(1,1),align=(0,0));
```

A deferred drawing routine is an object of type `drawer`, which is a function with signature `void(frame f, transform t)`. For example, if the drawing routine

```
void d(frame f, transform t){
  fill(f,shift(3cm,0)*circle(t*(1,1),2pt));
}
```

is added to `currentpicture` with

```
currentpicture.add(d);
```

a filled circle of radius 2pt will be drawn on currentpicture centered 3cm to the right of user coordinate `(1,1)`. The parameter `t` is the affine transformation from user coordinates to `PostScript` coordinates.

Even though the actual drawing is deferred, you still need to specify to the `Asymptote` scaling routines that ultimately a fixed-size circle of radius 2pt will be drawn 3cm to the right of user-coordinate (1,1), by adding a frame about (1,1) that extends beyond (1,1) from `(3cm-2pt,-2pt)+min(currentpen)` to `(3cm+2pt,2pt)+max(currentpen)`, where we have even accounted for the pen linewidth. The following example will then produce a PDF file 10 cm wide:

```
settings.outformat="pdf";
size(10cm);
dot((0,0));
dot((1,1),red);
add(new void(frame f, transform t) {
  fill(f,shift(3cm,0)*circle(t*(1,1),2pt));
});
pair trueMin=(3cm-2pt,-2pt)+min(currentpen);
pair trueMax=(3cm+2pt,2pt)+max(currentpen);
currentpicture.addPoint((1,1),trueMin);
currentpicture.addPoint((1,1),trueMax);
```

Here we specified the minimum and maximum user and truesize (fixed) coordinates of the drawer with the `picture` routine

```
void addPoint(pair user, pair truesize);
```

Alternatively, one can use

```
void addBox(pair userMin, pair userMax, pair trueMin=0, pair trueMax=0) {
```

to specify a bounding box with bottom-left corner `t*(1,1)+trueMin` and top-right corner `t*(1,1)+trueMax`, where `t` is the transformation that transforms from user coordinates to `PostScript` coordinates.

For more details about deferred drawing, see "Asymptote: A vector graphics language," John C. Bowman and Andy Hammerlindl, TUGBOAT: The Communications of the TeX Users Group, 29:2, 288-294 (2008),

`https://www.math.ualberta.ca/~bowman/publications/asyTUG.pdf`.

6.7 Files

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

An input file can be opened with

```
input(string name=""
```

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 `comment` 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 `Asymptote` is compiled with support for `libcurl`, `name` can be a URL.

Unless the `-noglobalread` command-line option is specified, one can change the current working directory for read operations 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, string comment="#", string mode="");
```

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`, any 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 `-nosafe`) command-line option is specified. Reading from files in other directories is allowed unless the `-noglobalread` command-line option is specified. The function `string mktemp(string s)` may be used to create and return the name of a unique temporary file in the current directory based on the string `s`.

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

The last form accepts a heterogeneous list of writeable values, optionally prefixed by a `file` and/or an identifying `string`, and optionally suffixed by a `suffix` (a function of type `void(file)`). It is used when no type-specific `write` overload matches, for example

`write(42, 3.14, "done")`. The file, identifying string, and suffix may instead be given anywhere in the list as the named arguments `file`, `s`, and `suffix`, as in `write(x, y, suffix=none)`. The elements of a rest argument, as in `write(x ... a)`, are written after the other values. Values of any struct type that defines a publicly accessible `write(file, suffix)` method (such as `Label`), and arrays of such values, are written by calling that method. The optional identifying string (if present) is written flush against the first data value (no tab separator), so `write("a", "b", 1)` produces

```
ab      1
```

while `write(1, "a", "b")` produces

```
1       a       b
```

If `file` is not specified, `stdout` is used and the default `suffix` is `endl`. If specified, the optional identifying string `s` is written before the data. The `suffix` may be one of the following: `none` (do nothing), `flush` (output buffered data), `endl` (terminate with a newline and flush), `newl` (terminate with a newline), `DOSendl` (terminate with a DOS newline and flush), `DOSnewl` (terminate with a DOS newline), `tab` (terminate with a tab), or `comma` (terminate with a comma). Here are some simple examples of data output:

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

When multiple arrays of the same scalar type `T` are listed, they are written as indexed columns, one index per line:

```
write(new int[] {1,2}, new int[]{3,4});
// 0:      1       3
// 1:      2       4
```

Arrays of different types (or any mix of values) are written by the heterogeneous form, with each value on its own line:

```
write(new int[] {1,2}, new pair[]{1, I});
// 1
// 2
// (1,0)
// (0,1)
```

A file may be opened with `mode="xdr"` to read or write double precision (64-bit) reals and single precision (32-bit) integers in Sun Microsystem's XDR (External Data Representation) portable binary format. A file may instead be opened with `mode="binary"` to read or write double precision reals and single precision integers in the native (nonportable) machine binary format. The virtual member functions `file singlereal(bool b=true)` and `file singleint(bool b=true)` may be used to change the precision of real and integer I/O operations, respectively, for an XDR or binary file. Similarly, the function `file signedint(bool b=true)` can be used to modify the signedness of integer reads and writes for an XDR or binary file. Since there are no end-of-line terminators in these formats, reading a string from an XDR or binary file will by default read in the remainder of the file as a string. The virtual member function `file word(bool b=true)` may be used to change this behaviour: a double precision integer specifying the length of the string is first read or written, followed by that many characters.

The virtual fields `name`, `mode`, `singlereal`, `singleint`, and `signedint` may be used to query these respective states of a file.

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)`. The function `int precision(file file=stdout, int digits=0)` sets the number of digits of output precision for `file` to `digits`, provided `digits` is nonzero, and returns the previous precision setting. The function `int tell(file)` returns the current position in a file relative to the beginning. The routine `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 to the next `n` output lines, `s` followed by `Enter` to scroll without further interruption, or `q` followed by `Enter` to quit the current output operation. If `n` is negative, the output scrolls a page at a time (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);
triple gettriple(string name="", triple default=(0,0,0), 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 208). 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`. The routine `history(int n=0)` returns the interactive history. For example,

```
write(output("transcript.asy"),history());
```

outputs the interactive history to the file `transcript.asy`.

The function `int delete(string s)` deletes the file named by the string `s`. Unless the `-globalwrite` (or `-nosafe`) option is enabled, the file must reside in the current directory. The function `int rename(string from, string to)` may be used to rename file `from` to file `to`. Unless the `-globalwrite` (or `-nosafe`) option is enabled, this operation is restricted to the current directory. The functions

```
int convert(string args="", string file="", string format="");
int animate(string args="", string file="", string format="");
```

call the `ImageMagick` commands `magick` and `animate`, respectively, with the arguments `args` and the file name constructed from the strings `file` and `format`.

6.8 Variable initializers

A variable can be assigned a value when it is declared `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.

The special type `var` may be used to infer the type of a variable from its initializer. If the initializer is an expression of a unique type, then the variable will be defined with that type. For instance,

```
var x=5;
var y=4.3;
var reddash=red+dashed;
```

is equivalent to

```
int x=5;
real y=4.3;
pen reddash=red+dashed;
```

`var` may also be used with the extended `for` loop syntax.

```
int[] a = {1,2,3};
for (var x : a)
  write(x);
```

6.9 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;
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;
T foo;
```

Here, `new T` produces a new instance of the struct, which causes `Tcount` to be incremented, tracking the number of instances produced. The declarations `T foo=new T` and `T foo` are equivalent: the second form implicitly creates a new instance of `T`. That is, 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 behavior is to avoid infinite recursion of creating new instances in code such as

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

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 expressions of the same nullable type refer to the same instance of that type. In the example at the beginning of this 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`. If `a` has a well-defined type, then `alias(a,null)` is an idiomatic way to check whether `a` is `null`; however, `alias(null, null)` is an error since neither operand has a well-defined type.

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). Because the default `==` and `!=` (like `alias`) require both operands to have the same type, no implicit casting is performed to reconcile differing operand types: comparing values of two different structure types is an error, even if a cast between them exists, unless a matching `operator ==` has been declared.

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;
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 `operator init`:

```
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 6.8 [Variable initializers], page 61). 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.

Internally, `operator init` implicitly defines a constructor function `Person(string,string)` as follows, where *args* is `string firstname, string lastname` in this case:

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

  static Person Person(args) {
    Person p=new Person;
    p.operator init(args);
    return p;
```

```
  }
}
```

which then can be used as:

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

The following is also implicitly generated in the enclosing scope, after the end of the structure definition.

```
from Person unravel Person;
```

It allows us to use the constructor without qualification, otherwise we would have to refer to the constructor by the qualified name `Person.Person`.

Much like in C++, casting (see Section 6.14 [Casts], page 85) provides for an elegant implementation of structure inheritance, including a virtual function `v`:

```
struct parent {
  real x;
  void operator init(int x) {this.x=x;}
  void v(int) {write(0);}
  void f() {v(1);}
}

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

struct child {
  parent parent;
  real y=3;
  void operator init(int x) {parent.operator init(x);}
  void v(int x) {write(x);}
  parent.v=v;
  void f()=parent.f;
}

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

parent p=parent(1);
child c=child(2);

write(c);                       // Outputs 2;

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

write(c.parent.x);              // Outputs 2;
write(c.y);                     // Outputs 3;
```

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

6.10 Operators

6.10.1 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 typically what is intended; otherwise one can use the function `int quotient(int x, int y)`, which returns 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

`#` integer division; equivalent to `quotient(x,y)`. Noting that the Python3 community adopted our comment symbol (`//`) for integer division, we decided to reciprocate and use their comment symbol for integer division in Asymptote!

`%` modulo; the result always has the same sign as the divisor. In particular, this makes `q*(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 `(1-t)*a+t*b` for nonintegral built-in arithmetic types `T`. If `a` and `b` are pens, they are first promoted to the same color space.

Asymptote also defines bitwise functions `int AND(int,int)`, `int OR(int,int)`, `int XOR(int,int)`, `int NOT(int)`, `int CLZ(int)` (count leading zeros), `int CTZ(int)` (count trailing zeros), `int popcount(int)` (count bits populated by ones), and `int bitreverse(int a, int bits)` (reverse bits within a word of length bits).

6.10.2 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.

6.10.3 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);
```

See Section 6.18 [Autounravel], page 93, for an example of overloading the `+` operator.

6.10.4 Bracket operators

If `a` is not an array, the notation `a[i]` is equivalent to `a.operator[](i)`. Thus, the `[]` operator can be defined to provide this notation for a user-defined structure. Similarly, the notation `a[i]=b` is equivalent to `a.operator[=](i,b)` (except that the former always returns `b`).

The builtin type `string` has a virtual member function `string operator[](int pos)`. No primitive type has a virtual member function `operator [=]`, since primitive types are immutable.

There are some restrictions on how these operators can be defined:

- Neither operator can be overloaded within a single structure (although different structures can have different definitions for these operators).
- `operator []` must take a single argument and have a non-void return type.
- `operator [=]` can only be defined if `operator[]` is also defined.
- `operator[=]` must take two arguments and have a void return type. Furthermore, if `K` and `V` are the types of the first and second arguments to `operator[=]`, respectively, then `K` must be the type of the argument to `operator[]` and `V` must be the return type of `operator[]`.

Here is a simple example. (See Section 6.20.3 [Maps], page 99, for data structures similar to `slowMap` but more flexible, more powerful, and faster.)

```
struct slowMap {
  string[] keys;
  real[] values;
  real operator [](string key) {
    for(int i=0; i < keys.length; ++i) {
      if(keys[i] == key) return values[i];
    }
    return -inf;
  }
  void operator [=](string key, real value) {
    assert(all(keys != key), 'duplicate key');
    keys.push(key);
    values.push(value);
  }
}

slowMap m;
m['pi']=3.14159;
m['e']=2.71828;
assert(m['pi']==3.14159);
assert(m['e']==2.71828);
assert(m['sqrt(2)'] == -inf);
write("m['pi']=", m['pi']);
```

6.10.5 Cast operators

See Section 6.14 [Casts], page 85, for information on `operator cast` and `operator ecast`.

6.11 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
```

6.12 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.

Function arguments are passed by value. To pass an argument by reference, simply enclose it in a structure (see Section 6.9 [Structures], page 62).

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. "Higher-order" functions, in other words functions that return functions, are allowed. The return type must be given a signature-free alias with `using` or `typedef`:

```
using intop = int(int);
// typedef int intop(int); // Equivalent to previous line
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 or illegal instruction, unless a fully working version of the GNU library `libsigsegv` is installed at configuration time).

6.12.1 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 6.14 [Casts], page 85) 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.

6.12.2 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.

Parameters can be specified as "keyword-only" by putting `keyword` immediately before the parameter name, as in `int f(int keyword x)` or `int f(int keyword x=77)`. This forces the caller of the function to use a named argument to give a value for this parameter. That is, `f(x=42)` is legal, but `f(25)` is not. Keyword-only parameters must be listed after normal parameters in a function definition.

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.

6.12.3 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
```

The comma before an ellipsis is optional, so that the following two declarations are equivalent:

```
int subtract(int start, ... int[] subs);
int subtract(int start ... int[] subs);
```

Note, however, that a comma cannot directly follow an open parenthesis: `int sum(, ... int[] nums)` is a syntax error.

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.

In some cases, keyword-only parameters are helpful to avoid arguments intended for the rest parameter to be assigned to other parameters. For example, here the use of `keyword` is to avoid `pnorm(1.0,2.0,0.3)` matching `1.0` to `p`.

```
real pnorm(real keyword p=2.0, ... real[] v)
{
  return sum(v^p)^(1/p);
}
```

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
```

6.12.4 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`, `expm1`, `log1p`, as well as the identity function `identity`. `Asymptote` also defines the order `n` Bessel functions of the first kind `Jn(int n, real)` and second kind `Yn(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 that use degrees rather than radians. We also define complex versions of the `sqrt`, `sin`, `cos`, `exp`, `log`, and `gamma` functions.

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 `intMin` 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)`), an `abs(pair)` function (equivalent to `length(pair)`).

Asymptote's random number generators can be seeded with `srand(int)`. If a negative argument is given to `srand`, an internal entropy source is used as a seed. Random numbers are initially seeded with `srand(-1)`. The function `int rand(int a=0, int b=randMax)` returns a random integer in [a,b]. The `unitrand()` function returns a random number uniformly distributed in the interval [0,1). The `randString(int len, string chars)` function returns a random string of length `len` composed of characters from the string `chars`. If `chars` is the empty string, the string `'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'` is used. In each position of the output, the probability of a character appearing is proportional to the number of times it appears in `chars`.

A Gaussian random number generator `Gaussrand` and a collection of statistics routines, including `histogram`, are provided in the module `stats`.

The functions `factorial(int n)`, which returns $n!$, and `choose(int n, int k)`, which returns $n!/(k!(n-k)!)$, are also defined.

When configured with the GNU Scientific Library (GSL), available from `https://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 Jacobi elliptic functions `real[] sncndn(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));
```

`Asymptote` also provides a few general purpose numerical routines:

`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.

`real simpson(real f(real), real a, real b, real acc=realEpsilon, real dxmax=b-a)`

returns the integral of `f` from `a` to `b` using adaptive Simpson integration.

`cputime cputime()`

returns a structure `cputime` with cumulative CPU times broken down into the fields `parent.user`, `parent.system`, `child.user`, and `child.system`, along with the cumulative wall clock time in `parent.clock`, all measured in seconds. For convenience, the incremental fields `change.user`, `change.system`, and `change.clock` indicate the change in the corresponding fields 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".

6.13 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`. A convenient shorthand exists for iterating over all elements of an array; see [array iteration], page 25.

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};
```

An explicit array initialization can also end with a copy of another array:

```
real[] A={0,1};
real[] B={...A};  // Now B contains {0,1}.
real[] C={4 ...A};  // Now C contains {4,0,1}.
```

(Side note: without the space, `4...A` would be parsed as `4.0 .. A` and thus give a syntax error.)

Array assignment in `Asymptote` does a shallow copy: only the pointer is copied (if one copy if modified, the other will be too).

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

```
real[] B = A;
B[0] = 3;
write(A[0]); // Outputs 3.
```

The `copy` function listed below provides a deep copy of an array.

Every array `A` of type `T[]` has the virtual members

- `int length`,
- `bool cyclic`,
- `int[] keys`,
- `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.keys` evaluates to an array of integers containing the indices of initialized entries in the array in ascending order. Hence, for an array of length `n` with all entries initialized, `A.keys` evaluates to `{0,1,...,n-1}`. A new keys array is produced each time `A.keys` is evaluated.

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, `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 `array`, `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);

`T[] array(int n, T value, int depth=intMax)`
: returns an array consisting of `n` copies of `value`. If `value` is itself an array, a deep copy of `value` is made for each entry. If `depth` is specified, this deep copying only recurses to the specified number of levels;

`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); see also Section 6.16.1 [range], page 91;

`int[] sequence(int n, int m, int skip)`
: if `(m-n)/skip >= 0` returns an array `{n,n+skip,...,m}` skipping by `skip` (otherwise returns a null array); see also Section 6.16.1 [range], page 91;

`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);

`T[] map(T f(T), T[] a)`
: returns the array obtained by applying the function `f` to each element of the array `a`. This is equivalent to `sequence(new T(int i) {return f(a[i]);},a.length)`;

`T2[] map(T2 f(T1), T1[] a)`
: constructed by running `from mapArray(Src=T1, Dst=T2) access map;`, returns the array obtained by applying the function `f` to each element of the array `a`;

`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 `{0,1,2,...,n-1}`, so that `b[complement(a,b.length)]` yields the complement of `b[a]`;

`real[] uniform(real a, real b, int n)`
: if `n >= 1` returns a uniform partition of `[a,b]` into `n` subintervals (otherwise returns a null array);

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

`int[] findall(bool[] a)`
: returns the indices of all `true` values in the boolean array `a`;

`int search(T[] a, T key)`
: For built-in ordered types `T`, searches a sorted array `a` of `n` elements for `key`, returning the index `i` if `a[i] <= key < a[i+1]`, `-1` if `key` is less than all elements of `a`, or `n-1` if `key` is greater than or equal to the last element of `a`;

`int search(T[] a, T key, bool less(T i, T j))`
: searches an array `a` for `key` sorted in ascending order such that element `i` precedes element `j` if `less(i,j)` is true;

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

`T[] concat(...T[][] a)`
: returns a new array formed by concatenating the given one-dimensional arrays given as arguments;

`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(sort(a));
```

produces

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

`T[] sort(T[] a, bool less(T i, T j), bool stable=true)`
: returns a copy of `a` sorted in ascending order such that element `i` precedes element `j` if `less(i,j)` is true, subject to (if `stable` is `true`) the stability

constraint that the original order of elements `i` and `j` is preserved if `less(i,j)` and `less(j,i)` are both `false`;

`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`. In the case where `T` is `bool`, the number of true elements in `a` is returned;

`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`;

`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[] pairs(real[] x, real[] y);`
for arrays `x` and `y` of the same length, returns the pair array `sequence(new pair(int i) {return (x[i],y[i]);},x.length)`;

`pair[] fft(pair[] a, int sign=1)`
returns the unnormalized 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);
```

`pair[][] fft(pair[][] a, int sign=1)`
returns the unnormalized two-dimensional Fourier transform of `a` using the given `sign`;

`pair[][][] fft(pair[][][] a, int sign=1)`
: returns the unnormalized three-dimensional Fourier transform of `a` using the given `sign`;

`realschur schur(real[][] a)`
: returns a structure `realschur` containing a unitary matrix `U` and a quasitriangular matrix `T` such that `a=U*T*transpose(U)`;

`schur schur(pair[][] a)`
: returns a structure `schur` containing a unitary matrix `U` and a triangular matrix `T` such that `a=U*T*conj(transpose(U))`;

`real dot(real[] a, real[] b)`
: returns the dot product of the vectors `a` and `b`;

`pair dot(pair[] a, pair[] b)`
: returns the complex dot product `sum(a*conj(b))` of the vectors `a` and `b`;

`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, bool warn=true)`
: 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 `a` is a singular matrix and `warn` is `false`, return an empty array. If the matrix `a` is tridiagonal, the routine `tridiagonal` provides a more efficient algorithm (see [tridiagonal], page 81);

`real[][] solve(real[][] a, real[][] b, bool warn=true)`
: 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. If `a` is a singular matrix and `warn` is `false`, return an empty matrix;

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

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

`real[][] inverse(real[][] a)`
: returns the inverse of a square matrix `a`;

`real[] quadraticroots(real a, real b, real c);`
: This numerically robust solver returns the real roots of the quadratic equation $ax^2 + bx + c = 0$, in ascending order. Multiple roots are listed separately;

`pair[] quadraticroots(explicit pair a, explicit pair b, explicit pair c);`
: This numerically robust solver returns the complex 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 $ax^3 + bx^2 + cx + d = 0$. Multiple roots are listed separately.

`pair[] complexcubicroots(real a, real b, real c, real d);`
: This numerically robust solver returns the complex roots of the cubic equation $ax^3 + bx^2 + cx + d = 0$. Multiple roots are listed separately.

`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).

The virtual member functions `line()`, `word()`, `csv()`, `dimension()`, and `read()` of a file are useful for qualifying array reads, while the virtual fields `line`, `word`, `csv`, and `dimension` may be used to query these respective states.

If line mode is set with `file line(bool b=true)`, then reading will stop once the end of the line is reached:

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

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, `file word(bool b=true)`, which causes string reads to respect white-space delimiters, instead of the default end-of-line delimiter:

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

Another useful mode is comma-separated-value mode, `file csv(bool b=true)`, which causes reads to respect comma delimiters:

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

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

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

Sometimes the array dimensions are stored with the data as integer fields at the beginning of an array. Such 1, 2, or 3 dimensional arrays can be read in with the virtual member functions `read(1)`, `read(2)`, or `read(3)`, respectively:

```
file fin=input("test.txt");
real[] A=fin.read(1);
real[][] B=fin.read(2);
real[][][] C=fin.read(3);
```

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.

6.13.1 Slices

Asymptote allows a section of an array to be addressed as a slice using a Python-like syntax. If `A` is an array, the expression `A[m:n]` returns a new array consisting of the elements of `A` with indices from `m` up to but not including `n`. For example,

```
int[] x={0,1,2,3,4,5,6,7,8,9};
int[] y=x[2:6];  // y={2,3,4,5};
int[] z=x[5:10]; // z={5,6,7,8,9};
```

If the left index is omitted, it is taken be `0`. If the right index is omitted it is taken to be the length of the array. If both are omitted, the slice then goes from the start of the array to the end, producing a non-cyclic deep copy of the array. For example:

```
int[] x={0,1,2,3,4,5,6,7,8,9};
int[] y=x[:4];  // y={0,1,2,3}
int[] z=x[5:];  // z={5,6,7,8,9}
```

```
int[] w=x[:];   // w={0,1,2,3,4,5,6,7,8,9}, distinct from array x.
```

If A is a non-cyclic array, it is illegal to use negative values for either of the indices. If the indices exceed the length of the array, however, they are politely truncated to that length.

For cyclic arrays, the slice `A[m:n]` still consists of the cells with indices in the set [`m`,`n`), but now negative values and values beyond the length of the array are allowed. The indices simply wrap around. For example:

```
int[] x={0,1,2,3,4,5,6,7,8,9};
x.cyclic=true;
int[] y=x[8:15];  // y={8,9,0,1,2,3,4}.
int[] z=x[-5:5];  // z={5,6,7,8,9,0,1,2,3,4}
int[] w=x[-3:17]; // w={7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6}
```

Notice that with cyclic arrays, it is possible to include the same element of the original array multiple times within a slice. Regardless of the original array, arrays produced by slices are always non-cyclic.

If the left and right indices of a slice are the same, the result is an empty array. If the array being sliced is empty, the result is an empty array. Any slice with a left index greater than its right index will yield an error.

Slices can also be assigned to, changing the value of the original array. If the array being assigned to the slice has a different length than the slice itself, elements will be inserted or removed from the array to accommodate it. For instance:

```
string[] toppings={"mayo", "salt", "ham", "lettuce"};
toppings[0:2]=new string[] {"mustard", "pepper"};
    // Now toppings={"mustard", "pepper", "ham", "lettuce"}
toppings[2:3]=new string[] {"turkey", "bacon" };
    // Now toppings={"mustard", "pepper", "turkey", "bacon", "lettuce"}
toppings[0:3]=new string[] {"tomato"};
    // Now toppings={"tomato", "bacon", "lettuce"}
```

If an array is assigned to a slice of itself, a copy of the original array is assigned to the slice. That is, code such as `x[m:n]=x` is equivalent to `x[m:n]=copy(x)`. One can use the shorthand `x[m:m]=y` to insert the contents of the array `y` into the array `x` starting at the location just before `x[m]`.

For a cyclic array, a slice is bridging if it addresses cells up to the end of the array and then continues on to address cells at the start of the array. For instance, if `A` is a cyclic array of length 10, `A[8:12]`, `A[-3:1]`, and `A[5:25]` are bridging slices whereas `A[3:7]`, `A[7:10]`, `A[-3:0]` and `A[103:107]` are not. Bridging slices can only be assigned to if the number of elements in the slice is exactly equal to the number of elements we are assigning to it. Otherwise, there is no clear way to decide which of the new entries should be `A[0]` and an error is reported. Non-bridging slices may be assigned an array of any length.

For a cyclic array `A` an expression of the form `A[A.length:A.length]` is equivalent to the expression `A[0:0]` and so assigning to this slice will insert values at the start of the array. `A.append()` can be used to insert values at the end of the array.

It is illegal to assign to a slice of a cyclic array that repeats any of the cells.

6.14 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 module three. 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
```

In situations where casting from a string to a type T fails, an uninitialized variable is returned; this condition can be detected with the function bool initialized(T);

```
int i=(int) "2.5";
assert(initialized(i),"Invalid cast.");

real x=(real) "2.5a";
assert(initialized(x),"Invalid cast.");
```

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`.

6.15 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 6) 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 is contained in a subdirectory, the directory structure can be specified with a dot-separated name. For instance, the command

```
import directoryName.moduleName;
access directoryName.moduleName;
```

will load the file `directoryName/moduleName.asy` as module `moduleName`. Likewise,

```
from directoryName.moduleName access functionName;
```

will access `functionName` from `directoryName/moduleName.asy`. The directory `directoryName` is relative to the current directory, the `base` directory, or any directory in the search path (see Section 2.5 [Search paths], page 6). Nested directory specifications are allowed, as long as the directory and module names are also valid as variable names. Otherwise (for example if the directory is an absolute path), the filename must be enclosed with quotation marks:

```
import '/usr/local/share/asymptote/graph.asy' as graph;
```

If `Asymptote` is compiled with support for `libcurl`, the file name can even be a URL:

```
import 'https://raw.githubusercontent.com/vectorgraphics/asymptote/HEAD/doc/axis3.asy'
as axis3;
```

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 205).

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 an existing 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`.

6.15.1 Templated imports

In Asymptote types are specified when they are imported. The first executable line of any such module must be of the form `typedef import(<types>)`, where `<types>` is a list of required type parameters. For instance,

```
typedef import(T, S, Number);
```

could be the first line of a module that requires three type parameters. The remaining code in the module can then use `T`, `S`, and `Number` as types.

To import such a module, one must specify the types to be used. For instance, if the module above were named `templatedModule`, it could be accessed for types `string`, `int[]`, and `real` with the import command

```
access templatedModule(T=string, S=int[], Number=real)
    as templatedModule_string_int_real;
```

Note that this is actually an *access* command rather than an *import* command, so a type, function, or variable `A` defined in `templatedModule.asy` would need to be accessed qualified as `templatedModule_string_int_real.A`.

Alternatively, the module could be imported via a command like

```
from templatedModule(T=string, S=int[], Number=real) access
    Wrapper_Number as Wrapper_real,
    operator ==;
```

This command would automatically rename `Wrapper_Number` to `Wrapper_real` and would also allow the use of any `operator ==` overloads defined in the module.

Further examples can be found in the `tests/template` subdirectory of the `Asymptote` source directory.

Issues: Certain expected operators (such as `operator ==`) may only be available for type arguments that are builtin or defined in module `plain`.

6.16 Iterators

If a structure `a` has a member function `operator iter()`, the code

```
for(var i : a) {
  <statements>
}
```

is syntactic sugar for

```
for(var it=a.operator iter(); it.valid(); it.advance()) {
  var i=it.get();
  <statements>
}
```

Thus, we can make a structure iterable by defining an `operator iter` method that returns an object with the following methods:

`T get()`
: returns the value at the iterator's current position (without changing the position). Note that `T` can be any type, builtin or user-defined;

`void advance()`
: advances the iterator to the next position;

`bool valid()`
: returns `true` if the iterator is at a valid position or `false` if the iterator has advanced past the last item.

Although the return type of `operator iter` can theoretically be any type that has these three methods, it is strongly recommended to use the `Iter_T` structure defined in the templated module `collections.iter(T)` so that other utilities (see Section 6.20.2 [Iterators

and utilities], page 96) can be used with the iterator. The three methods of this structure can be set as fields.

As an example, here is an iterator for the even-numbered elements of an array of strings:

```
from collections.iter(T=string) access
    Iter_T as Iter_string,
    Iterable_T as Iterable_string;
struct EvenStrings {
  string[] a;
  void operator init(string[] a) {
    this.a=a;
  }
  Iter_string operator iter() {
    int i = 0;
    Iter_string result;
    result.get = new string() { return a[i]; };
    result.advance = new void() { i += 2; };
    result.valid = new bool() { return i < a.length; };
    return result;
  }
  autounravel Iterable_string operator cast(EvenStrings es) {
    Iterable_string result;
    result.operator iter = es.operator iter;
    return result;
  }
}
string[] a = {'a', 'b', 'c', 'd', 'e', 'f'};
for (string s : EvenStrings(a)) {
  write(s);
}
```

Running this code prints out the following:

```
a
c
e
```

The autounraveled (see Section 6.18 [Autounravel], page 93) implicit cast to `Iterable_string` is not strictly necessary, but it allows the use of iterable utilities; see Section 6.20.2 [Iterators and utilities], page 96. This enables code like the following, which uses the `enumerate` utility to attach counters to the iterated strings:

```
from collections.enumerate(T=string) access enumerate;
for (var kv : enumerate(EvenStrings(a))) {
  write(string(kv.k) + ": " + kv.v);
}
```

Running this code prints out the following:

```
0: a
1: c
2: e
```

6.16.1 range

The following functions produce iterators over integers:

`Iterable_int range(int n)`
: returns an iterable that produces the integers `0,...,n-1`;

`Iterable_int range(int n, int m, int skip=1)`
: returns an iterable that produces the integers `n, n+skip, ..., m` if `(m-n)/skip >= 0`, and otherwise produces no integers.

Generally speaking, `for (int i : range(n))` is equivalent to `for (int i : sequence(n))`. The latter uses $O(n)$ memory but is usually faster. The former uses $O(1)$ memory but is slower. If an iterator is specifically required, however, `range(n)` is faster than `range(sequence(n))` (as well as being more memory-efficient).

6.17 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 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.

6.18 Autounravel

The `autounravel` modifier can be used to automatically unravel a field. This is useful when building an associated library of functions that operate on a structure. For instance, consider a simple implementation of the `rational` structure defined in `rational.asy`:

```
struct rational {
  int p=0, q=1;
  void operator init(int p, int q) {
    this.p=p;
    this.q=q;
  }
}
rational operator +(rational a, rational b) {
  return rational(a.p*b.q+b.p*a.q, a.q*b.q);
}
```

To allow `rational` to be used as a type parameter for a templated import that adds instances of `rational`, we should move `operator +` into the body of the `rational` structure and add the `autounravel` modifier:

```
struct rational {
  int p=0, q=1;
  void operator init(int p, int q) {
    this.p=p;
    this.q=q;
  }
  autounravel rational operator +(rational a, rational b) {
    return rational(a.p*b.q+b.p*a.q, a.q*b.q);
  }
}
```

This is almost equivalent to the previous code, but now the `+` operator will be accessible wherever the `rational` structure is.

Currently, types cannot be autounraveled. Users who encounter a case where this might be useful can submit a feature request at `https://github.com/vectorgraphics/asymptote/issues`.

6.18.1 Where fields are autounraveled

If a `struct` contains fields (including functions) that are declared with `autounravel`, these fields will be unraveled from the `struct` at:

- the end of the `struct` definition;

- a `typedef import` statement, if the `struct` is the argument for one of the type parameters;
- an `unravel` or `access` statement that unravels the `struct` from a module or outer `struct` (for instance, `from rational access rational;` would make the `+` operator available from the `struct rational` defined in `rational.asy`);
- A `typedef` statement like `typedef rational.rational rat;` that renames a `struct`.

6.18.2 Where `autounravel` is legal

The `autounravel` modifier implies `static` and can be used in many of the same places as `static`. However, specifying `autounravel` at the top level of a module (outside of any structure or function) is an error, whereas `static` gives only a warning.[1] In front of a `struct` definition or `typedef` statement, `autounravel` is forbidden because types cannot be autounraveled. While `static static` results in an error, `static autounravel` and `autounravel static` are both legal and have exactly the same effect as `autounravel` alone.

6.19 Hash functions

For an object to be used as a key in a hashmap (see Section 6.20.3 [Maps], page 99) or a member of a hashset (see Section 6.20.5 [Sets], page 103), it must have a method `int hash()` that returns a nonnegative integer. For correctness, this method must return the same value for two objects of the same type that are equal according to the `==` operator. For efficiency, it should behave like a pseudorandom function, so that the hash values of distinct objects are unrelated. Also note that it is not necessary for hash functions to be consistent across different runs of the program; `a.hash() == a.hash()` should always be true, but `write(a.hash())` may produce a different output each time the program is run.

In `Asymptote`, the `hash` method is defined for three of the builtin types: `int`, `real`, and `string`. Note that things like `3.hash()` and `'hello'.hash()` are *not* valid syntax; the correct version of these would be `(3).hash()` and `('hello').hash()`, with parentheses around the literal. If you want to hash a `real`, there are two important things to note: first, even if two `real`s are extremely close, their hash values will almost certainly not be; second, the same number will almost certainly have different hash values when hashed as an `int` versus a `real`.

6.19.1 Hashing user-defined structures

Additionally, there is a function `hash(int[])` that can be helpful when constructing hash methods for user-defined structures: convert all the fields to integers (possibly by hashing them), put these integers into an array, and hash the array.

Let's return to the example of the `Person` structure from Section 6.9 [Structures], page 62, adding an `age` field to make it more interesting. For this structure, there are two reasonable ways we might define a hash function for it. The first is to create a hash function based on all the fields of the structure:

```
struct Person {
```

[1] If top-level `autounravel` were allowed, a user might incorrectly assume that the field would be unraveled whenever the module is `access`ed. The `static` modifier is allowed at the top level because, while it does nothing, it does not mislead the user.

```
  string firstname;
  string lastname;
  int age;
  int hash() {
    return hash(new int[] {firstname.hash(), lastname.hash(), age});
  }
  autounravel bool operator ==(Person a, Person b) {
    if (alias(a, null)) return alias(b, null);
    if (alias(b, null)) return false;
    return a.firstname == b.firstname &&
           a.lastname == b.lastname &&
           a.age == b.age;
  }
  autounravel bool operator !=(Person a, Person b) {
    return !(a == b);
  }
}
```

This approach is necessary if we are redefining the `==` operator. Two `Person`s are equal if they have the same first name, last name, and age.

In contrast, the default `==` operator for structs considers two objects equal if they are the same object, meaning that changes to one will be reflected in the other. A hash method that plays well with this operator must remain the same even if the fields change. In this case, we can add a never-changing `id` field that uniquely identifies a `Person` object, and use that for hashing:

```
struct Person {
  string firstname;
  string lastname;
  int age;
  private static int lastID=-1;
  restricted int id = ++lastID;
  int hash() {
    return id.hash();
  }
}
```

The builtin hash functions are designed to remove undesirable regularity (with respect to modular arithmetic) when applied exactly once. This explains why the second example returns `id.hash()` instead of `id`, while `age` is not independently hashed in the first example. It may sometimes be most practical to feed the hash function's output back into it, as in `firstname` and `lastname` above, but this does not increase the hash quality; in theory, if we could serialize a `Person` object to a string and then hash the string, the hash quality would be even higher.

6.20 Collections (containers)

The `collections` subdirectory provides a number of useful containers and utilities via templated modules.

See Section 6.20.3 [Maps], page 99, for information on hashmaps (sometimes called dictionaries or associative arrays), which are arguably the single most useful container in the `collections` module.

6.20.1 Pairs

`Asymptote` has a builtin type for pairs of `reals`. To obtain a pair type for *any* two types, one can use the `collections.genericpair(K,V)` templated module. The fields are called `k` and `v` rather than `x` and `y`. The pair type is called `Pair_K_V`, which users are encouraged to rename based on the types being used. There is also a function `makePair` that creates a pair from two values; because of function overloading, the name of the function does not need to specify the two types.

Here's an example:

```
from collections.genericpair(K=int, V=string) access
    Pair_K_V as Pair_int_string,
    makePair;
var p=makePair(3, 'hello');
write(p.k);  // Outputs 3
write(p.v);  // Outputs hello
var q = makePair(4, 'hello');
var s = makePair(3, 'world');
Pair_int_string t = makePair(3, 'hello');
assert(p != q);
assert(p != s);
assert(p == t);
```

Note that the `==` and `!=` operators are overloaded for pairs, and the overloads are based on the `==` operators of the two types in the pair. An unfortunate consequence of this is that this module cannot be used for pairs of arrays, because the `==` operator for arrays returns a `bool[]` rather than a `bool`. See Section 6.20.8 [Wrapping arrays], page 110, for a workaround.

The `Pair_K_V` type also has an `int hash()` method initialized to `null`. If you want to use pairs as keys in a hashmap, you should define this method, something like this:

```
from collections.genericpair(K=int, V=string) access
    Pair_K_V as Pair_int_string;
Pair_int_string p = makePair(3, 'hello');
p.hash = new int() {
  return hash(new int[] {p.k, p.v.hash()});
};
```

See Section 6.19 [Hash functions], page 94, for more information.

6.20.2 Iterators and utilities

6.20.2.1 `collections.iter(T)`

The `collections.iter(T)` templated module provides structs `Iter_T` and `Iterable_T` that can be convenient for defining iterators. Additionally, there are several utilities for manipulating iterators of these types. Technically, the syntax `for(T x : a)` does not require anything from this module, but several utilities only work for iterables of type `Iterable_T`.

The `Iter_T` structure has the following fields, all of which are null by default:

- `T get()`
- `void advance()`
- `bool valid()`

`Iter_T` is intended as the return type of an `operator iter` method; See Section 6.16 [Iterators], page 89, for more information on how to use these fields.

`Iterable_T` is a generic structure for something that has an `operator iter` method returning an `Iter_T`. This structure, not `Iter_T`, is the type that can appear inside a range-based `for` loop. Consequently, most of the utilities below manipulate instances of `Iterable_T` rather than `Iter_T`.

It is good practice for a structure that defines an `operator iter` method to also define an implicit cast to `Iterable_T` for better compatibility with the utilities in this module. Additionally, an instance `a` of `Iterable_T` can be cast to a `T[]` with the syntax `(T[]) a`. (This is an explicit cast, so it requires parentheses.) Going the other way, a `T[]` can be implicitly cast to an `Iterable_T` (no parentheses required) as long as the appropriate `Iterable_T` type has been accessed (see Section 6.18 [Autounravel], page 93). This can be done explicitly using the `range` function listed below.

The `collections.iter` module provides the following functions:

`Iterable_T Iterable(Iter_T iter())`
: creates an `Iterable_T` with its `operator iter` method set to `iter` (autounraveled whenever `Iterable_T` is accessed). This function is an alias for the constructor `Iterable_T(Iter_T iter())`.

`Iterable_T range(T[] items)`
: creates an `Iterable_T` that iterates over the elements of `items`. Separate calls to `operator iter` return independent iterators.

`Iter_T Iter_T(T[] items)`
: creates an `Iter_T` that iterates over the elements of `items`.

6.20.2.2 Combining (zipping) iterators

The `collections.zip2(K,V)` templated module provides a way of iterating over two containers of potentially different types in a single `for` loop, for example,

```
from collections.zip2(K=int, V=string) access zip, operator cast;
int[] x = {0, 1, 2, 3};
string[] y = {'0', '1', '2', '3'};
for (var ab : zip(x, y)) {
  assert(string(ab.k) == ab.v);
}
```

where

```
Iterable_Pair_K_V zip(Iterable_K a, Iterable_V b,
                      Pair_K_V keyword default=null)
```

returns an iterable that produces pairs of elements from `a` and `b`. If `default` is not provided, the iterable stops when either `a` or `b` runs out of elements. If `default` is provided, the

returned iterable will be the length of the longer of `a` and `b`, and the default value will be used for any missing elements.

Additionally, the module exposes the following types and all their autounravels:

type	from
`Pair_K_V`	`collections.genericpair(K,V)`
`Iterable_K`	`collections.iter(T=K)`
`Iterable_V`	`collections.iter(T=V)`
`Iterable_Pair_K_V`	`collections.iter(T=Pair_K_V)`
`Iter_K`	`collections.iter(T=K)`
`Iter_V`	`collections.iter(T=V)`
`Iter_Pair_K_V`	`collections.iter(T=Pair_K_V)`

The `collections.zip(T)` templated module provides a way to iterate over multiple containers in a single `for` loop. It is similar to `collections.zip2(K,V)`, but condenses the iterables into an iterable over arrays rather than over pairs. This requires that all the iterables be over the same type (for example, all string iterators or all integer iterators):

```
from collections.zip(T=int) access zip;
int[] x = {0, 1, 2, 3};
int[] y = {0, 1, 2};
int[] z = {0, 1, 2, 3, 4};
for (int[] abc : zip(default=-1, x, y, z)) {
  write(string(abc[0]) + '\t' + string(abc[1]) + '\t' + string(abc[2]));
}
```

which outputs

```
0       0       0
1       1       1
2       2       2
3       -1      3
-1      -1      4
```

The `collections.zip(T)` module exposes the types

type	from
`Iterable_array_T`	`collections.iter(T=T[])`
`Iter_array_T`	`collections.iter(T=T[])`
`Iterable_T`	`collections.iter(T)`
`Iter_T`	`collections.iter(T)`

and defines the following functions (all called `zip`):

`Iterable_array_T zip(...Iterable_T[] iterables)`
: zips together one or more iterables of `T`s. If `a` is an item from the result, then `a[0]` is from the first iterable, `a[1]` is from the second iterable, and so on. If the iterables are not all the same length, the result will be the length of the shortest iterable.

`Iterable_array_T zip(T keyword default ...Iterable_T[] iterables)`
: zips together one or more iterables of `T`s. The result will be the length of the longest iterable, with the default value filling in any missing elements.

`T[][] zip(...T[][] arrays)`
: zips together one or more arrays. If the arrays are not all the same length, the result will be the length of the shortest array.

`T[][] zip(T keyword default ...T[][] arrays)`
: zips together one or more arrays. The result will be the length of the longest array, with the default value filling in any missing elements. This is basically equivalent to applying `transpose` and then replacing any uninitialized elements with the default value.

6.20.2.3 Attaching a counter to an iterator

The `collections.enumerate(T)` module provides a way to attach a counter to an iterator. There are two functions, both called `enumerate`:

- `Iterable_Pair_int_T enumerate(Iterable_T iterable)`
- `Iterable_Pair_int_T enumerate(T[] array)`

Here's a simple example:

```
from collections.enumerate(T=int) access enumerate;
int[] x = {1, 4, 9, 16};
for (var ab : enumerate(x))
  write('Element ' + string(ab.k) + ' is ' + string(ab.v));
```

which outputs

```
Element 0 is 1
Element 1 is 4
Element 2 is 9
Element 3 is 16
```

Note that `enumerate` is not necessarily worthwhile for arrays since they are already indexed by integers:

```
int[] x = {1, 4, 9, 16};
for (int i=0; i < x.length; ++i)
  write('Element ' + string(i) + ' is ' + string(x[i]));
```

The `enumerate` utility is more useful for structures that do not have a builtin integer index (such as `Map` and `Set`).

6.20.3 Maps

Many modern programming languages have a builtin datastructure that is variously called a map, dictionary, or associative array. This structure is based on the mathematical concept of a function, but differs from the programming functions discussed previously (see Section 6.12 [Functions], page 69):

- A programming function computes an output value for each valid input. Unlike a mathematical function, it can have side effects.
- A `Map` stores a set of input-output pairs. It has a finite domain consisting of those inputs whose outputs were explicitly added. Typically, the inputs to a `Map` are called keys, and the outputs are called values.

This example of a `Map` assigns integer values to string keys:

```
from collections.hashmap(K=string, V=int) access
    HashMap_K_V as HashMap_string_int;

// Create a Map called h.
HashMap_string_int h;

// Add some key-value pairs to h.
h['hello'] = 5;
h['world'] = 3;

// Iterate over the keys (the domain) of h.
for (string key : h) {

  // Compute the output for the given input.
  int value = h[key];
  write(key + ' ' + string(value));
}
```

producing the output

```
hello 5
world 3
```

6.20.3.1 Initializing a `Map`

The `Map_K_V` structure itself is defined in the templated module `collections.map(K,V)`. To initialize a `Map_K_V`, you need to use a function from `collections.hashmap` or `collections.btreemap`:

`HashMap_K_V()`
`HashMap_K_V(V keyword nullValue, bool keyword isNullValue(V v) = new bool(V v) {return v == nullValue; })`
: These two functions, which create a hash map, must be accessed from the `collections.hashmap(K,V)` templated module. The key type must have a `hash` method, see Section 6.19 [Hash functions], page 94.

`BTreeMap_K_V()`
`BTreeMap_K_V(V keyword nullValue, bool keyword isNullValue(V v) = new bool(V v) {return v == nullValue; })`
: These two functions must be accessed from the `collections.btreemap(K,V)` templated module. They create a b-tree map, which stores its keys in sorted order. The key type must have a `<` operator.

Note: `HashMap_K_V` is an appropriate choice for most purposes.

If `nullValue` is not provided, looking up a missing key will throw an error. The `nullValue` parameter gives the map something to return when a nonexistent key is looked up. This is useful in scenarios where missing keys are expected and should be handled gracefully. The `isNullValue` parameter defaults to a function that checks whether its argument is equal to `nullValue` via `operator ==`. This is usually the desired behavior, but

it is occasionally necessary to override it; for instance, if you have overridden `operator ==` assuming its parameters are not null, then something like `isNullValue=new bool(V v){return alias(v,null);}` might be appropriate.

Both `HashMap_K_V` and `BTreeMap_K_V` can be implicitly cast to `Map_K_V`. Even without being cast, the two types have the same methods as `Map_K_V`. Similarly, the `nullValue` and `isNullValue` optional parameters behave the same way for both types. See Section 6.20.3.2 [The Map API], page 101.

In theory, adding, removing, or looking up an element should require, on average, $O(1)$ time for `HashMap_K_V` and $O(\log n)$ time for `BTreeMap_K_V` (where n represents the number of elements in the container). In practice, `HashMap_K_V` is faster than `BTreeMap_K_V` by about a factor of 2, assuming no more than a few million entries. (This is a very rough estimate that depends on the key type `K` having cheap hash, equality, and comparison functions.) The relative performance of these map implementations could change in the future as the code is optimized.

6.20.3.2 The `Map` API

The `Map_K_V` structure has the following methods. Note that its "subclasses" `HashMap_K_V` and `BTreeMap_K_V` each have the same fields.

Depending on which constructor is used, a `Map` may or may not have valid instances of `nullValue` and `isNullValue(V)`. If one of these is valid, the other will be too. In some methods below, an error will be thrown unless `nullValue` can be returned to indicate an element is not present. When `nullValue` is defined for a map `m`, the expression `m.isNullValue(m[k])` is equivalent to `m.contains(k)`.

`int size()`
: returns the number of distinct keys in the map.

`bool empty()`
: returns true iff `size() == 0`.

`bool contains(K key)`
: returns true if `key` is present as a key in the map.

`V operator [] (K key)`
: returns the value associated with the element `key`. If `key` is not present, this method either returns `nullValue` or, if `nullValue` does not exist, throws an error. See Section 6.10.4 [Bracket operators], page 68.

`void operator [=] (K key, V value)`
: adds the pair `(key, value)` to the map. If a key equivalent to `key` is already present, its associated value is replaced (as is the key itself, but that should rarely matter since the old and new keys are equivalent). However, if `isNullValue(value)` is true, then the assignment instead removes `key` from the map if it was already there. See Section 6.10.4 [Bracket operators], page 68.

`void delete(K key)`
: removes `key` from the map. If `key` was not already in the map, an error is thrown. (An alternative that will not throw an error is to set the value of `key` to `nullValue`.)

`Iter_K operator iter()`
: returns an iterator over the keys of the map. It allows the map to be used in range-based `for` loops, as in

```
for (K key : m) {<statements>}
```

If a key is deleted or a new key is inserted, then any iterators created before that change are undermined and may behave unpredictably. The current implementations attempt to throw errors if an iterator is used after being undermined, but this behavior is not guaranteed and might be removed in future versions for performance reasons.

`Iterable_K_V pairs()`
: returns an iterable over the key-value pairs. For example,

```
for (Pair_K_V p : m.pairs()) {
  K key = p.k;
  V value = p.v;
  <statements>
}
```

`K[] keys()`
: returns an array of all the keys. The array is a shallow copy; changes to it will not affect the map (and vice versa), but changes to the entries of the keys will undermine the integrity of the map.

`void add(Iterable_K_V other)`
: adds the key-value pairs from `other`. Running `m.add(o);` should be equivalent to running

```
for (var pr : o) { m[pr.k] = pr.v; }
```

In addition to these methods, there is also an autounraveled implicit cast from `Map_K_V` to `Iterable_K`. This cast allows a `Map_K_V` to be passed as a parameter to utilities like `zip` or `enumerate`.

6.20.4 Using Maps as Sets

Sometimes, you want a collection of objects of type `K` that lets you quickly add items, delete items, and determine whether an item is present. In Asymptote, the most straightforward option for such a collection is a `Map` from keys of type `K` to `bool` values, with `nullValue` set to `false`.

One needs to access some version of `Map_K_bool`. If `K` is hashable, one could write

```
from collections.hashmap(K=K,V=bool) access
    Map_K_V as Map_K_bool,
    HashMap_K_V as HashMap_K_bool;
Map_K_bool mset = HashMap_K_bool(nullValue=false);
```

If `K` is not hashable but does have `operator <`, something like this may be used instead:

```
from collections.btreemap(K=K,V=bool) access
    Map_K_V as Map_K_bool,
    BTreeMap_K_V as BTreeMap_K_bool;
Map_K_bool mset = BTreeMap_K_bool(nullValue=false);
```

To add an item `k` to `mset`, assign it the value `true`:

```
mset[k] = true;
```

If the item was already in the set, this statement will have no effect.

To test whether an item is in the set, you can either use the `contains` method or use brackets to access the value associated with `k`:

```
if (mset.contains(k))
  write('membership verified');
```

or, equivalently,

```
if (mset[k])
  write('membership verified');
```

To remove an item from the set, you can either use the `delete` method (which throws an error if the item was not already present) or set its value to `false` (which does nothing if the item was not already present):

```
mset.delete(k);  // guaranteed to reduce mset.size() by 1 or throw an error
mset[k] = false; // would delete k if we had not already deleted it
```

Once a key has its value set to `false`, it will not show up in iterators over the set, nor will it contribute to the `size()` of the set:

```
mset[k] = true;
int size1 = mset.size();
mset[k] = false;
assert(mset.size() == size1 - 1);
```

One can iterate over the set or check its size:

```
int count = 0;
for (K k : mset) { ++count; }
assert(mset.size() == count);
```

6.20.5 Sets

There is also a dedicated `Set_T` datastructure which is a bit more efficient (not storing an unneeded `bool`) and allows more precise methods (especially when it comes to sorted sets). The `Set_T` API attempts to respect the principle of reversibility, which states that a method that mutates the data structure should ideally return precisely the amount of information needed to reverse the modification. For instance, a method that removes an element should return that element — unless the element is already known to the caller (e.g., passed as an argument), in which case it suffices to return a boolean to indicate whether the removal was successful. (By contrast, the `Map_K_V` API follows the principle that a method should either return something or mutate the datastructure, but not both.)

6.20.5.1 Initializing a `Set`

The `Set_T` datastructure itself is defined in the templated module `collections.set(T)`. However, the `Set_T` structure is abstract, with many of its methods defaulting to `null`. To get a working implementation, you need either a `HashSet_T` or a `BTreeSet_T`:

```
HashSet_T()
HashSet_T(T nullT,           bool equiv(T a, T b) = operator ==,           bool
isNullT(T) = new bool(T t) { return equiv(t, nullT); }          )
```

These two functions, which create a hash set, must be accessed from the `collections.hashset(T)` templated module. The type `T` must have a `hash` method, see Section 6.19 [Hash functions], page 94. If `isNullT` is not defined, or if `!isNullT(null)`, then passing `null` to most methods will throw an error.

```
BTreeSet_T()
BTreeSet_T(T nullT, bool isNullT(T) = new bool(T t) { return t == nullT; })
```

These two functions must be accessed from the `collections.btree(T)` module. They create a b-tree set, which stores its elements in order based on the `<` operator. The `<` operator must define a strict weak ordering on all items of type `T`, except those satisfying `isNullT`. Two items are considered equivalent if neither is less than the other.

Accessing the `collections.btree(T)` module requires that `operator <` is defined for the type `T`; if not one can use the constructors defined in `collections.btreegeneral(T)` module, as discussed next.

```
BTreeSet_T(bool lessThan(T,T))
BTreeSet_T(bool lessThan(T,T),           T nullT,           bool isNullT(T)
= new bool(T t) { return t == nullT; })
```

These two functions must be accessed from the `collections.btreegeneral(T)` module. They create a BTreeSet that stores its elements in sorted order based on the `lessThan` function. The `lessThan` function must define a strict weak ordering on all items of type `T`, except those satisfying `isNullT`. Two items are considered equivalent if neither is less than the other.

Both `HashSet_T` and `BTreeSet_T` can be implicitly cast to `Set_T` and have all of the methods of `Set_T`. Similarly, the `nullT` and `isNullT` optional parameters behave the same way for both types. See Section 6.20.5.2 [The `Set` API], page 104.

Additionally, `BTreeSet_T` can be implicitly cast to `SortedSet_T` and has all its methods, which provide additional functionality specific to sorted sets (such as retrieving the least element or the next element greater than a given element). See Section 6.20.5.3 [The `SortedSet` API], page 107.

6.20.5.2 The `Set` API

Depending on which constructor is used, `nullT` and `isNullT` may or may not be defined for a `Set`. If one of these is defined the other will be too. In some methods below, an error will be thrown unless `nullT` can be returned to indicate an element is not present.

The `Set_T` structure has the following methods:

```
int size()
```

returns the number of distinct elements in the set.

```
bool empty()
```

returns true iff `size() == 0`.

`bool contains(T item)`
: returns true if `item` (or an equivalent item) is present in the set. If `isNullT` is defined and `isNullT(item)` is true, this method always returns false.

`bool add(T item)`
: adds `item` to the set if it is not already present. This method does nothing if `isNullT(item)` is defined and true, or if `item` or an equivalent item is already present. The return value is true iff `item` was added.

`void add(Iterable_T other)`
: adds the elements from `other` to the set. This method violates the principle of reversibility.

`bool delete(T item)`
: removes `item` from the set if it is present, otherwise does nothing. This method does nothing if `isNullT(item)` is defined and true. The return value is true iff the set was modified.

`void delete(Iterable_T other)`
: removes the elements from `other` from the set. Elements of `other` that are not in the set are ignored. This method violates the principle of reversibility.

`Iter_T operator iter()`
: returns an iterator over the elements of the set. It allows a set `s` to be used in a range-based `for` loop:

  `for (T item : s) {<statements>}`

  If an item is deleted or a new item is inserted, then any iterators created before that change are undermined and may behave unpredictably. The current implementations attempt to throw errors if an iterator is used after being undermined, but this behavior is not guaranteed and might be removed in future versions for performance reasons.

`Set_T newEmpty()`
: returns a new empty set with the same `nullT`, `isNullT`, equivalence relation, and (if applicable) ordering as the current set. Implementors of new set types should implement this method, otherwise the binary operators that are supposed to return a set will instead throw errors.

The preceeding methods are sufficient assuming that equivalent elements are interchangeable. However, on rare occasions, one may care which of two equivalent elements is in the set. For example, if the set was configured such that two ordered pairs are considered equivalent if they have the same first element, then the methods above would provide no way to look up a pair based on its first element. For this situation, the `Set_T` API provides the following methods:

`T get(T item)`
: returns the item in the set equivalent to `item`. If no such item exists, `nullT` is returned; if `nullT` is not defined, an error is thrown.

`T push(T item)`
: inserts `item` into the set even if it is equivalent to an existing item. If `item` is equivalent to an existing item, the replaced item is returned; otherwise `nullT`

is returned (unless `nullT` is not defined, in which case an error is thrown). If `isNullT(item)` is true, the set is not modified and `nullT` is returned.

The `push` method can be used to add a new item to the set, but only if `nullT` is defined.

`T extract(T item)`
: removes and returns an item in the set equivalent to `item`. If no such item exists, `nullT` is returned; if `nullT` is not defined, an error is thrown. If `isNullT(item)` is true, the set is not modified and `nullT` is returned.

In addition to the methods above, there are a number of autounraveled functions that deal with sets.

`Iterable_T operator cast(Set_T set)`
: implicitly casts a set to an iterable so that it can be passed as a parameter to utilities like `zip` or `enumerate`.

`bool operator <= (Set_T a, Set_T b)`
: returns true if `a` is a subset of `b`.

`bool operator >= (Set_T a, Set_T b)`
: returns true if `b` is a subset of `a`.

`bool operator == (Set_T a, Set_T b)`
: returns true if `a` and `b` are subsets of each other.

`bool operator != (Set_T a, Set_T b)`
: returns true if either `a` or `b` contains an element not in the other.

`bool sameElementsInOrder(Set_T a, Set_T b)`
: returns true if iterating over `a` and `b` produces the same elements in the same order.

The following operators leave their two arguments unchanged; the result is initialized by calling the `newEmpty()` method of the first argument to produce a new set with the same underlying implementation as the first argument.

`Set_T operator + (Set_T a, Iterable_T b)`
: returns the union of `a` and `b`.

`Set_T operator - (Set_T a, Set_T b)`
: returns a new set containing the elements of `a` that are not in `b`.

`Set_T operator & (Set_T a, Set_T b)`
: returns the intersection of `a` and `b`.

`Set_T operator ^ (Set_T a, Set_T b)`
: returns the symmetric difference of `a` and `b`, in other words, the set of all items in exactly one of `a,b`.

Note: While it is possible to use the `+=` and `-=` operators, they may not function as expected. In particular, these two operators will construct a new set, rather than modifying the set in place. For most purposes, the `add(Iterable_T)` and `delete(Iterable_T)` methods are preferred.

6.20.5.3 The `SortedSet` API

The `BTreeSet_T` implementation of `Set_T` (see Section 6.20.5.1 [Initializing a `Set`], page 103) maintains its elements in sorted order. For the `Set_T` API, this has no effect other than the iteration order. However, the `SortedSet_T` API provides additional methods to take advantage of this ordering. The `SortedSet_T` structure is defined in the `collections.sortedset(T)` module. Here is a list of the relations among the various APIs implemented by `BTreeSet_T`:

- `BTreeSet_T` has all the methods of `Set_T` and can be implicitly cast to `Set_T`.
- `BTreeSet_T` has all the methods of `SortedSet_T` and can be implicitly cast to `SortedSet_T`.
- `SortedSet_T` has all the methods of `Set_T` and can be implicitly cast to `Set_T`.
- All of these types can be implicitly cast to `Iterable_T`.

In addition to the methods of `Set_T`, `SortedSet_T` has the following methods:

`T min()` returns the least element in the set, or `nullT` if the set is empty. If the set is empty and `nullT` is not defined, an error is thrown.

`T max()` returns the greatest element in the set, or `nullT` if the set is empty. If the set is empty and `nullT` is not defined, an error is thrown.

`T popMin()`
: removes and returns the least element in the set, or returns `nullT` if the set is empty. If the set is empty and `nullT` is not defined, an error is thrown.

`T popMax()`
: removes and returns the greatest element in the set, or returns `nullT` if the set is empty. If the set is empty and `nullT` is not defined, an error is thrown.

`T after(T item)`
: returns the least element in the set that is strictly greater than `item`. If no such element exists, `nullT` is returned; if `nullT` is not defined, an error is thrown.

`T atOrAfter(T item)`
: returns the least element in the set that is greater than or equal to `item`. If no such element exists, `nullT` is returned; if `nullT` is not defined, an error is thrown.

`T before(T item)`
: returns the greatest element in the set that is strictly less than `item`. If no such element exists, `nullT` is returned; if `nullT` is not defined, an error is thrown.

`T atOrBefore(T item)`
: returns the greatest element in the set that is less than or equal to `item`. If no such element exists, `nullT` is returned; if `nullT` is not defined, an error is thrown.

As an example, the following code iterates over the elements of a sorted set `sset` between `a` (inclusive) and `b` (exclusive):

```
for (var x = sset.atOrAfter(a);
     !sset.isNullT(x) && x < b;
```

```
     x = sset.after(x)) {
  <statements using x>
}
```

Of course, this code assumes that `nullT` is defined for `sset`.

Note that if you want to iterate over most or all of the elements in a sorted set, it is usually more efficient to use a range-based `for` loop and skip any elements that are not in the desired range:

```
for (var x : sset) {
  if (x < a) continue;
  if (x >= b) break;
  <statements using x>
}
```

6.20.6 Using Sets as Maps

In mathematics, a map is sometimes defined as a set of ordered pairs. In the Asymptote programming language, this definition can be taken literally: a `Set_T` of `Pair_K_V` objects can be used as a map from `K` to `V`, provided that the equivalence, hash, and comparison functions are configured to look only at the first element of the pair. This can be useful in particular for sorted sets, which have functionality such as `atOrAfter` and `after` that is not currently available in the `Map` API.

We illustrate this approach by implementing a piecewise linear function (in this case we can use the builtin `pair` type rather than a `Pair_K_V`).

```
from collections.btreegeneral(T=pair) access
    SortedSet_T as SortedSet_pair,
    BTreeSet_T as BTreeSet_pair;

bool keysLT(pair a, pair b) { return a.x < b.x; }
bool isnan(pair a) { return isnan(a.y); }

struct PLFunction {
  SortedSet_pair pairs = BTreeSet_pair(
    lessThan=keysLT,
    nullT=(nan,nan),
    isNullT = isnan
  );
  void operator [=] (real x, real y) {
    pairs.push((x,y));
  }
  real operator [] (real x) {
    pair p = (x, 0);
    pair left = pairs.atOrBefore(p);
    pair right = pairs.atOrAfter(p);
    if (isnan(left.y) || isnan(right.y)) {
      return nan;
    }
    if (left.x == right.x) {
```

```
      return left.y;
    }
    real t = (x - left.x) / (right.x - left.x);
    return interp(left.y, right.y, t);
  }
}

PLFunction f;
f[0] = 17;
f[1] = 18;
f[2001] = 19;

for (real x : new real[] {-1, 0, 0.5, 1, 2, 2001, 2002}) {
  write('f[' + (string)x + '] = ' + (string)f[x]);
}
```

outputs

```
f[-1] = nan
f[0] = 17
f[0.5] = 17.5
f[1] = 18
f[2] = 18.0005
f[2001] = 19
f[2002] = nan
```

6.20.7 Queues

Two standard datastructures are stacks and queues. A stack is a last-in-first-out (LIFO) structure; in Asymptote, arrays can be used as stacks via the `push` and `pop` methods. A queue is a first-in-first-out (FIFO) structure; the `collections.queue(T)` module provides a queue implementation.

`Queue_T makeQueue(T[] initialData)`
: returns a queue initialized with a copy of the array `initialData`. The parameter `initialData` is required in order to specify the type `T` in case there are multiple `Queue_T` types in the current scope. If no initial data is needed, use `makeQueue(new T[])` to create an empty array as a parameter.

The `Queue_T` structure has the following methods:

`void push(T item)`
: adds `item` to the end of the queue.

`T pop()`
: removes and returns the first item in the queue. If the queue is empty, an error is thrown.

`T peek()`
: returns the first item in the queue without removing it. If the queue is empty, an error is thrown.

`int size()`
: returns the number of items in the queue.

`Iter_T operator iter()`
: returns an iterator over the items in the queue, in the order they will be returned by `pop()`.

  If an item is pushed or popped, then any iterators created before that change are undermined and may behave unpredictably. The current implementation attempts to throw errors if an iterator is used after being undermined, but this behavior is not guaranteed and might be removed in future versions for performance reasons.

In addition to these methods, there is also an autounraveled implicit cast from `Queue_T` to `Iterable_T`. This cast allows a `Queue_T` to be passed as a parameter to utilities like `zip` or `enumerate`.

6.20.8 Wrapping arrays

`Asymptote` arrays have a builtin `==` operator that returns an array of bools rather than a single bool. This is useful in a number of contexts, but it also prevents them from being used in pairs or as values in hashmaps. The `collections.wraparray(T)` templated module provides a wrapper for arrays that has a more conventional `==` operator that returns a single bool. It also provides a `hash()` method that, if initialized, allows the array to be used as a key in a hashmap.

The main type in `collections.wraparray(T)` is `Array_T`. While `Array_T` is not a drop-in replacement for `T[]`, it is as close as any structure can be. For convenience, the function `Array_T wrap(T[] data)` can be used to convert an array to an `Array_T`. Additionally, `Array_T` has autounraveled implicit casts to and from `T[]`, so that it can be used in most places where `T[]` is expected (and vice versa). Note that these casts do *not* make copies, so if you modify the original array, the changes will be reflected in the `Array_T` and vice versa.

The methods and fields of `Array_T` are identical to those of `T[]`, with the following exceptions:

`int length()`
: is a method (requires parentheses afterwards) that returns the length of the array.

`void cyclic(bool b)`
: sets the `cyclic` field of the wrapped array.

`bool cyclic()`
: returns the `cyclic` field of the wrapped array.

`int[] keys()`
: is a method (requires parentheses afterwards) that returns the `keys` field of the wrapped array.

`T[] data` is a field that holds the wrapped array.

Additionally, the iteration syntax `for (T x : a)` is supported if `a` is an `Array_T`, as is element access `a[i]`. However, slices are not supported.

To make an `Array_T` hashable, pass a second argument to the `wrap` function that computes an integer hash of a single element. For instance, the following constructs hashable arrays of strings and integers:

```
from collections.wraparray(T=string) access Array_T as Array_string;
from collections.wraparray(T=int) access Array_T as Array_int;
string[] s = {'hello', 'world'};
Array_string ss = wrap(s, new int(string s) { return s.hash(); });
int[] a = {1, 2, 3};
Array_int aa = wrap(a, new int(int i) { return i; });
```

The resulting hash function works by converting all the elements of the array to integers using the provided function, and then hashing the resulting `int[]`, see Section 6.19 [Hash functions], page 94.

Important note: If you use an `Array_T` as a key in a hashmap, you must ensure that the array is not modified while it is a key in the hashmap.

7 LaTeX usage

Asymptote comes with a convenient LaTeX style file asymptote.sty (v1.36 or later required) 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 trailing commands/comments. A blank line is not allowed after \begin{asy}.

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

```
latex latexusage
asy latexusage-*.asy
latex latexusage
```

or

```
pdflatex latexusage
asy latexusage-*.asy
pdflatex latexusage
```

To switch between using inline Asymptote code with latex and pdflatex you may first need to remove the files latexusage-*.tex.

An even better method for processing a LaTeX file with embedded Asymptote code is to use the latexmk utility from

https://mirror.ctan.org/support/latexmk/

after putting the contents of https://raw.githubusercontent.com/vectorgraphics/asymptote/HEAD/doc/latexmkrc in a file latexmkrc in the same directory. The command

```
latexmk -pdf latexusage
```

will then call Asymptote automatically, recompiling only the figures that have changed. Since each figure is compiled in a separate system process, this method also tends to use less memory. To store the figures in a separate directory named asy, one can define

```
\def\asydir{asy}
```

in latexusage.tex. External Asymptote code can be included with

```
\asyinclude[<options>]{<filename.asy>}
```

so that latexmk will recognize when the code is changed. Note that latexmk requires perl, available from https://www.perl.org/.

One can specify width, height, keepAspect, viewportwidth, viewportheight, attach, and inline. keyval-style options to the asy and asyinclude environments. Three-dimensional PRC files may either be embedded within the page (the default) or attached as annotated (but printable) attachments, using the attach option and the attachfile2 (or older attachfile) LaTeX package. The inline option generates inline LaTeX code instead of EPS or PDF files. This makes 2D LaTeX symbols visible to the \begin{asy}...\end{asy} environment. In this mode, Asymptote correctly aligns 2D 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 the `latex` TeX engine is used with the `inline` option, 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).

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 by default:
\usepackage[inline]{asymptote}

% Use this form with latex or pdflatex to create PDF attachments by default:
%\usepackage[attach]{asymptote}

% Enable this line to support the attach option:
%\usepackage[dvips]{attachfile2}

\begin{document}

% Optional subdirectory for latex files (no spaces):
\def\asylatexdir{}
% Optional subdirectory for asy files (no spaces):
\def\asydir{}

\begin{asydef}
// Global Asymptote definitions can be put here.
import three;
usepackage("bm");
texpreamble("\def\V#1{\bm{#1}}");
// One can globally override the default toolbar settings here:
// settings.toolbar=true;
\end{asydef}

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

\def\A{A}
\def\B{\V{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;
path c1=circle(z1,r);
path 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);

//draw("$\A$",box,z1);              // Requires [inline] package option.
//draw(Label("$\B$","$B$"),box,z2); // Requires [inline] package option.
draw("$A$",box,z1);
draw("$\V{B}$",box,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. This 3D example can be viewed
interactively either with Adobe Reader or Asymptote's fast OpenGL-based
renderer. To support {\tt latexmk}, 3D figures should specify
\verb+inline=true+. It is sometimes desirable to embed 3D files as annotated
attachments; this requires the \verb+attach=true+ option as well as the
\verb+attachfile2+ \LaTeX\ package.
\begin{center}
```

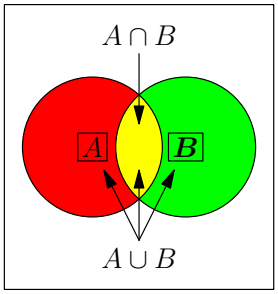
```
\begin{asy}[height=4cm,inline=true,attach=false,viewportwidth=\linewidth]
currentprojection=orthographic(5,4,2);
draw(unitcube,blue);
label("$V-E+F=2$",(0,1,0.5),3Y,blue+fontsize(17pt));
\end{asy}
\end{center}

One can also scale the figure to the full line width:
\begin{center}
\begin{asy}[width=\the\linewidth,inline=true]
pair z0=(0,0);
pair z1=(2,0);
pair z2=(5,0);
pair zf=z1+0.75*(z2-z1);

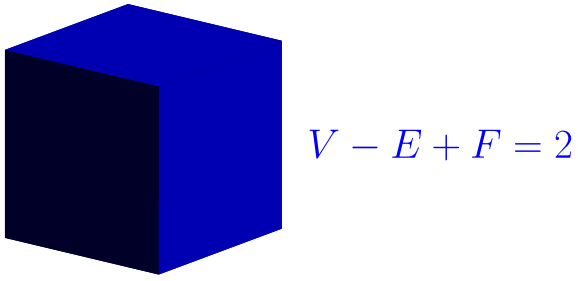
draw(z1--z2);
dot(z1,red+0.15cm);
dot(z2,darkgreen+0.3cm);
label("$m$",z1,1.2N,red);
label("$M$",z2,1.5N,darkgreen);
label("$\hat{\ }$",zf,0.2*S,fontsize(24pt)+blue);

pair s=-0.2*I;
draw("$x$",z0+s--z1+s,N,red,Arrows,Bars,PenMargins);
s=-0.5*I;
draw("$\bar{x}$",z0+s--zf+s,blue,Arrows,Bars,PenMargins);
s=-0.95*I;
draw("$X$",z0+s--z2+s,darkgreen,Arrows,Bars,PenMargins);
\end{asy}
\end{center}
\end{document}
```

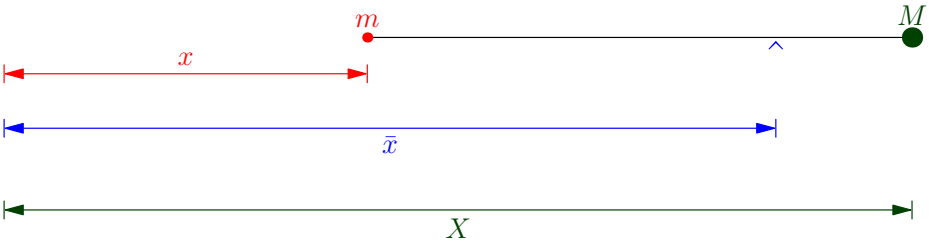
Here is a venn diagram produced with Asymptote, drawn to width 4cm:



Each graph is drawn in its own environment. One can specify the width and height to LaTeX explicitly. This 3D example can be viewed interactively either with Adobe Reader or Asymptote's fast OpenGL-based renderer. To support `latexmk`, 3D figures should specify `inline=true`. It is sometimes desirable to embed 3D files as annotated attachments; this requires the `attach=true` option as well as the `attachfile2` LaTeX package.



One can also scale the figure to the full line width:



8 User modules

These are the most commonly used base modules for creating graphics and visualizations.

8.1 graph

This module 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, real T(real)=identity,
              interpolate join=operator --);
  guide[] graph(picture pic=currentpicture, real f(real), real a, real b,
                int n=ngraph, real T(real)=identity, bool3 cond(real),
                interpolate join=operator --);
  ```

  return a graph using the scaling information for picture `pic` (see [automatic scaling], page 134) of the function `f` on the interval [`T(a)`,`T(b)`], sampling at `n` points evenly spaced in [`a`,`b`], optionally restricted by the bool3 function `cond` on [`a`,`b`]. If `cond` is:

  - `true`, the point is added to the existing guide;
  - `default`, the point is added to a new guide;
  - `false`, the point is omitted and a new guide is begun.

  The points are connected using the interpolation specified by `join`:

  - `operator --` (linear interpolation; the abbreviation `Straight` is also accepted);
  - `operator ..` (piecewise Bezier cubic spline interpolation; the abbreviation `Spline` is also accepted);
  - `linear` (linear interpolation),
  - `Hermite` (standard cubic spline interpolation using boundary condition `notaknot`, `natural`, `periodic`, `periodicOffset`,[1] `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, real T(real)=identity,
              interpolate join=operator --);
  guide[] graph(picture pic=currentpicture, real x(real), real y(real),
                real a, real b, int n=ngraph, real T(real)=identity,
                bool3 cond(real), interpolate join=operator --);
  ```

  returns a graph using the scaling information for picture `pic` of the parametrized function (`x`(t),`y`(t)) for t in the interval [`T(a)`,`T(b)`], sampling at `n` points evenly spaced in [`a`,`b`], optionally restricted by the bool3 function `cond` on [`a`,`b`], using the given interpolation type.

[1] like `periodic` but for data that repeats after a constant offset, as in $y = x + \sin x$.

- ```
  guide graph(picture pic=currentpicture, pair z(real), real a, real b,
              int n=ngraph, real T(real)=identity,
              interpolate join=operator --);
  guide[] graph(picture pic=currentpicture, pair z(real), real a, real b,
                int n=ngraph, real T(real)=identity, bool3 cond(real),
                interpolate join=operator --);
  ```
  returns a graph using the scaling information for picture `pic` of the parametrized function `z`(t) for t in the interval [`T(a)`,`T(b)`], sampling at `n` points evenly spaced in [`a`,`b`], optionally restricted by the bool3 function `cond` on [`a`,`b`], using the given interpolation type.
- ```
  guide graph(picture pic=currentpicture, pair[] z,
              interpolate join=operator --);
  guide[] graph(picture pic=currentpicture, pair[] z, bool3[] cond,
                interpolate join=operator --);
  ```
  returns a graph using the scaling information for picture `pic` of the elements of the array `z`, optionally restricted to those indices for which the elements of the boolean array `cond` are `true`, using the given interpolation type.
- ```
  guide graph(picture pic=currentpicture, real[] x, real[] y,
              interpolate join=operator --);
  guide[] graph(picture pic=currentpicture, real[] x, real[] y,
                bool3[] cond, interpolate join=operator --);
  ```
  returns a graph using the scaling information for picture `pic` of the elements of the arrays (`x`,`y`), optionally restricted to those indices for which the elements of the boolean array `cond` are `true`, using the given interpolation type.
- ```
  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.
- ```
  guide polargraph(picture pic=currentpicture, real[] r, real[] theta,
                   interpolate join=operator--);
  ```
  returns a polar-coordinate graph using the scaling information for picture `pic` of the elements of the arrays (`r`,`theta`), using 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 above=false);
  ```

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 19), 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 14). The axis is drawn before any existing objects in `pic` unless `above=true`. 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 the module `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 30. If the format string is `trailingzero`, trailing zeros will be added to the tick labels; 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 134), divide a linear axis evenly into this many intervals, separated by major ticks; for a logarithmic axis, this is the number of decades between labelled ticks;

`int n`
: divide each 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 tickmodifiers `OmitTick(...real[] x)`, `OmitTickInterval(real a, real b)`, and `OmitTickIntervals(real[] a, real[] b)` 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 abbreviation 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 above=false,
             bool autorotate=true);
  ```

  Draw a y axis on picture `pic` from y=`ymin` to y=`ymax` using pen `p`, optionally labelling it with a Label `L` that is autorotated unless `autorotate=false`. The relative location of the label (a real number from [0,1]) defaults to 1 (see [Label], page 19). 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 14). The axis is drawn before any existing objects in `pic` unless `above=true`. 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 above=true,
            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 above=true,
            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 axes(picture pic=currentpicture, Label xlabel="", Label ylabel="",
          bool extend=true,
          pair min=(-infinity,-infinity), pair max=(infinity,infinity),
          pen p=currentpen, arrowbar arrow=None, bool above=false);
```

This convenience routine draws both x and y axes on picture `pic` from `min` to `max`, with optional labels `xlabel` and `ylabel` and any arrows specified by `arrow`. The axes are drawn on top of existing objects in `pic` only if `above=true`.

-

```
void axis(picture pic=currentpicture, Label L="", path g,
          pen p=currentpen, ticks ticks, ticklocate locate,
          arrowbar arrow=None, int[] divisor=new int[],
          bool above=false, 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 before any existing objects in `pic` unless `above=true`. 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 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=currentpen);
void labelx(picture pic=currentpicture, Label L="", real x,
            align align=S, string format="", pen p=currentpen);
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=currentpen);
void labely(picture pic=currentpicture, Label L="", real y,
            align align=W, string format="", pen p=currentpen);
void labely(picture pic=currentpicture, Label L,
            string format="", explicit pen p=currentpen);
```

Here are some simple examples of two-dimensional graphs:

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

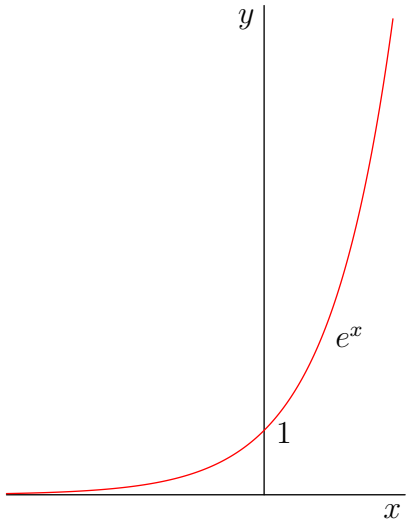
```
import graph;
size(150,0);

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

draw(graph(f,-4,2,operator ..),red);

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

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 (see Chapter 13 [GUI], page 210). 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);

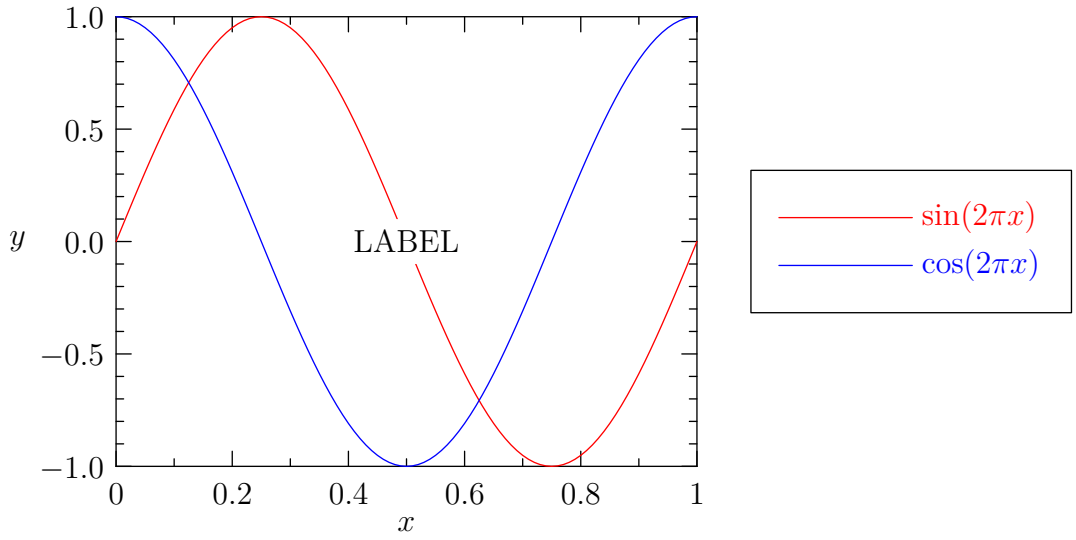
real Sin(real t) {return sin(2pi*t);}
real Cos(real t) {return cos(2pi*t);}

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(trailingzero));

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(trailingzero));

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

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

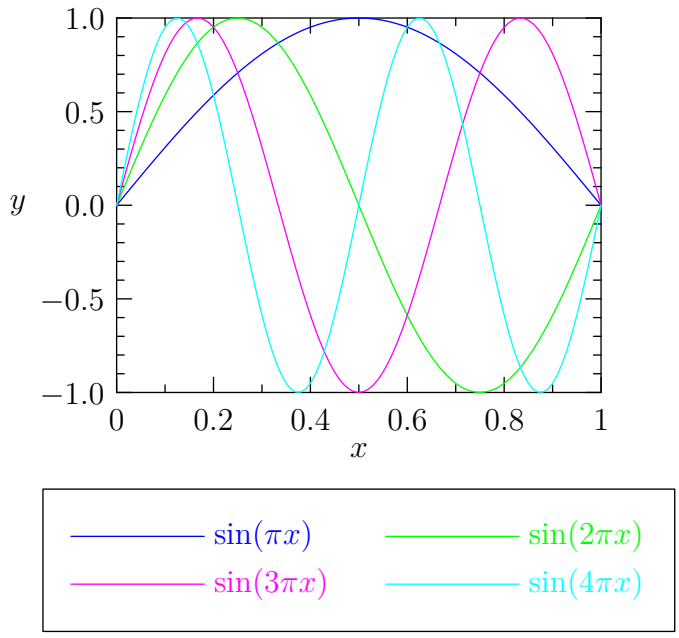
A legend can have multiple entries per line:

```
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(trailingzero));
```

```
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.

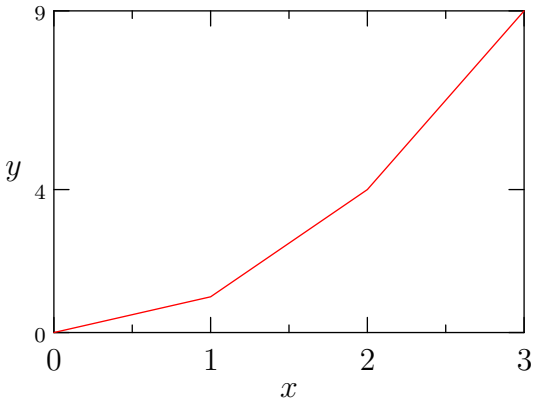
```
import graph;

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(8pt)),new real[]{0,4,9}));
```



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

```
import graph;

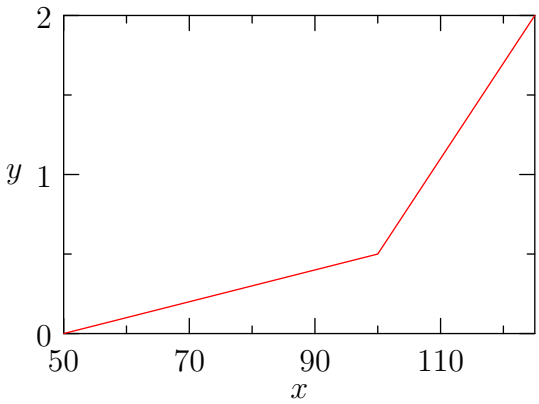
size(200,150,IgnoreAspect);

file in=input("filegraph.dat").line();
real[][] a=in;
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 above=true);
```

using the path `unitcircle` (see [filltype], page 52), are drawn below each node. Any frame can be converted to a marker, using

```
marker marker(frame f, markroutine markroutine=marknodes,
              bool above=true);
```

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). Alternatively, one can use markroutine `marknodes` to request that the marks be placed at each Bezier node of the path, or markroutine `markuniform(pair z(real t), real a, real b, int n)` to place marks at points `z(t)` for n evenly spaced values of `t` in `[a,b]`.

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(diamond),marker(plus);
};

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

scale(pic,true);

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)));

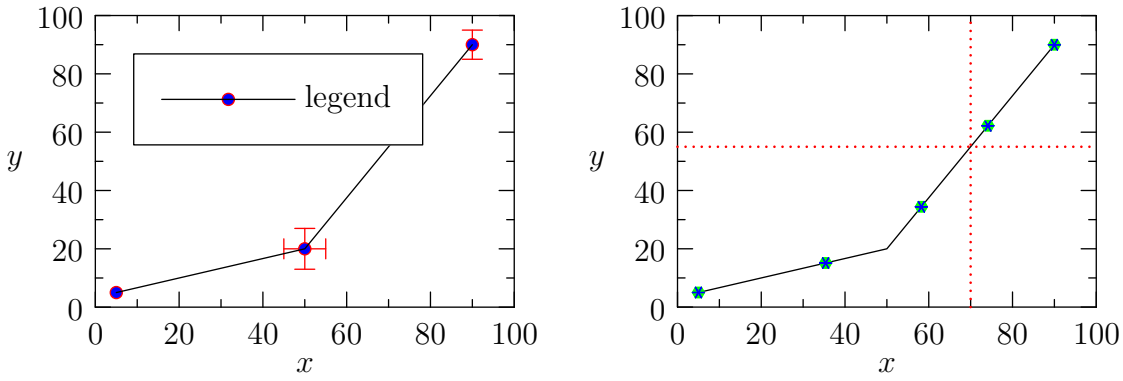
scale(pic2,true);

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. A custom mark routine can be also be specified:

```
import graph;

size(200,100,IgnoreAspect);

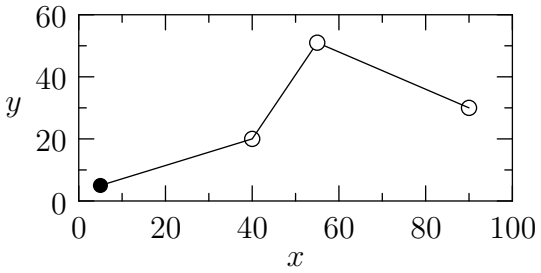
markroutine marks() {
  return new void(picture pic=currentpicture, frame f, path g) {
    path p=scale(1mm)*unitcircle;
    for(int i=0; i <= length(g); ++i) {
      pair z=point(g,i);
      frame f;
      if(i % 4 == 0) {
        fill(f,p);
        add(pic,f,z);
      } else {
        if(z.y > 50) {
          pic.add(new void(frame F, transform t) {
              path q=shift(t*z)*p;
              unfill(F,q);
              draw(F,q);
            });
        } else {
          draw(f,p);
          add(pic,f,z);
        }
      }
    }
  };
}

pair[] f={(5,5),(40,20),(55,51),(90,30)};

draw(graph(f),marker(marks()));

scale(true);
```

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



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

```
import graph;

size(400,150,IgnoreAspect);

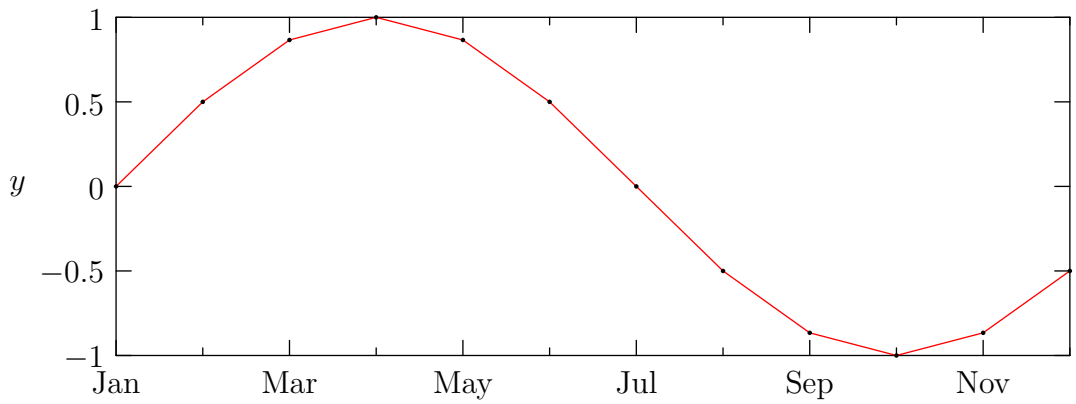
real[] x=sequence(12);
real[] y=sin(2pi*x/12);

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));
```



8. The next 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

```
void limits(picture pic=currentpicture, pair min, pair max, bool crop=NoCrop);
```

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)` can be used to crop a graph to the current graph limits.

```
import graph;

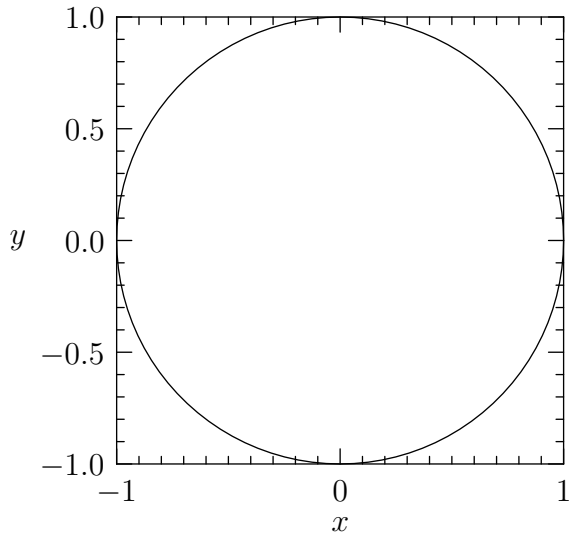
size(0,200);

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

draw(graph(x,y,0,1));

//limits((0,-1),(1,0),Crop);

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

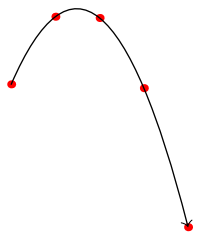


The function

```
guide graphwithderiv(pair f(real), pair fprime(real), real a, real b,
                     int n=ngraph#10);
```

can be used to construct the graph of the parametric function `f` on `[a,b]` with the control points of the `n` Bezier segments determined by the specified derivative `fprime`:

```
unitsize(2cm);
import graph;
pair F(real t) {
  return (1.3*t,-4.5*t^2+3.0*t+1.0);
}
pair Fprime(real t) {
  return (1.3,-9.0*t+3.0);
}
path g=graphwithderiv(F,Fprime,0,0.9,4);
dot(g,red);
draw(g,arrow=Arrow(TeXHead));
```



The next example illustrates how one can extract a common axis scaling factor.

```
import graph;

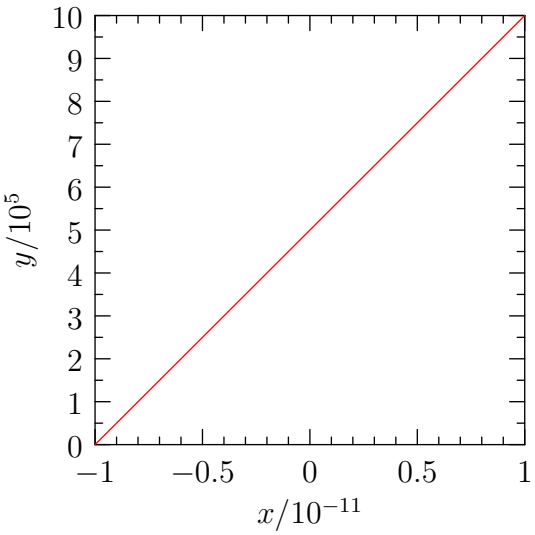
axiscoverage=0.9;
size(200,IgnoreAspect);

real[] x={-1e-11,1e-11};
real[] y={0,1e6};

real xscale=round(log10(max(x)));
real yscale=round(log10(max(y)))-1;

draw(graph(x*10^(-xscale),y*10^(-yscale)),red);

xaxis("$x/10^{"+(string) xscale+"}$",BottomTop,LeftTicks);
yaxis("$y/10^{"+(string) yscale+"}$",LeftRight,RightTicks(trailingzero));
```



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 x and y components can be individually scaled using the analogous routines

```
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=false` and `automax=automin`. These default to `false` but can be respectively set to `true` to enable automatic selection of "nice" axis minimum and maximum values. The `Linear` scaling can also take as optional final arguments a multiplicative scaling factor and intercept (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;}
```

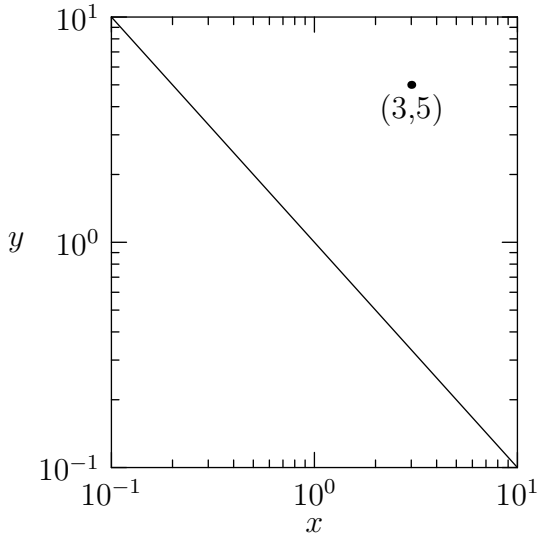
```
scale(Log,Log);

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

//limits((1,0.1),(10,0.5),Crop);

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

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

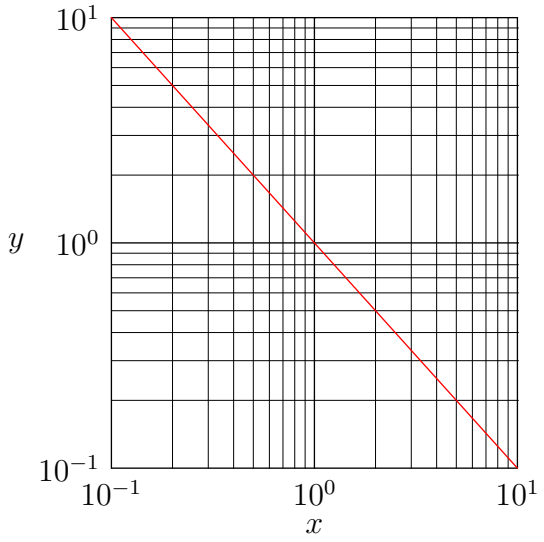


By extending the ticks, one can easily produce a logarithmic grid:

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

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

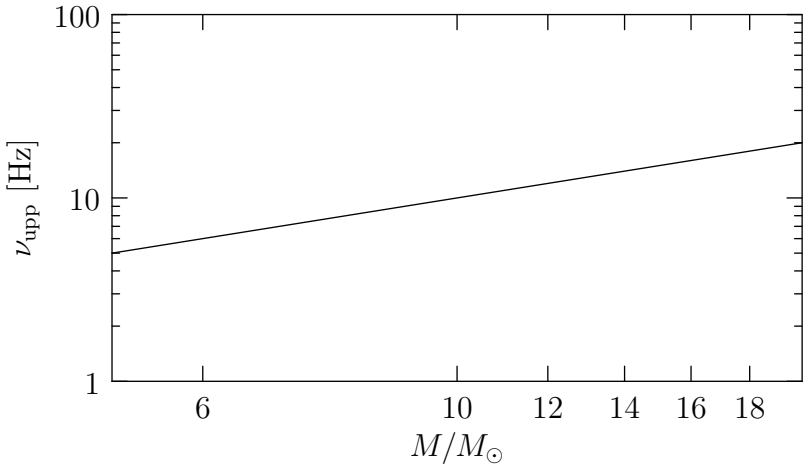
scale(Log,Log);
draw(graph(f,0.1,10),red);
pen thin=linewidth(0.5*linewidth());
xaxis("$x$",BottomTop,LeftTicks(begin=false,end=false,extend=true,
                                ptick=thin));
yaxis("$y$",LeftRight,RightTicks(begin=false,end=false,extend=true,
                                 ptick=thin));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



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:

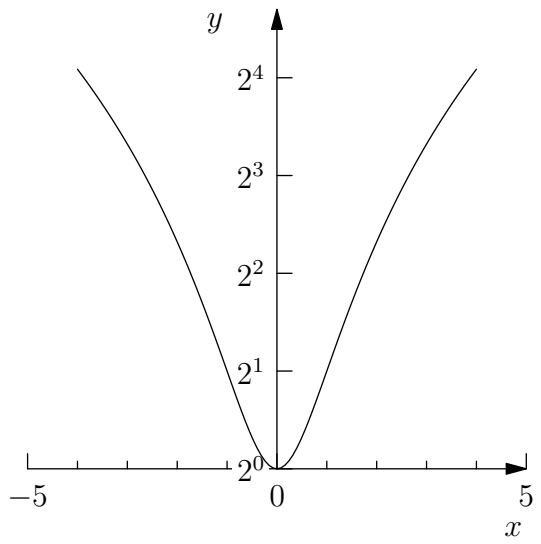
real log2(real x) {static real log2=log(2); return log(x)/log2;}
real pow2(real x) {return 2^x;}

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

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

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

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

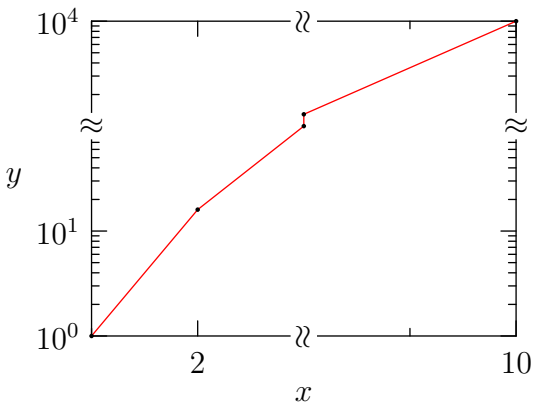
```
scale(Broken(a,b),BrokenLog(c,d));

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)));
```



9. `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}");

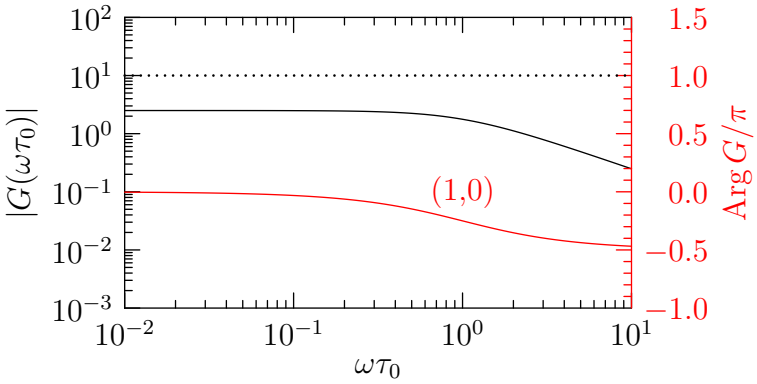
size(10cm,5cm,IgnoreAspect);

real ampl(real x) {return 2.5/sqrt(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=input(data).line().csv();

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

real[][] a=in;
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];

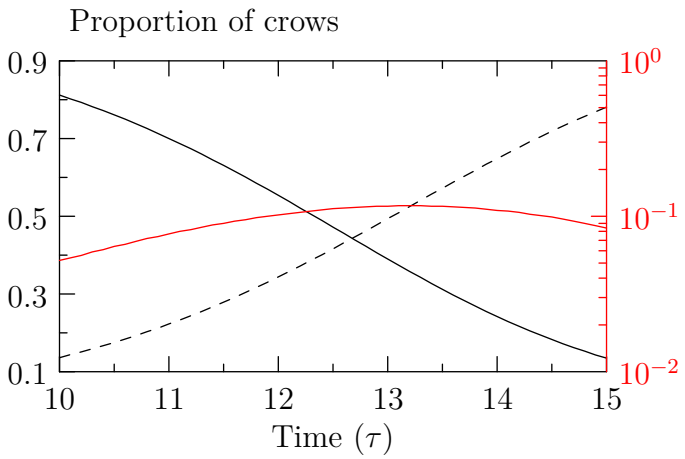
scale(true);
```

```
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(true),Log(true));
    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);
```



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

```
import graph;
import stats;

size(400,200,IgnoreAspect);

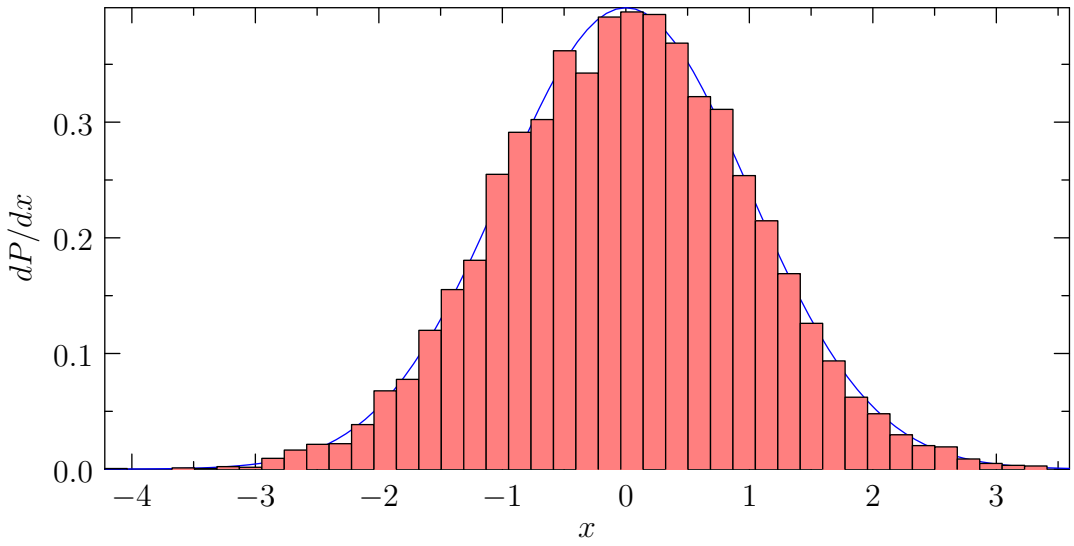
int n=10000;
real[] a=new real[n];
for(int i=0; i < n; ++i) a[i]=Gaussrand();

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

// Optionally calculate "optimal" number of bins a la Shimazaki and Shinomoto.
int N=bins(a);

histogram(a,min(a),max(a),N,normalize=true,low=0,lightred,black,bars=true);
```

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



11. 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=input("leastsquares.dat").line();

real[][] a=fin;
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(true),Linear(true));

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);

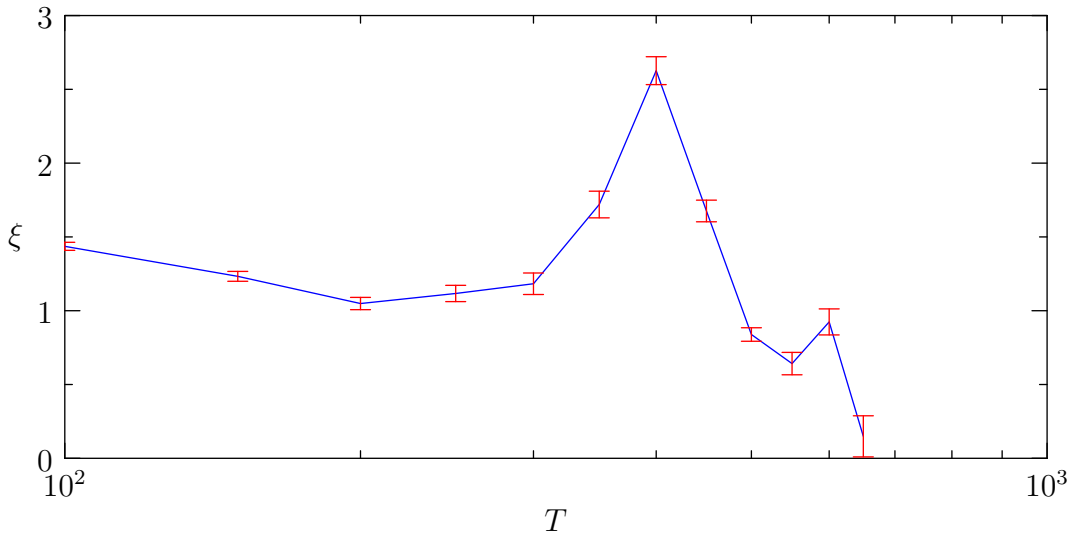
  T.push(first);
  xi.push(L.m);
  dxi.push(L.dm);
}

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

crop();

ylimits(0);

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



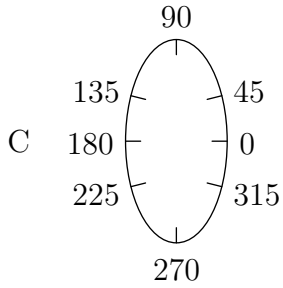
12. Here is an example that illustrates the general `axis` routine.

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

path g=ellipse((0,0),1,2);

scale(true);

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];}));
```



13. To draw a vector field of `n` arrows evenly spaced along the arclength of a path, use the routine

```
picture vectorfield(path vector(real), path g, int n, bool truesize=false,
                    pen p=currentpen, arrowbar arrow=Arrow);
```

as illustrated in this simple example of a flow field:

```
import graph;
defaultpen(1.0);
```

```
size(0,150,IgnoreAspect);

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

typedef path vector(real);

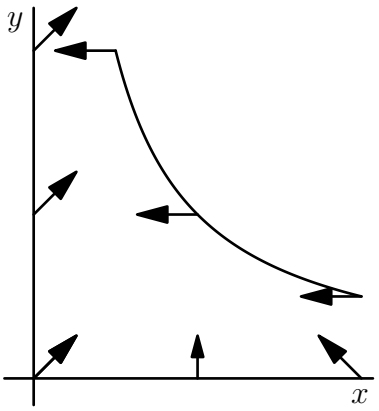
// Return a vector interpolated linearly between a and b.
vector vector(pair a, pair b) {
  return new path(real x) {
    return (0,0)--arrowlength*interp(a,b,x);
  };
}

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

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

int n=3;
draw(g);
xaxis("$x$");
yaxis("$y$");

add(vectorfield(vector(W,W),g,n,true));
add(vectorfield(vector(NE,NW),(0,0)--(point(E).x,0),n,true));
add(vectorfield(vector(NE,NE),(0,0)--(0,point(N).y),n,true));
```



14. To draw a vector field of `nx`×`ny` arrows in `box(a,b)`, use the routine

```
picture vectorfield(path vector(pair), pair a, pair b,
                    int nx=nmesh, int ny=nx, bool truesize=false,
                    real maxlength=truesize ? 0 : maxlength(a,b,nx,ny),
```

```
                    bool cond(pair z)=null, pen p=currentpen,
                    arrowbar arrow=Arrow, margin margin=PenMargin)
```

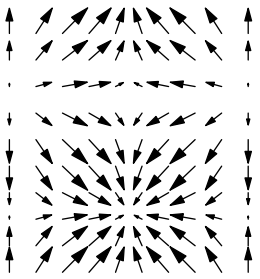
as illustrated in this example:

```
import graph;
size(100);

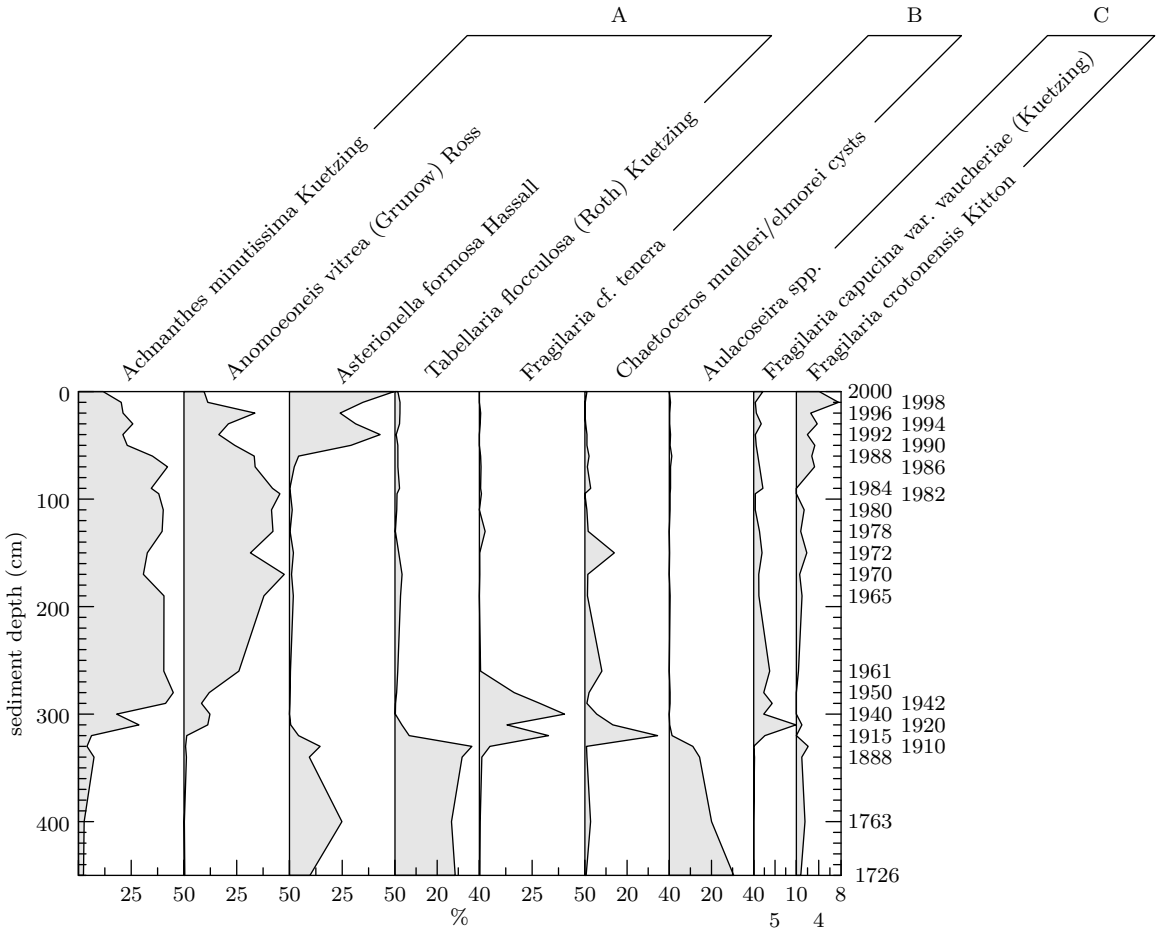
pair a=(0,0);
pair b=(2pi,2pi);

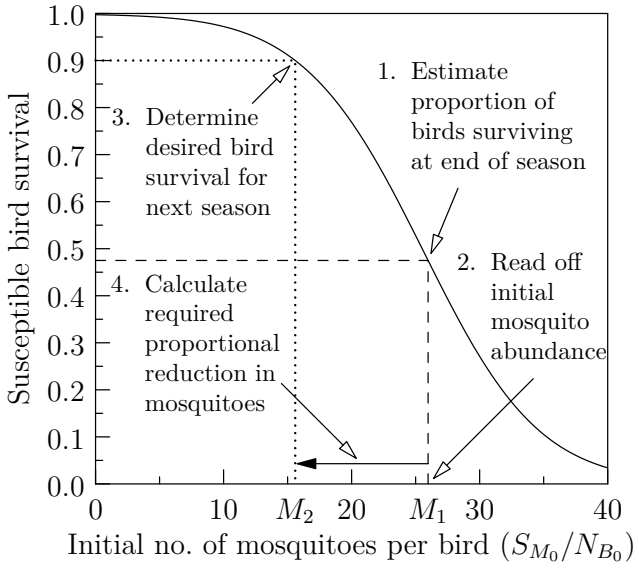
path vector(pair z) {return (sin(z.x),cos(z.y));}

add(vectorfield(vector,a,b));
```



15. 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
25 50 25 50 25 50 20 40 25 50 20 40 20 40 10 8
%
5 4


Susceptible bird survival
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
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})

8.2 three

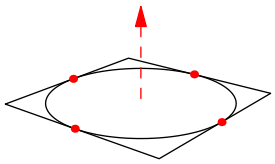
This module fully extends the notion of guides and paths in Asymptote to three dimensions. It introduces the new types guide3, path3, and surface. Guides in three dimensions are specified with the same syntax as in two dimensions except that triples (x,y,z) are used in place of pairs (x,y) for the nodes 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 [see J. C. Bowman, Proceedings in Applied Mathematics and Mechanics, 7:1, 2010021-2010022 (2007)].

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

```
import three;

size(100);

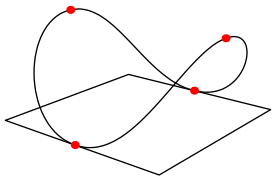
path3 g=(1,0,0)..(0,1,0)..(-1,0,0)..(0,-1,0)..cycle;
draw(g);
draw(O--Z,red+dashed,Arrow3);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle));
dot(g,red);
```



and then distorted into a saddle:

```
import three;

size(100,0);
path3 g=(1,0,0)..(0,1,1)..(-1,0,0)..(0,-1,1)..cycle;
draw(g);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle));
dot(g,red);
```



Module three provides constructors for converting two-dimensional paths to three-dimensional ones, and vice-versa:

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

```
path path(path3 p, pair P(triple)=xypart);
```

A Bezier surface, the natural two-dimensional generalization of Bezier curves, is defined in `three_surface.asy` as a structure containing an array of Bezier patches. Surfaces may drawn with one of the routines

```
void draw(picture pic=currentpicture, surface s, int nu=1, int nv=1,
          material surfacepen=currentpen, pen meshpen=nullpen,
          vertexPen vertexPen=null,
          light light=currentlight, light meshlight=nolight, string name="",
          render render=defaultrender);
void draw(picture pic=currentpicture, surface s, int nu=1, int nv=1,
          material[] surfacepen, pen meshpen,
          light light=currentlight, light meshlight=nolight, string name="",
          render render=defaultrender);
void draw(picture pic=currentpicture, surface s, int nu=1, int nv=1,
          material[] surfacepen, pen[] meshpen=nullpens,
          vertexPen vertexPen=null,
          light light=currentlight, light meshlight=nolight, string name="",
          render render=defaultrender);
```

The parameters `nu` and `nv` specify the number of subdivisions for drawing optional mesh lines for each Bezier patch (these parameters are ignored for Bezier triangles). The optional `name` parameter is used as a prefix for naming the surface patches in the PRC model tree. The optional `vertexPen` parameter specifies a function `pen(surfaceVertex)` that is evaluated at each patch corner to assign vertex colors (see Section 8.4.1 [Surface coloring], page 173). Here material is a structure defined in `three_light.asy`:

```
struct material {
  pen[] p; // diffusepen,emissivepen,specularpen
  real opacity;
  real shininess;
  real metallic;
  real fresnel0;
}
```

These material properties are used to implement physically based rendering (PBR) using light properties defined in `plain_prethree.asy` and `three_light.asy`:

```
struct light {
  real[][] diffuse;
  real[][] specular;
  pen background=nullpen; // Background color of the canvas.
  real specularfactor;
  triple[] position; // Only directional lights are currently implemented.
}

light Viewport=light(specularfactor=3,(0.25,-0.25,1));

light White=light(new pen[] {rgb(0.38,0.38,0.45),rgb(0.6,0.6,0.67),
```

```
                              rgb(0.5,0.5,0.57)},specularfactor=3,
  new triple[] {(-2,-1.5,-0.5),(2,1.1,-2.5),(-0.5,0,2)});

light Headlamp=light(gray(0.8),specular=gray(0.7),
                     specularfactor=3,dir(42,48));

currentlight=Headlamp;

light nolight;
```

The `currentlight.background` (or `background` member of the specified `light`) can be used to set the background color for 2D (or 3D) images. The default background is white for `HTML` images and transparent for all other formats. One can request a completely transparent background for 3D `WebGL` images with `currentlight.background=black+opacity(0.0);`

`render`

A function `render()` may be assigned to the optional `render` parameter allows one to pass specialized rendering options to the surface drawing routines, via arguments such as:

```
bool tessellate;   // use tessellated mesh to store straight patches
real margin;       // shrink amount for rendered OpenGL viewport, in bp.
bool partnames;    // assign part name indices to compound objects
bool defaultnames; // assign default names to unnamed objects
interaction interaction; // billboard interaction mode
```

along with the rendering parameters for the legacy PRC format described in `three.asy`.

Asymptote also supports image-based lighting with the setting `settings.ibl=true`. This uses pre-rendered EXR images from the directory specified by `-imageDir` (which defaults to `ibl`) or, for `WebGL` rendering, the URL specified by `-imageURL` (which defaults to `https://vectorgraphics.gitlab.io/asymptote/ibl`). If the requested environment image (specified by `-image`, which defaults to `snowyField`) is not found locally, Asymptote will automatically download it from the `-imageURL` directory into the `-imageDir` search path, unless the `-imageURL` has been changed from its default, in which case `-nosafe` must also be given. To disable this automatic download, set `-image=""`. Additional rendered images can be generated on an `NVIDIA` GPU using the `reflect` program in the `cudareflect` subdirectory of the `Asymptote` source directory. For HTML (`WebGL`) output, the EXR images and the `tinyexr` decoder are loaded by the browser at runtime from the `-imageURL` directory.

Sample Bezier surfaces are contained in the example files `BezierSurface.asy`, `teapot.asy`, `teapotIBL.asy`, and `parametricsurface.asy`.

The structure `render` contains specialized rendering options documented at the beginning of module `three`.

The example `elevation.asy` illustrates how to draw a surface with patch-dependent colors. The examples `sphericalharmonic.asy`, `vertexshading.asy` and `smoothelevation.asy` illustrate vertex-dependent colors, which are supported by `Asymptote`'s native `OpenGL`/`WebGL` renderers, the three-dimensional vector output format `V3D`, and two-dimensional vector output (`settings.render=0`). Since the legacy PRC output format does not support vertex shading of Bezier surfaces, PRC patches are shaded with the mean of the four vertex colors.

Vertex colors may be assigned at draw time using the `vertexPen` parameter of `draw(surface,...)`, for example to color a surface `s` according to its height:

```
draw(s,s.palette(zpart,Rainbow()));
```

Predefined palettes such as `Rainbow()` are listed in Section 8.4 [palette], page 168; further palettes are available in the module `colormap` (see Section 8.5 [colormap], page 176). Functions for coloring a surface by position, by parametric coordinates, by patch, or by corner are described in Section 8.4.1 [Surface coloring], page 173.

A surface can be constructed from a cyclic `path3` with the constructor

```
surface surface(path3 external, triple[] internal=new triple[],
                pen[] colors=new pen[], bool3 planar=default);
```

and then filled:

```
draw(surface(unitsquare3,new triple[] {X,Y,Z,O}),red);
draw(surface(O--X{Y}..Y{-X}--cycle,new triple[] {Z}),red);
draw(surface(path3(polygon(5))),red,nolight);
draw(surface(unitcircle3),red,nolight);
draw(surface(unitcircle3,new pen[] {red,green,blue,black}),nolight);
```

The first example draws a Bezier patch and the second example draws a Bezier triangle. The third and fourth examples are planar surfaces. The last example constructs a patch with vertex-specific colors. A three-dimensional planar surface in the plane `plane` can be constructed from a two-dimensional cyclic path `g` with the constructor

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

and then filled:

```
draw(surface((0,0)--E+2N--2E--E+N..0.2E..cycle),red);
```

Planar Bezier surfaces patches are constructed using Orest Shardt's `bezulate` routine, which decomposes (possibly nonsimply connected) regions bounded (according to the `zerowinding` fill rule) by simple cyclic paths (intersecting only at the endpoints) into subregions bounded by cyclic paths of length `4` or less.

A more efficient routine also exists for drawing tessellations composed of many 3D triangles, with specified vertices, and optional normals or vertex colors:

```
void draw(picture pic=currentpicture, triple[] v, int[][] vi,
          triple[] n={}, int[][] ni=vi, material m=currentpen, pen[] p={},
          int[][] pi=vi, light light=currentlight);
```

Here, the triple array `v` lists the (typically distinct) vertices, while the array `vi` contains integer arrays of length 3 containing the indices of the elements in `v` that form the vertices of each triangle. Similarly, the arguments `n` and `ni` contain optional normal data and `p` and `pi` contain optional pen vertex data. If more than one normal or pen is specified for a vertex, the last one is used. An example of this tessellation facility is given in `triangles.asy`.

Arbitrary thick three-dimensional curves and line caps (which the `OpenGL` standard does not require implementations to provide) are constructed with

```
tube tube(path3 p, real width, render render=defaultrender);
```

this returns a tube structure representing a tube of diameter `width` centered approximately on `g`. The tube structure consists of a surface `s` and the actual tube center, path3 `center`.

Drawing thick lines as tubes can be slow to render, especially with the `Adobe Reader` renderer. The setting `thick=false` can be used to disable this feature and force all lines to be drawn with `linewidth(0)` (one pixel wide, regardless of the resolution). By default, mesh and contour lines in three-dimensions are always drawn thin, unless an explicit line width is given in the pen parameter or the setting `thin` is set to `false`. The pens `thin()` and `thick()` defined in `plain_pens.asy` can also be used to override these defaults for specific draw commands.

There are six choices for viewing 3D `Asymptote` output:

1. Use the native `Asymptote` adaptive `OpenGL`-based renderer (with the command-line option `-V` and the default settings `outformat=""` and `render=-1`). On `UNIX` systems with graphics support for multisampling, the sample width can be controlled with the setting `multisample`. An initial screen position can be specified with the pair setting `position`, where negative values are interpreted as relative to the corresponding maximum screen dimension. The default settings

   ```
   import settings;
   leftbutton=new string[] {"rotate","zoom","shift","pan"};
   middlebutton=new string[] {""};
   rightbutton=new string[] {"zoom","rotateX","rotateY","rotateZ"};
   wheelup=new string[] {"zoomin"};
   wheeldown=new string[] {"zoomout"};
   ```

   bind the mouse buttons as follows:

   - Left: rotate
   - Shift Left: zoom
   - Ctrl Left: shift viewport
   - Alt Left: pan
   - Wheel Up: zoom in
   - Wheel Down: zoom out
   - Right: zoom
   - Shift Right: rotate about the X axis
   - Ctrl Right: rotate about the Y axis
   - Alt Right: rotate about the Z axis

   The keyboard bindings are:

   - h: home
   - f: toggle fitscreen
   - x: spin about the X axis
   - y: spin about the Y axis
   - z: spin about the Z axis
   - s: stop spinning
   - m: rendering mode (solid/patch/mesh)
   - e: export
   - c: show camera parameters

- p: play animation
- r: reverse animation
- : step animation
- +: expand
- =: expand
- >: expand
- -: shrink
- _: shrink
- <: shrink
- q: exit
- Ctrl-q: exit

2. Generate `WebGL` interactive vector graphics output with the the command-line option and `-f html` (or the setting `outformat="html"`). The resulting 3D HTML file can then be viewed directly in any modern desktop or mobile browser, or even embedded within another web page:

   ```
   <iframe src="logo3.html" width="561" height="321" frameborder="0">
   </iframe>
   ```

   Normally, `WebGL` files generated by `Asymptote` are dynamically remeshed to fit the browser window dimensions. However, the setting `absolute=true` can be used to force the image to be rendered at its designed size (accounting for multiple device pixels per `css` pixel).

   For specialized applications, the setting `keys=true` can be used to generate an identifying key immediately before the `WebGL` code for each generated object. The default key, the `"line:column"` of the associated function call of the top-level source code, can be overwritten by adding `KEY="x"` as the first argument of the function call, where `x` represents user-supplied text.

   The interactive `WebGL` files produced by `Asymptote` use the default mouse and (many of the same) key bindings as the `OpenGL` renderer. Zooming via the mouse wheel of a `WebGL` image embedded within another page is disabled until the image is activated by a click or touch event and will remain enabled until the `ESC` key is pressed.

   By default, viewing the 3D HTML files generated by Asymptote requires network access to download the `AsyGL` rendering library, which is normally cached by the browser for future use. However, the setting `offline=true` can be used to embed this small (about 48kB) library within a stand-alone HTML file that can be viewed offline.

3. Render the scene to a specified rasterized format `outformat` at the resolution of `n` pixels per `bp`, as specified by the setting `render=n`. The default value of `render` is 2. By default, the scene is internally rendered at twice the specified resolution; this can be disabled by setting `antialias=1`. High resolution rendering is done by tiling the image. If your graphics card allows it, the rendering can be made more efficient by increasing the maximum tile size `maxtile` to your screen dimensions (indicated by `maxtile=(0,0)`. If your video card generates unwanted black stripes in the output, try setting the horizontal and vertical components of `maxtiles` to something less than your screen dimensions. The tile size is also limited by the setting `maxviewport`,

which restricts the maximum width and height of the viewport. Some graphics drivers support batch mode (`-noV`) rendering in an iconified window; this can be enabled with the setting `iconify=true`.

4. Embed the 3D legacy PRC format in a PDF file and view the resulting PDF file with version `9.0` or later of `Adobe Reader`. This requires `settings.outformat="pdf"` and `settings.prc=true`, which can be specified by the command-line options `-f pdf` and `-f prc`, put in the `Asymptote` configuration file (see [configuration file], page 205), or specified in the script before module `three` (or `graph3`) is imported. The `media9` LaTeX package is also required (see Section 9.10 [embed], page 195). The example `100d.asy` illustrates how one can generate a list of predefined views (see `100d.views`). A stationary preview image with a resolution of `n` pixels per `bp` can be embedded with the setting `render=n`; this allows the file to be viewed with other `PDF` viewers. Alternatively, the file `externalprc.tex` illustrates how the resulting PRC and rendered image files can be extracted and processed in a separate LaTeX file. However, see Chapter 7 [LaTeX usage], page 112, for an easier way to embed three-dimensional `Asymptote` pictures within LaTeX. For specialized applications where only the raw PRC file is required, specify `settings.outformat="prc"`. The PRC specification is available from `https://web.archive.org/web/20081204104459/http://livedocs.adobe.com/acrobat_sdk/9/Acrobat9_HTMLHelp/API_References/PRCReference/PRC_Format_Specification/`
5. Output a V3D portable compressed vector graphics file using `settings.outformat="v3d"`, which can be viewed with an external viewer or converted to an alternate 3D format using the Python `pyv3d` library. V3D content can be automatically embedded within a PDF file using the options `settings.outformat="pdf"` and `settings.v3d=true`. Alternatively, a V3D file `file.v3d` may be manually embedded within a PDF file using the `media9` LaTeX package:

   ```
   \includemedia[noplaybutton,width=100pt,height=200pt]{}{file.v3d}%
   ```

   An online `Javascript`-based V3D-aware `PDF` viewer is available at `https://github.com/vectorgraphics/pdfv3dReader`. An `Okular` plugin for interactive viewing of V3D files and V3D-embedded content in PDF files is available at `https://github.com/vectorgraphics/Okular-v3d-Plugins`.

   The V3D specification and the `pyv3d` library are available at `https://github.com/vectorgraphics/v3d`. A V3D file `file.v3d` may be imported and viewed by `Asymptote` either by specifying `file.v3d` on the command line

   ```
   asy -V file.v3d
   ```

   or using the `v3d` module and `importv3d` function in interactive mode (or within an `Asymptote` file):

   ```
   import v3d;
   importv3d("file.v3d");
   ```
6. Project the scene to a two-dimensional vector (EPS or PDF) format with `render=0`. Only limited support for hidden surface removal, lighting, and transparency is available with this approach (see [PostScript3D], page 160).

Automatic picture sizing in three dimensions is accomplished with double deferred drawing. The maximal desired dimensions of the scene in each of the three dimensions can optionally be specified with the routine

```
void size3(picture pic=currentpicture, real x, real y=x, real z=y,
           bool keepAspect=pic.keepAspect);
```

A simplex linear programming problem is then solved to produce a 3D version of a frame (actually implemented as a 3D picture). The result is then fit with another application of deferred drawing to the viewport dimensions corresponding to the usual two-dimensional picture `size` parameters. The global pair `viewportmargin` may be used to add horizontal and vertical margins to the viewport dimensions. Alternatively, a minimum `viewportsize` may be specified. A 3D picture `pic` can be explicitly fit to a 3D frame by calling

```
frame pic.fit3(projection P=currentprojection);
```

and then added to picture `dest` about `position` with

```
void add(picture dest=currentpicture, frame src, triple position=(0,0,0));
```

For convenience, the `three` module defines `O=(0,0,0)`, `X=(1,0,0)`, `Y=(0,1,0)`, and `Z=(0,0,1)`, along with a unitcircle in the XY plane:

```
path3 unitcircle3=X..Y..-X..-Y..cycle;
```

A general (approximate) 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 166, for an example of a three-dimensional circular arc.

The representation `O--O+u--O+u+v--O+v--cycle` of the plane passing through point `O` with normal `cross(u,v)` is returned 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 box is predefined as

```
path3[] unitbox=box(O,(1,1,1));
```

`Asymptote` also provides optimized definitions for the three-dimensional paths `unitsquare3` and `unitcircle3`, along with the surfaces `unitdisk`, `unitplane`,

`unitcube`, `unitcylinder`, `unitcone`, `unitsolidcone`, `unitfrustum(real t1, real t2)`, `unitsphere`, and `unithemisphere`.

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, triple target=0, real zoom=1, pair viewportshift=0, bool showtarget=true, bool center=true)`
: 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. The bounding volume is expanded to include `target` if `showtarget=true`. If `center=true`, the target will be adjusted to the center of the bounding volume.

`orthographic(real x, real y, real z, triple up=Z, triple target=0, real zoom=1, pair viewportshift=0, bool showtarget=true, bool center=true)`
: This is equivalent to

  ```
  orthographic((x,y,z),up,target,zoom,viewportshift,showtarget,center)
  ```

  The routine

  ```
  triple camera(real alpha, real beta);
  ```

  can be used to compute the camera position with the x axis below the horizontal at angle `alpha`, the y axis below the horizontal at angle `beta`, and the z axis up.

`perspective(triple camera, triple up=Z, triple target=0, real zoom=1, real angle=0, pair viewportshift=0, bool showtarget=true, bool autoadjust=true, bool center=autoadjust)`
: 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. If `autoadjust=true`, the camera will automatically be adjusted to lie outside the bounding volume for all possible

interactive rotations about `target`. If `center=true`, the target will be adjusted to the center of the bounding volume.

```
perspective(real x, real y, real z, triple up=Z, triple target=0,
real zoom=1, real angle=0, pair viewportshift=0,             bool
showtarget=true, bool autoadjust=true,             bool center=autoadjust)
```

This is equivalent to

```
perspective((x,y,z),up,target,zoom,angle,viewportshift,showtarget,
            autoadjust,center)
```

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

We also define standard orthographic views used in technical drawing:

```
projection LeftView=orthographic(-X,showtarget=true);
projection RightView=orthographic(X,showtarget=true);
projection FrontView=orthographic(-Y,showtarget=true);
projection BackView=orthographic(Y,showtarget=true);
projection BottomView=orthographic(-Z,showtarget=true);
projection TopView=orthographic(Z,showtarget=true);
```

The function

```
void addViews(picture dest=currentpicture, picture src,
              projection[][] views=SixViewsUS,
              bool group=true, filltype filltype=NoFill);
```

adds to picture `dest` an array of views of picture `src` using the layout projection[][] `views`. The default layout `SixViewsUS` aligns the projection `FrontView` below `TopView` and above `BottomView`, to the right of `LeftView` and left of `RightView` and `BackView`. The predefined layouts are:

```
projection[][] ThreeViewsUS={{TopView},
                             {FrontView,RightView}};

projection[][] SixViewsUS={{null,TopView},
                           {LeftView,FrontView,RightView,BackView},
                           {null,BottomView}};

projection[][] ThreeViewsFR={{RightView,FrontView},
                             {null,TopView}};

projection[][] SixViewsFR={{null,BottomView},
                           {RightView,FrontView,LeftView,BackView},
                           {null,TopView}};

projection[][] ThreeViews={{FrontView,TopView,RightView}};

projection[][] SixViews={{FrontView,TopView,RightView},
                         {BackView,BottomView,LeftView}};
```

A triple or path3 can be projected to a pair or path, with `project(triple, projection P=currentprojection)` or `project(path3, 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);
```

A pair `z` on the projection plane can be inverted to a triple with the routine

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

A pair direction `dir` on the projection plane can be inverted to a triple direction relative to a point `v` with the routine

```
triple invert(pair dir, triple v, projection P=currentprojection).
```

Three-dimensional objects may be transformed with one of the following built-in transform3 types (the identity transformation is `identity4`):

`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;

`scale(real x, real y, real z)`
: scales by `x` in the x direction, by `y` in the y direction, and by `z` in the z direction;

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

`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`.

When not multiplied on the left by a transform3, three-dimensional TEX Labels are drawn as Bezier surfaces directly on the projection plane:

```
void label(picture pic=currentpicture, Label L, triple position,
           align align=NoAlign, pen p=currentpen,
           light light=nolight, string name="",
           render render=defaultrender, interaction interaction=
           settings.autobillboard ? Billboard : Embedded)
```

The optional `name` parameter is used as a prefix for naming the label patches in the PRC model tree. The default interaction is `Billboard`, which means that labels are rotated interactively so that they always face the camera. The interaction `Embedded` means that the label interacts as a normal `3D` surface, as illustrated in the example `billboard.asy`.

Alternatively, a label can be transformed from the XY plane by an explicit transform3 or mapped to a specified two-dimensional plane with the predefined transform3 types XY, YZ, ZX, YX, ZY, ZX. There are also modified versions of these transforms that take an optional argument projection P=currentprojection that rotate and/or flip the label so that it is more readable from the initial viewpoint.

A transform3 that projects in the direction dir onto the plane with normal n through point O is returned by

```
transform3 planeproject(triple n, triple O=O, triple dir=n);
```

One can use

```
triple normal(path3 p);
```

to find the unit normal vector to a planar three-dimensional path p. As illustrated in the example planeproject.asy, a transform3 that projects in the direction dir onto the plane defined by a planar path p is returned by

```
transform3 planeproject(path3 p, triple dir=normal(p));
```

The functions

```
surface extrude(path p, triple axis=Z);
surface extrude(Label L, triple axis=Z);
```

return the surface obtained by extruding path p or Label L along axis.

Three-dimensional versions of the path functions length, size, point, dir, accel, radius, precontrol, postcontrol, arclength, arctime, reverse, subpath, intersect, intersections, intersectionpoint, intersectionpoints, min, max, cyclic, and straight are also defined.

The routine

```
real[] intersect(path3 p, surface s, real fuzz=-1);
```

returns a real array of length 3 containing the intersection times, if any, of a path p with a surface s. The routine

```
real[][] intersections(path3 p, surface s, real fuzz=-1);
```

returns all (unless there are infinitely many) intersection times of a path p with a surface s as a sorted array of real arrays of length 3, and

```
triple[] intersectionpoints(path3 p, surface s, real fuzz=-1);
```

returns the corresponding intersection points. Here, the computations are performed to the absolute error specified by fuzz, or if fuzz < 0, to machine precision. The routine

```
real orient(triple a, triple b, triple c, triple d);
```

is a numerically robust computation of dot(cross(a-d,b-d),c-d), which is the determinant

```
|a.x a.y a.z 1|
|b.x b.y b.z 1|
|c.x c.y c.z 1|
|d.x d.y d.z 1|
```

The result is negative (positive) if a, b, c appear in counterclockwise (clockwise) order when viewed from d or zero if all four points are coplanar.

The routine

```
real insphere(triple a, triple b, triple c, triple d, triple e);
```

returns a positive (negative) value if `e` lies inside (outside) the sphere passing through points `a,b,c,d` oriented so that `dot(cross(a-d,b-d),c-d)` is positive, or zero if all five points are cospherical. The value returned is the determinant

```
|a.x a.y a.z a.x^2+a.y^2+a.z^2 1|
|b.x b.y b.z b.x^2+b.y^2+b.z^2 1|
|c.x c.y c.z c.x^2+c.y^2+c.z^2 1|
|d.x d.y d.z d.x^2+d.y^2+d.z^2 1|
|e.x e.y e.z e.x^2+e.y^2+e.z^2 1|
```

Here is an example showing all five guide3 connectors:

```
import graph3;

size(200);

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

triple[] z=new triple[10];

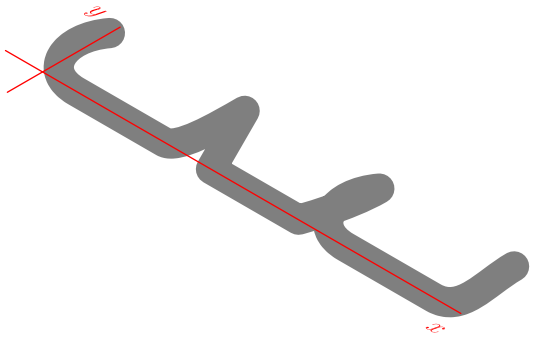
z[0]=(0,100,0); z[1]=(50,0,0); z[2]=(180,0,0);

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),currentlight);

xaxis3(Label(XY()*"$x$",align=-3Y),red,above=true);
yaxis3(Label(XY()*"$y$",align=-3X),red,above=true);
```



Three-dimensional versions of bars or arrows can be drawn with one of the specifiers `None`, `Blank`, `BeginBar3`, `EndBar3` (or equivalently `Bar3`), `Bars3`, `BeginArrow3`, `MidArrow3`,

EndArrow3 (or equivalently Arrow3), Arrows3, BeginArcArrow3, EndArcArrow3 (or equivalently ArcArrow3), MidArcArrow3, and ArcArrows3. Three-dimensional bars accept the optional arguments (real size=0, triple dir=0). If size=0, the default bar length is used; if dir=0, the bar is drawn perpendicular to the path and the initial viewing direction. The predefined three-dimensional arrowhead styles are DefaultHead3, HookHead3, TeXHead3. Versions of the two-dimensional arrowheads lifted to three-dimensional space and aligned according to the initial viewpoint (or an optionally specified normal vector) are also defined: DefaultHead2(triple normal=0), HookHead2(triple normal=0), TeXHead2(triple normal=0). These are illustrated in the example arrows3.asy.

Module three also defines the three-dimensional margins NoMargin3, BeginMargin3, EndMargin3, Margin3, Margins3, BeginPenMargin2, EndPenMargin2, PenMargin2, PenMargins2, BeginPenMargin3, EndPenMargin3, PenMargin3, PenMargins3, BeginDotMargin3, EndDotMargin3, DotMargin3, DotMargins3, Margin3, and TrueMargin3.

The routine

```
void pixel(picture pic=currentpicture, triple v, pen p=currentpen,
           real width=1);
```

can be used to draw on picture pic a pixel of width width at position v using pen p.

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

Limited support for projected vector graphics (effectively three-dimensional nonrendered PostScript) is available with the setting render=0. This currently only works for piecewise planar surfaces, such as those produced by the parametric surface routines in the graph3 module. Surfaces produced by the solids module will also be properly rendered if the parameter nslices is sufficiently large.

In the module bsp, hidden surface removal of planar pictures is implemented using 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 two-dimensional 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);
```

Labels may be projected to two dimensions, using projection P, onto the plane passing through point O with normal cross(u,v) by multiplying it on the left by the transform

```
transform transform(triple u, triple v, triple O=O,
                    projection P=currentprojection);
```

Here is an example that shows how a binary space partition may be used to draw a two-dimensional vector graphics projection of three orthogonal intersecting planes:

```
size(6cm,0);
import bsp;

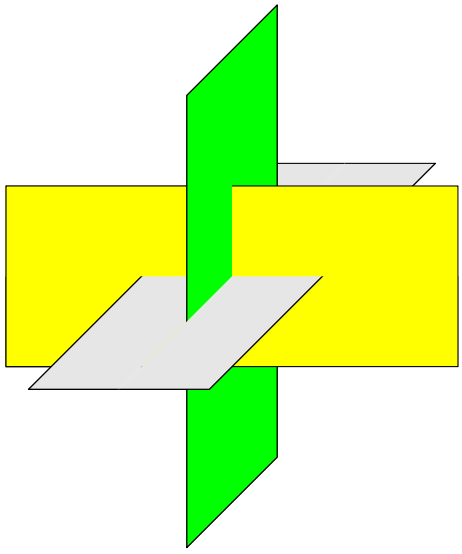
real u=2.5;
real v=1;
```

```
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),project(y),yellow);
filldraw(faces.push(l),project(l),lightgrey);
filldraw(faces.push(g),project(g),green);

add(faces);
```



8.3 graph3

This module implements three-dimensional versions of the functions in `graph.asy`. To draw an x axis in three dimensions, use the routine

```
void xaxis3(picture pic=currentpicture, Label L="", axis axis=YZZero,
            real xmin=-infinity, real xmax=infinity, pen p=currentpen,
            ticks3 ticks=NoTicks3, arrowbar3 arrow=None,
            margin3 margin=NoMargin3, bool above=false,
            projection P=currentprojection);
```

Analogous routines `yaxis` and `zaxis` can be used to draw y and z axes in three dimensions. There is also a routine for drawing all three axis:

```
void axes3(picture pic=currentpicture,
           Label xlabel="", Label ylabel="", Label zlabel="",
           bool extend=false,
           triple min=(-infinity,-infinity,-infinity),
```

```
            triple max=(infinity,infinity,infinity),
            pen p=currentpen, pen py=p, pen pz=p,
            arrowbar3 arrow=None, margin3 margin=NoMargin3,
            projection P=currentprojection);
```

The predefined three-dimensional axis types are

```
axis YZEquals(real y, real z, triple align=0, bool extend=false);
axis XZEquals(real x, real z, triple align=0, bool extend=false);
axis XYEquals(real x, real y, triple align=0, bool extend=false);
axis YZZero(triple align=0, bool extend=false);
axis XZZero(triple align=0, bool extend=false);
axis XYZero(triple align=0, bool extend=false);
axis Bounds(int type=Both, int type2=Both, triple align=0, bool extend=false);
```

The optional `align` parameter to these routines can be used to specify the default axis and tick label alignments. The `Bounds` axis accepts two type parameters, each of which must be one of `Min`, `Max`, or `Both`. These parameters specify which of the four possible three-dimensional bounding box edges should be drawn.

The three-dimensional tick options are `NoTicks3`, `InTicks`, `OutTicks`, and `InOutTicks`. These specify the tick directions for the `Bounds` axis type; other axis types inherit the direction that would be used for the `Bounds(Min,Min)` axis.

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

```
import graph3;

size(0,200);
size3(200,IgnoreAspect);

currentprojection=orthographic(4,6,3);

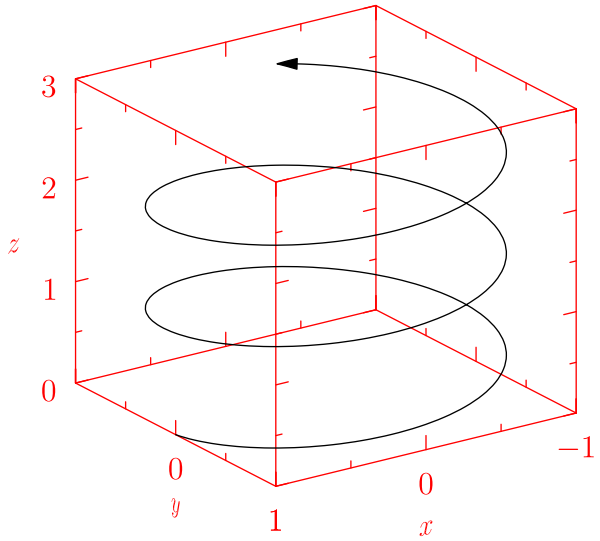
real x(real t) {return cos(2pi*t);}
real y(real t) {return sin(2pi*t);}
real z(real t) {return t;}

path3 p=graph(x,y,z,0,2.7,operator ..);

draw(p,Arrow3);

scale(true);

xaxis3(XZ()*"$x$",Bounds,red,InTicks(Label,2,2));
yaxis3(YZ()*"$y$",Bounds,red,InTicks(beginlabel=false,Label,2,2));
zaxis3(XZ()*"$z$",Bounds,red,InTicks);
```



The next example illustrates three-dimensional x, y, and z axes, without autoscaling of the axis limits:

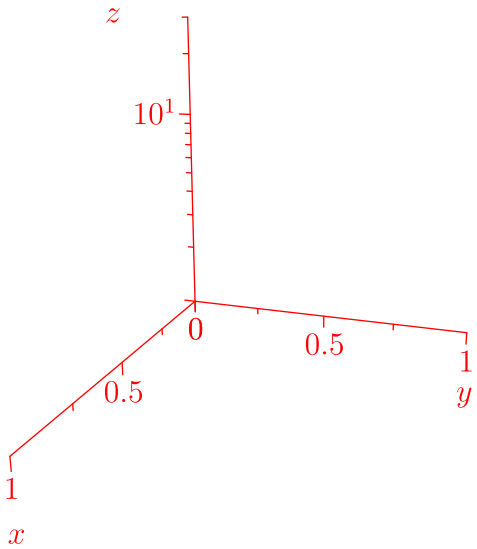
```
import graph3;

size(0,200);
size3(200,IgnoreAspect);

currentprojection=perspective(dir(75,20));

scale(Linear,Linear,Log);

xaxis3("$x$",0,1,red,OutTicks(2,2));
yaxis3("$y$",0,1,red,OutTicks(2,2));
zaxis3("$z$",1,30,red,OutTicks(beginlabel=false));
```



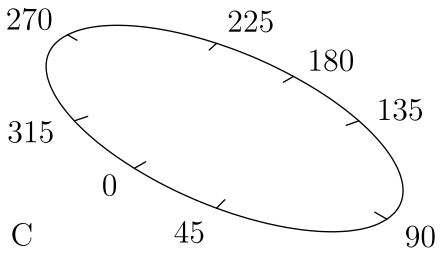
One can also place ticks along a general three-dimensional axis:

```
import graph3;

size(0,100);

path3 g=yscale3(2)*unitcircle3;
currentprojection=perspective(10,10,10);

axis(Label("C",position=0,align=15X),g,InTicks(endlabel=false,8,end=false),
     ticklocate(0,360,new real(real v) {
         path3 h=O--max(abs(max(g)),abs(min(g)))*dir(90,v);
         return intersect(g,h)[0];},
       new triple(real t) {return cross(dir(g,t),Z);}));
```



Surface plots of matrices and functions over the region `box(a,b)` in the XY plane are also implemented. A surface can be constructed from a matrix `f` of heights, at evenly spaced points over `box(a,b)` or at the points `(x[i],y[j])`, or from a matrix of points:

```
surface surface(real[][] f, pair a, pair b, bool[][] cond={});
surface surface(real[][] f, pair a, pair b, splinetype xsplinetype,
                splinetype ysplinetype=xsplinetype, bool[][] cond={});
surface surface(real[][] f, real[] x, real[] y,
                splinetype xsplinetype=Spline,
                splinetype ysplinetype=xsplinetype,
                bool[][] cond={});
surface surface(triple[][] f, bool[][] cond={});
```

The graph of a real function $f(x,y)$ over `box(a,b)` is constructed with

```
surface surface(real f(pair z), pair a, pair b, int nx=nmesh, int ny=nx,
                bool cond(pair z)=null);
surface surface(real f(pair z), pair a, pair b, int nx=nmesh, int ny=nx,
                splinetype xsplinetype, splinetype ysplinetype=xsplinetype,
                bool cond(pair z)=null);
```

On a logarithmic axis (see [automatic scaling], page 134), the points at which `f` is sampled are evenly spaced along the axis, as are the entries of a matrix over `box(a,b)`.

A parametric surface for a function $f(u,v)$ over the parameter space `box(a,b)`, as illustrated in the example `parametricsurface.asy`, is constructed with

```
surface surface(triple f(pair z), pair a, pair b, int nu=nmesh, int nv=nu,
```

```
                bool cond(pair z)=null);
surface surface(triple f(pair z), pair a, pair b, int nu=nmesh, int nv=nu,
                splinetype[] usplinetype, splinetype[] vsplinetype=Spline,
                bool cond(pair z)=null);
```

The parameters, in contrast, are sampled at evenly spaced values over `box(a,b)` even in a picture with logarithmic axes. To sample them at other values, specify the values in the arrays `u` and `v`:

```
surface surface(triple f(pair z), real[] u, real[] v,
                bool cond(pair z)=null);
surface surface(triple f(pair z), real[] u, real[] v,
                splinetype[] usplinetype, splinetype[] vsplinetype=Spline,
                bool cond(pair z)=null);
```

A `surface` constructor with no splinetype parameter interpolates linearly; the others interpolate with the splinetypes given, such as `Spline` for a smooth surface. The boolean array or function `cond` can be used to control which surface mesh cells are actually drawn (by default all mesh cells are drawn). A surface constructed with these routines may be colored at draw time as a function of position, parametric coordinates, or mesh cell (see Section 8.4.1 [Surface coloring], page 173); the parametric coordinates of such a surface are described in [Surface parametrization], page 165.

One can also construct the surface generated by rotating a path `g` between `angle1` to `angle2` (in degrees) sampled `n` times about the line `c--c+axis`:

```
surface surface(triple c, path3 g, triple axis, int n=nslice,
                real angle1=0, real angle2=360, pen color(int i, real j)=null);
```

The optional argument `color(int i, real j)` can be used to override the surface color at the point obtained by rotating vertex `i` by angle `j` (in degrees).

A surface constructed with any of the above routines is *structured*: its patches form a rectangular array of `nu` by `nv` mesh cells, and each point of the surface has parametric coordinates (u, v). A surface assembled in any other way, from paths or from individual patches, is *unstructured* and has no parametric coordinates. Unstructured surfaces include `unitsphere` and the other predefined surfaces of module `three`, as well as the implicit surfaces produced by the modules `contour3` and `smoothcontour3`.

The point and a normal vector (not normalized) of a structured surface `s` at the parametric coordinates (u, v) are returned by

```
triple s.paramPoint(real u, real v);
triple s.paramNormal(real u, real v);
```

These functions abort with an error if `s` is unstructured; calling one of them is therefore a simple test of whether a given surface is structured. A structured surface remains structured, with the same parametric coordinates, when a `transform3` is applied to it.

The domain of the parametric coordinates depends on how the surface was constructed:

- for a surface constructed from a parametric function `triple f(pair z)` over `box(a,b)`, whether or not a splinetype is specified, (u, v) is the argument `z` of `f`: u ranges from `a.x` to `b.x` and v ranges from `a.y` to `b.y`;
- for the graph of a real function `real f(pair z)` or of a real matrix `f` over `box(a,b)`, (u, v) are the x and y coordinates of the point of the surface, so that they also range over

`box(a,b)`. On a logarithmic axis (see [automatic scaling], page 134) the coordinate of a point is the logarithm of the value plotted there, and this is the parametric coordinate as well: for x plotted from 1 to 100, u ranges from 0 to 2;

- for a surface constructed from any other matrix or arrays, or from a function sampled at the values in the arrays `u` and `v`, the parametric coordinates are mesh coordinates: u ranges from `0` to `nu` and v ranges from `0` to `nv`, where `nu` and `nv` are the number of mesh cells in each direction (one less than the number of samples). The sample with indices `i` and `j` has parametric coordinates (i, j);
- for a surface of revolution, u is the angle of rotation measured in radians (not degrees), ranging from `radians(angle1)` to `radians(angle2)`, and v is the time along the path `g`, ranging from `0` to `length(g)`.

A structured surface can be given a different domain with

```
void s.domain(pair a, pair b);
```

after which the parametric coordinates range over `box(a,b)`, with the mesh taken to be uniform: the sample with indices `i` and `j` then has the parametric coordinates returned by

```
pair s.paramCoords(real i, real j);
```

For example, if the entries of a matrix of triples are the values of a function on an evenly spaced grid over the unit square, the surface constructed from the matrix can be given the coordinates of that function with `s.domain((0,0),(1,1))`.

Parametric coordinates must normally lie within this domain. The exception is a surface that closes on itself: if `s.ucyclic()` is `true`, then u may be any real number and is reduced modulo the length of its domain, and likewise for v if `s.vcyclic()` is `true`. Whether a surface is cyclic in a given direction depends on how it was constructed:

- a surface of revolution is cyclic in u if `angle2-angle1` is a multiple of 360 degrees (as it is by default). It is cyclic in v if the path `g` is cyclic;
- a surface constructed from a triple-valued function or matrix, with no splinetype specified, is cyclic in a direction if the samples along the two opposite edges of the mesh coincide;
- a surface constructed from a triple-valued function with a splinetype specified must meet two requirements to be cyclic in a direction. First, the samples along the two opposite edges of the mesh must coincide. Second, the splinetype in that direction must be periodic. The splinetype `Spline` chooses a periodic splinetype by itself, but only if the samples also approach the two edges with nearly the same slope, so that the surface closes without a kink;
- a surface constructed from a real-valued function or matrix is cyclic in a direction only if a splinetype is specified and the splinetype in that direction is `periodic`.

It is an error to evaluate the surface outside its domain in a direction in which it is not cyclic, or at a point belonging only to mesh cells that were omitted by `cond`.

Surface lighting is illustrated in the example files `parametricsurface.asy` and `sinc.asy`. Lighting can be disabled by setting `light=nolight`, as in this example of a Gaussian surface:

```
import graph3;
```

```
size(200,0);

currentprojection=perspective(10,8,4);

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);

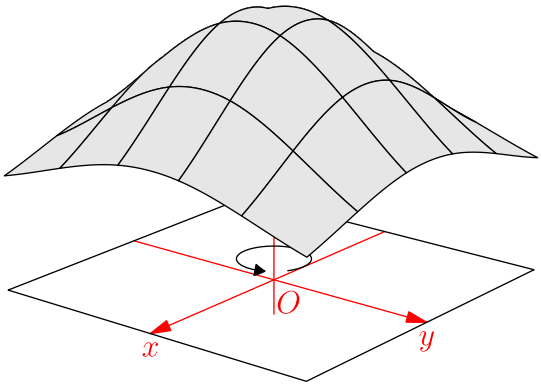
draw(arc(0.12Z,0.2,90,60,90,25),ArcArrow3);

surface s=surface(f,(-1,-1),(1,1),nx=5,Spline);

xaxis3(Label("$x$"),red,Arrow3);
yaxis3(Label("$y$"),red,Arrow3);
zaxis3(XYZero(extend=true),red,Arrow3);

draw(s,lightgray,meshpen=black+thick(),nolight,render(merge=true));

label("$O$",O,-Z+Y,red);
```



A mesh can be drawn without surface filling by specifying `nullpen` for the surfacepen.

A vector field of `nu`×`nv` arrows on a parametric surface `f` over `box(a,b)` can be drawn with the routine

```
picture vectorfield(path3 vector(pair v), triple f(pair z), pair a, pair b,
                    int nu=nmesh, int nv=nu, bool truesize=false,
                    real maxlength=truesize ? 0 : maxlength(f,a,b,nu,nv),
                    bool cond(pair z)=null, pen p=currentpen,
                    arrowbar3 arrow=Arrow3, margin3 margin=PenMargin3)
```

as illustrated in the examples `vectorfield3.asy` and `vectorfieldsphere.asy`.

8.4 palette

`Asymptote` can also generate color density images and palettes. For the use of palettes to color three-dimensional surfaces, see Section 8.4.1 [Surface coloring], page 173. The following palettes are predefined in `palette.asy`:

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

`pen[] Rainbow(int NColors=32766)`
: a rainbow spectrum;

`pen[] BWRainbow(int NColors=32761)`
: a rainbow spectrum tapering off to black/white at the ends;

`pen[] BWRainbow2(int NColors=32761)`
: a double rainbow palette tapering off to black/white at the ends, with a linearly scaled intensity.

`pen[] Wheel(int NColors=32766)`
: a full color wheel palette;

`pen[] Gradient(int NColors=256, ... pen[] p)`
: a palette varying linearly over the specified array of pens, using NColors in each interpolation interval;

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, int divs=0,
             bool antialias=false)
```

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 134). If `divs` > 1, the palette is quantized to `divs` $- 1$ values. 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)`. (The type `range` is defined in `palette.asy`.) 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 examples `fillcontour.asy` and `imagecontour.asy` illustrate how level sets (contour lines) can be drawn on a color density plot (see Section 8.9 [contour], page 177).

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, int divs=0,
             bool transpose=(initial.x < final.x && initial.y < final.y),
             bool copy=true, bool antialias=false);
```

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,
             bool copy=true, bool antialias=false);
```

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 119) 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;

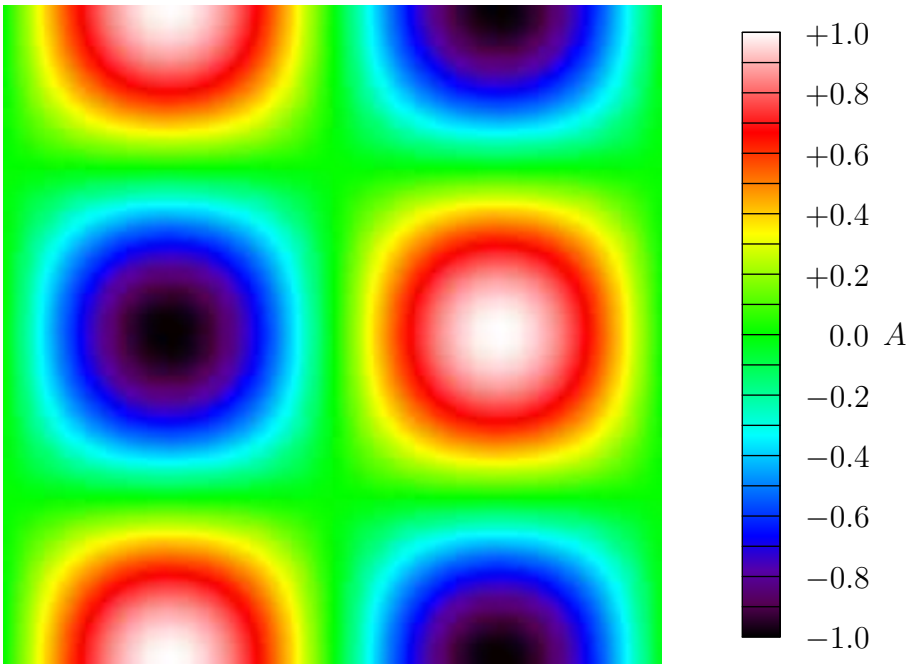
int n=256;
real ninv=2pi/n;
real[][] v=new real[n][n];

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)));
}

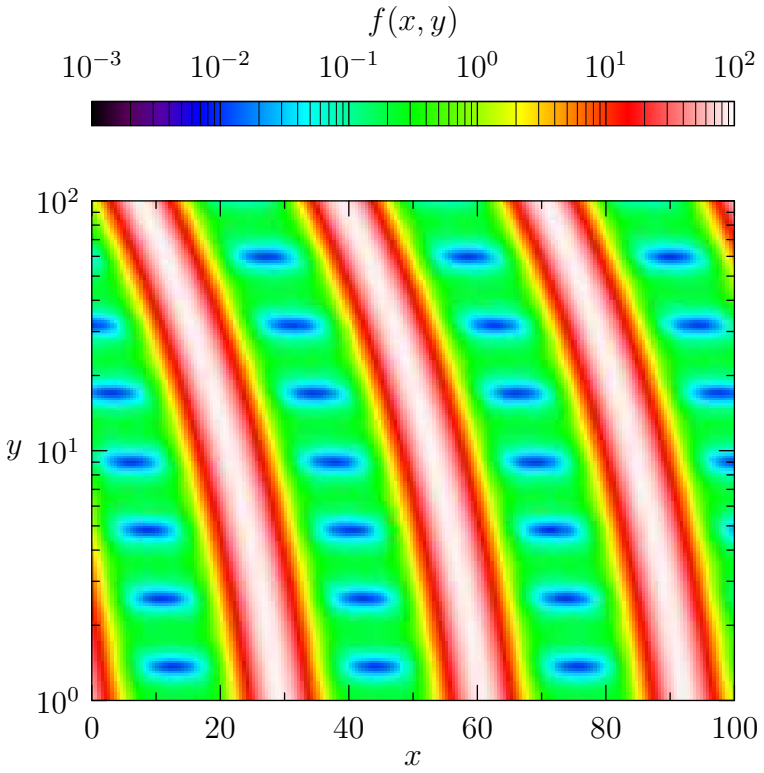
scale(Linear,Log,Log);

pen[] Palette=BWRainbow();

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

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

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 or a function `pen f(int, int)`:

```
void image(picture pic=currentpicture, pen[][] data,
           pair initial, pair final,
           bool transpose=(initial.x < final.x && initial.y < final.y),
           bool copy=true, bool antialias=false);
void image(picture pic=currentpicture, pen f(int, int), int width, int height,
           pair initial, pair final,
           bool transpose=(initial.x < final.x && initial.y < final.y),
           bool antialias=false);
```

as illustrated in the following examples:

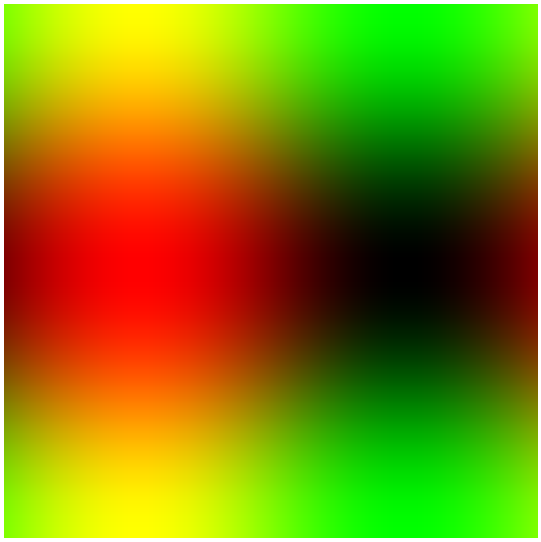
```
size(200);

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);

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



```
import palette;

size(200);

real fracpart(real x) {return (x-floor(x));}

pair pws(pair z) {
  pair w=(z+exp(pi*I/5)/0.9)/(1+z/0.9*exp(-pi*I/5));
  return exp(w)*(w^3-0.5*I);
}

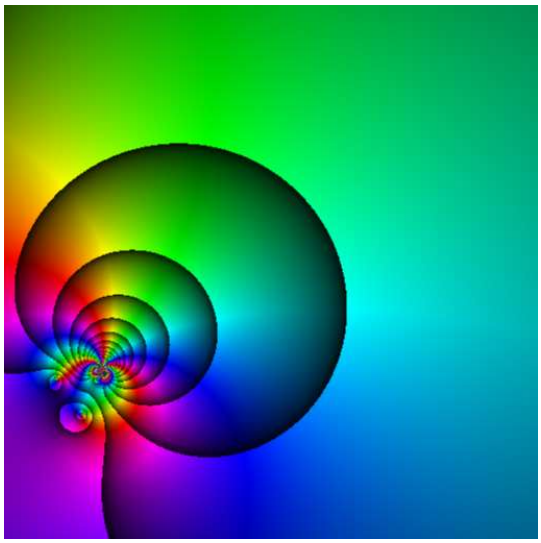
int N=512;

pair a=(-1,-1);
pair b=(0.5,0.5);
real dx=(b-a).x/N;
real dy=(b-a).y/N;

pen f(int u, int v) {
```

```
  pair z=a+(u*dx,v*dy);
  pair w=pws(z);
  real phase=degrees(w,warn=false);
  real modulus=w == 0 ? 0: fracpart(log(abs(w)));
  return hsv(phase,1,sqrt(modulus));
}

image(f,N,N,(0,0),(300,300),antialias=true);
```



For convenience, the module `palette` also defines functions that may be used to construct a pen array from a given function and palette:

```
pen[] palette(real[] f, pen[] palette);
pen[][] palette(real[][] f, pen[] palette);
```

8.4.1 Surface coloring

A surface is colored at draw time by passing a function to the `vertexPen` parameter of `draw(surface,...)`. The function is called once for each patch corner and returns the pen to be used there. Its type, also called `vertexPen`, is defined in the `palette` module as an abbreviation for a function type:

```
using vertexPen=pen(surfaceVertex);
```

The structure `surfaceVertex` provides the following information about the patch corner:

```
struct surfaceVertex {
  triple z;      // vertex position in user coordinates
  pair uv;       // parametric coordinates (structured surfaces only)
  int patch;     // index of patch (see below)
  int corner;    // index of corner in patch (0-3 or 0-2)
  int U, V;      // mesh cell of patch (structured surfaces only)
}
```

The fields `uv`, `U`, and `V` are meaningful only for structured surfaces (see [Surface parametrization], page 165); for unstructured surfaces (such as `unitsphere`) they are zero. A simple pen function of a single `triple` argument is automatically converted to a `vertexPen`, so the following are equivalent:

```
draw(s, vertexPen=new pen(triple v) {return hsv(360*zpart(v),1,1);});
draw(s, vertexPen=new pen(surfaceVertex sv) {
  return hsv(360*zpart(sv.z),1,1);
});
```

Color is thus a property of how a surface is drawn, rather than of the surface itself: drawing does not modify the surface, and the same surface can be drawn again with a different `vertexPen`. The vertex colors replace the color of `surfacepen`[1]: if `surfacepen` is `red`, the surface is no longer red. They also replace any colors stored in the surface when it was constructed. Other attributes of a `material`, such as its shininess, still apply. When the colors convey data, it is often preferable to show them exactly as specified, by turning the lighting off:

```
draw(s,s.palette(zpart,Rainbow()),nolight);
```

The convenience function `cornerPen` (defined in `three_surface.asy`) constructs a `vertexPen` that assigns a pen to each corner of each patch, using `sv.patch` and `sv.corner` for indexing:

```
vertexPen cornerPen(...pen[] p);
vertexPen cornerPen(pen[][] p);
```

The first form applies the same list of pens to every patch; it must supply a pen for every corner (four for a Bezier patch, three for a Bezier triangle). In the second form, corner `j` of patch `i` is drawn with pen `j` of the row `p[i % p.length]`. If that row is a cyclic array, `j` is likewise reduced modulo the length of the row; otherwise, it is an error for a patch to have more corners than the row has pens:

```
draw(s,cornerPen(red,green,blue,black));   // four pens per patch

pen[] p={red,blue};
p.cyclic=true;
draw(s,cornerPen(new pen[][] {p}));        // red,blue,red,blue
```

Corner `j` of a patch is node `j` of the boundary path of the patch, as returned by the patch method `external()`. For a surface constructed from a single cyclic path of length at most 4, this is node `j` of that path. The array `p` is copied, so subsequent changes to it do not affect the surface colors.

For a structured surface, `sv.uv` holds the parametric coordinates of the vertex, with the domain described in [Surface parametrization], page 165. Since these vary continuously from one patch to the next, a pen that depends on `sv.uv` colors the surface smoothly:

```
draw(s, vertexPen=new pen(surfaceVertex sv) {
  return hsv(360*sv.uv.x/(2pi), 1, 1);
});
```

[1] Strictly, its diffuse color: a pen used as a `surfacepen` becomes the diffuse pen of a `material` (see [material], page 148).

Both limits of the domain occur as values of `sv.uv`, even in a direction in which the surface is cyclic.

The integer fields `sv.U` and `sv.V` identify the mesh cell containing the patch, with `0 <= sv.U < nu` and `0 <= sv.V < nv`. All corners of a patch are assigned the same cell, so a pen that depends only on `sv.U` and `sv.V` gives each patch a single flat color:

```
pen checkerboard(surfaceVertex sv) {
  return (sv.U+sv.V) % 2 == 0 ? white : black;
}
draw(s, checkerboard);
```

The field `sv.patch` is the position of the patch in the array `s.s` of patches that make up the surface `s`. It is available for every surface, but the order of this array is an internal detail of how the surface was constructed, which most users need not be aware of. Coloring by `sv.patch` (including `cornerPen` with a different row of pens for each patch) should therefore be a last resort, appropriate mainly for a surface that was assembled patch by patch, so that the order is known. Wherever possible, use the position `sv.z`, the parametric coordinates `sv.uv`, or the mesh cell `sv.U`, `sv.V` instead.

The function `palette` (in the `palette` module) constructs a `vertexPen` that colors each vertex according to the value of a function `f` there:

```
vertexPen palette(real f(triple), real Min, real Max, pen[] palette);
```

The value `Min` is mapped to the first pen of the palette, `Max` to the last, and the values in between to the pens in between. For example, to color a surface by height:[2]

```
real height(triple v) {return v.z;}
draw(s,palette(height,0,1,Gradient(blue,red)));
```

Here a vertex at height 0 is blue, a vertex at height 1 is red, and a vertex at height 0.5 is halfway between the two. The value of `f` at every vertex must lie between `Min` and `Max`.

When the range of `f` is not known in advance, the surface method `palette` finds it, using the least and greatest values of `f` at the vertices of the surface:

```
draw(s,s.palette(height,Gradient(blue,red)));
```

Now the lowest vertex of `s` is blue and the highest is red. The resulting `vertexPen` should be used to draw that same surface. Here is a complete example, drawn with `nolight` so that the colors appear exactly as specified:

```
import graph3;
import palette;

size(200,0);

currentprojection=perspective(10,8,4);

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

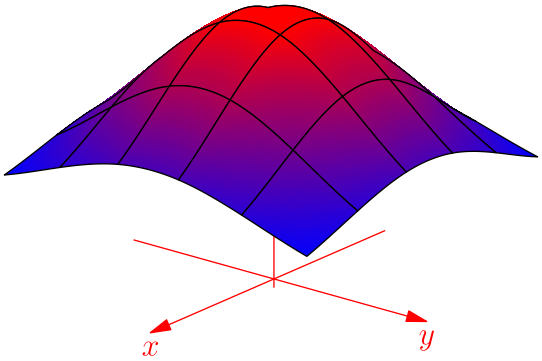
real height(triple v) {return v.z;}
```

[2] The built-in function `zpart` does the same as `height`, so it can be passed to `palette` directly, with nothing to define.

```
surface s=surface(f,(-1,-1),(1,1),nx=5,Spline);

draw(s,s.palette(height,Gradient(blue,red)),meshpen=black+thick(),nolight);

xaxis3(Label("$x$"),red,Arrow3);
yaxis3(Label("$y$"),red,Arrow3);
zaxis3(XYZero(extend=true),red,Arrow3);
```



The function `fitColors` (also in the `palette` module) constructs a `vertexPen` from the colors at a few points, such as the corners of a square or a box, blending smoothly between them; the pen `colors[i]` is the color at the point `coords[i]`:

```
vertexPen fitColors(triple[] coords, pen[] colors);
```

Each color component of the resulting `vertexPen` is a cubic polynomial in the position, chosen by least squares to fit the given colors, with a preference for polynomials of low degree. The fit works equally well whether or not the square or box is aligned with the coordinate axes. Since such a polynomial has only twenty coefficients, the given colors are closely approximated only when there are few of them; `fitColors` cannot produce an intricate pattern from a large number of points. If any of the pens is not opaque, the opacity is fit in the same way, using the blend mode of the first pen; otherwise the surface is opaque. If the pens are in different colorspaces, they are converted to that of the first pen `colors[0]`.

Surface coloring is illustrated in the examples `smoothelevation.asy` and `sphericalharmonic.asy` (coloring by height or radius with `surface.palette`), `vertexshading.asy` (`cornerPen`), `interlockedtori.asy` and `mobiustorus.asy` (coloring by parametric coordinates and by mesh cell), and `fitcolors.asy` (`fitColors`).

8.5 colormap

This module implements a list of colormaps converted from the Python library matplotlib. The colormaps can be used to generate palettes with the `palette()` method. For example:

```
import colormap;
pen[] Palette = wistia.palette();
```

There are two types of palettes: segmented palettes (which accept parameters for number of colors and gamma) and listed palettes (which return a fixed set of colors). See `https://`

matplotlib.org/tutorials/colors/colormaps.html for the complete list of available colormaps.

8.6 texcolors

This module defines the standard 68 CMYK TEX colors as named pens. These can be imported with:

```
import texcolors;
```

The colors include names like GreenYellow, Goldenrod, Melon, Mahogany, and many others defined in the standard TEX color set.

8.7 x11colors

This module defines the standard 140 RGB X11 colors as named pens. These can be imported with:

```
import x11colors;
```

The colors include names like AliceBlue, Aquamarine, CadetBlue, Chartreuse, and many others from the standard X11 color set.

Note that there is some overlap between the TEX and X11 color standards, and the definitions of some colors (such as Green) actually disagree.

8.8 fontsize

This module provides support for nonstandard font sizes by loading the type1cm TEX package. Import this module at the beginning of your file to use arbitrary font sizes with fontsize():

```
import fontsize;
defaultpen(fontsize(10pt));
```

8.9 contour

This module draws contour lines. To construct contours corresponding to the values in a real array c for a function f on box(a,b), use the routine

```
guide[][] contour(real f(real, real), pair a, pair b,
                  real[] c, int nx=ngraph, int ny=nx,
                  interpolate join=operator --, int subsample=1);
```

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 ..) may produce smoother contours but it can also lead to overshooting. The subsample parameter indicates the number of interior points that should be used to sample contours within each 1 x 1 box; the default value of 1 is usually sufficient.

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

```
guide[][] contour(real[][] f, pair a, pair b, real[] c,
                  interpolate join=operator --, int subsample=1);
```

To construct contours for an array of data values on a nonoverlapping regular mesh specified by the two-dimensional array `z`,

```
guide[][] contour(pair[][] z, real[][] f, real[] c,
                  interpolate join=operator --, int subsample=1);
```

To construct contours for an array of values `f` specified at irregularly positioned points `z`, use the routine

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

The contours themselves can be drawn with one of the routines

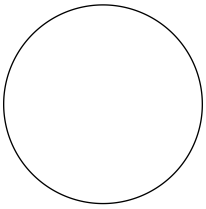
```
void draw(picture pic=currentpicture, Label[] L=new Label[],
          guide[][] g, pen p=currentpen);

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;

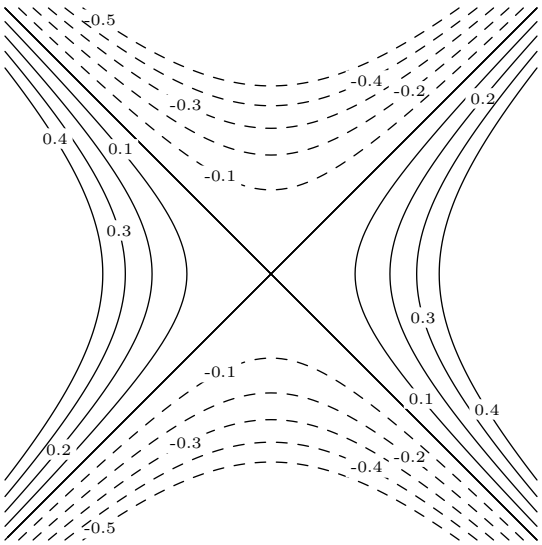
size(200);

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(6pt);
  },c.length);
```

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

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



The next examples illustrates how contour lines can be drawn on color density images, with and without palette quantization:

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

size(10cm,10cm);

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=1;
int n=Divs*divs;

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

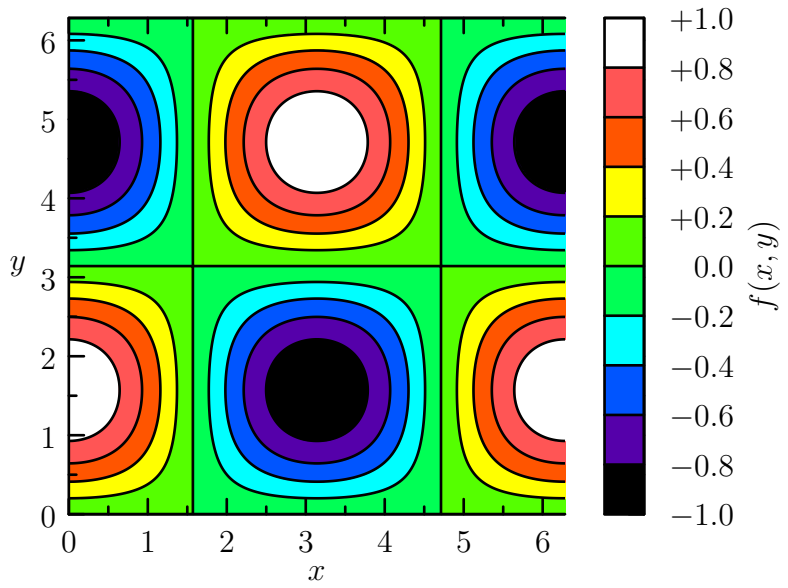
```
bounds range=image(f,Automatic,a,b,3N,Palette,n);

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

// Minor contours (if divs > 1)
real[] cvals;
for(int i=0; i < Cvals.length-1; ++i)
  cvals.append(uniform(Cvals[i],Cvals[i+1],divs)[1:divs]);
draw(contour(f,a,b,cvals,N,operator --),tickpen);

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

palette("$f(x,y)$",range,point(SE)+(0.5,0),point(NE)+(1,0),Right,Palette,
        PaletteTicks("$%+#0.1f$",N=Divs,n=divs,Tickpen,tickpen));
```




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

size(10cm,10cm);

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=1;

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

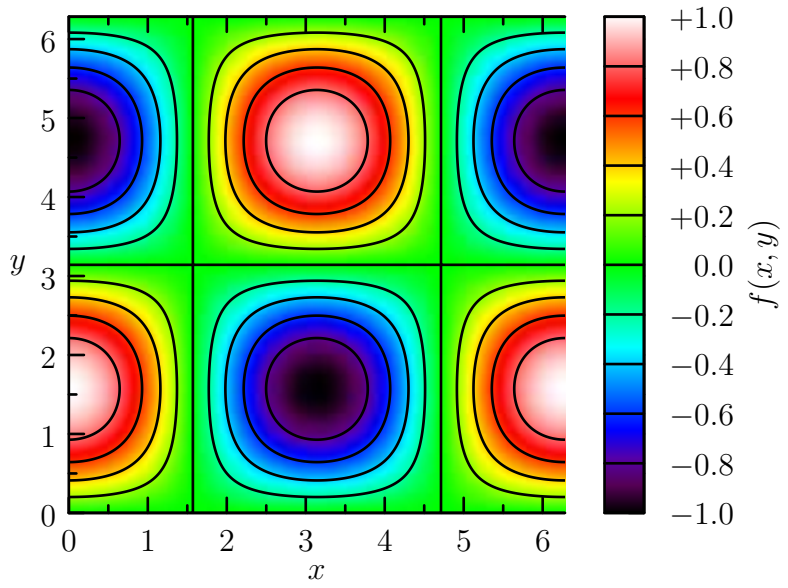
bounds range=image(f,Automatic,a,b,N,Palette);

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

// Minor contours (if divs > 1)
real[] cvals;
for(int i=0; i < Cvals.length-1; ++i)
  cvals.append(uniform(Cvals[i],Cvals[i+1],divs)[1:divs]);
draw(contour(f,a,b,cvals,N,operator --),tickpen+squarecap+beveljoin);

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

palette("$f(x,y)$",range,point(SE)+(0.5,0),point(NE)+(1,0),Right,Palette,
        PaletteTicks("$%+#0.1f$",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;

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

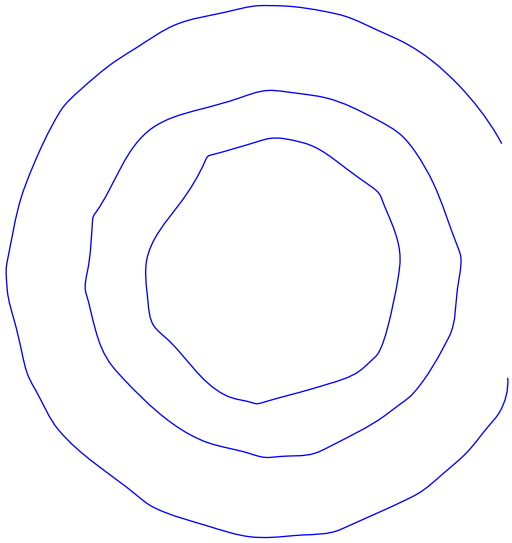
srand(1);

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

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

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

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

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

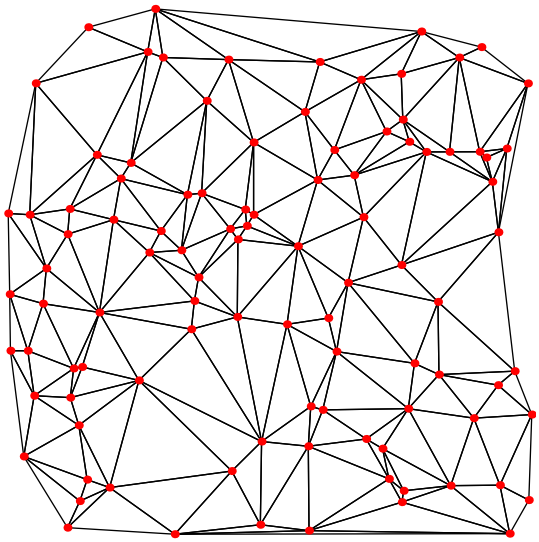
real r() {return 1.2*(rand()/randMax*2-1);}

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.asy` illustrates how to produce color density images over such irregular triangular meshes. `Asymptote` uses a robust version of Paul Bourke's Delaunay triangulation algorithm based on the public-domain exact arithmetic predicates written by Jonathan Shewchuk.

8.10 geometry

This module, written by Philippe Ivaldi, provides an extensive set of geometry routines, including `perpendicular` symbols and a `triangle` structure. Link to the documentation for the `geometry` module are posted here: `https://asymptote.sourceforge.io/links.html`, including an extensive set of examples, `https://web.archive.org/web/20201130113133/http://www.piprime.fr/files/asymptote/geometry/`, and an index:

`https://web.archive.org/web/20201130113133/http://www.piprime.fr/files/asymptote/geometry/modules/geometry.asy.index.type.html`

8.11 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 `https://web.archive.org/web/20201130113133/http://www.piprime.fr/files/asymptote/grid3/`):

```
import grid3;

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

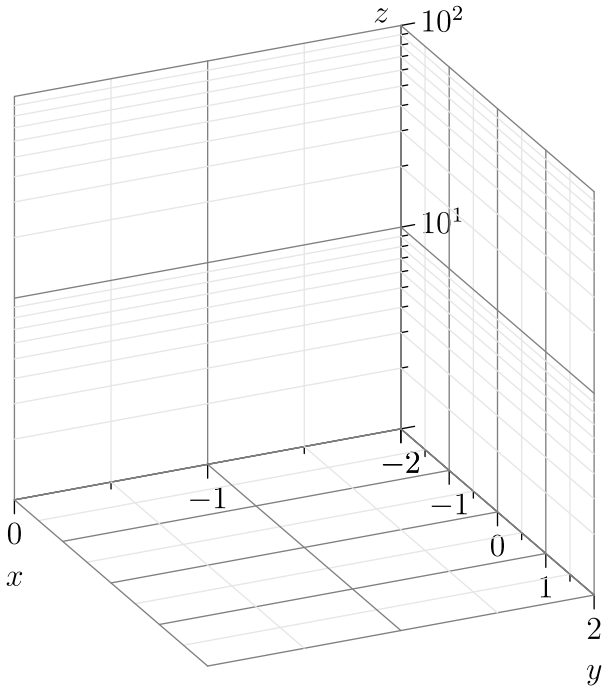
scale(Linear, Linear, Log);

limits((-2,-2,1),(0,2,100));

grid3(XYZgrid);

xaxis3(Label("$x$",position=EndPoint,align=S),Bounds(Min,Min),
```

```
       OutTicks());
yaxis3(Label("$y$",position=EndPoint,align=S),Bounds(Min,Min),OutTicks());
zaxis3(Label("$z$",position=EndPoint,align=(-1,0.5)),Bounds(Min,Min),
       OutTicks(beginlabel=false));
```



8.12 interpolate

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

8.13 markers

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

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

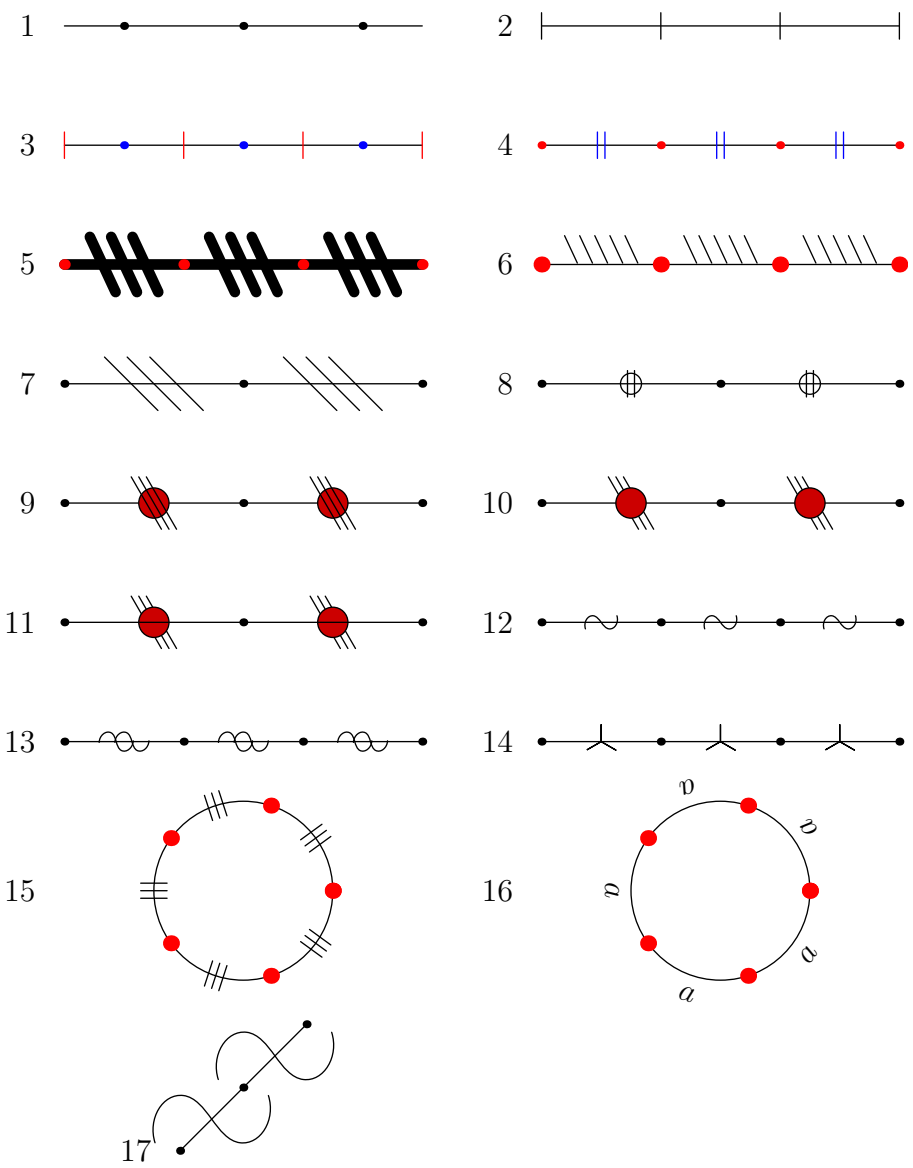
which centers `n` copies of frame `f` within uniformly space intervals in arclength along the path, optionally rotated by the angle of the local tangent.

The `marker` (see [marker], page 128) 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:

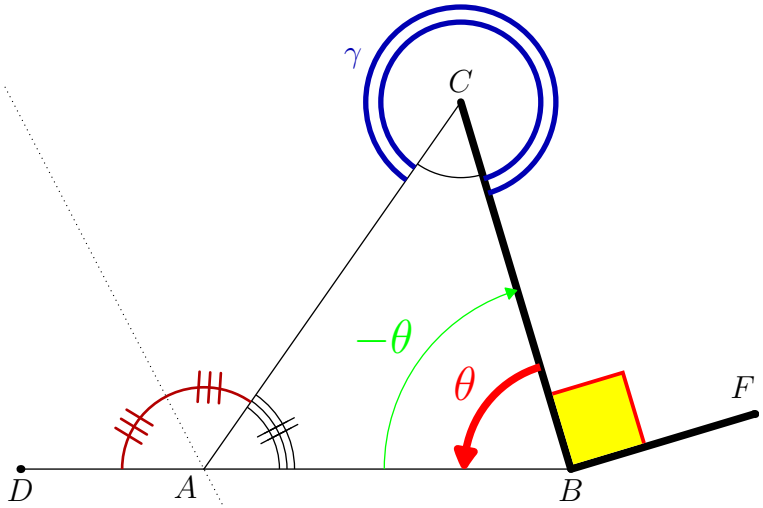


This module also provides a routine for marking an angle AOB:

```
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`.



8.14 math

This module extends `Asymptote`'s mathematical capabilities with useful functions such as

`void drawline(picture pic=currentpicture`
: 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);`
: returns 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)`.

`pair[] quarticroots(real a, real b, real c, real d, real e);`
: returns the four complex roots of the quartic equation $ax^4+bx^3+cx^2+dx+e=0$.

`real time(path g, real x, int n=0, real fuzz=-1)`
: returns the `n`th intersection time of path `g` with the vertical line through `x`.

`real time(path g, explicit pair z, int n=0, real fuzz=-1)`
: returns the `n`th intersection time of path `g` with the horizontal line through `(0,z.y)`.

`real value(path g, real x, int n=0, real fuzz=-1)`
: returns the `n`th `y` value of `g` at `x`.

`real value(path g, explicit pair z, int n=0, real fuzz=-1)`
returns the `n`th `x` value of `g` at `y=z.y`.

`real slope(path g, real x, int n=0, real fuzz=-1)`
returns the `n`th slope of `g` at `x`.

`real slope(path g, explicit pair z, int n=0, real fuzz=-1)`
returns the `n`th slope of `g` at `y=z.y`.

int[][] segment(bool[] b) returns the indices of consecutive true-element segments of bool[] `b`.

`real[] partialsum(real[] a)`
returns the partial sums of a real array `a`.

`real[] partialsum(real[] a, real[] dx)`
returns the partial `dx`-weighted sums of a real array `a`.

`bool increasing(real[] a, bool strict=false)`
returns, if `strict=false`, whether `i > j` implies `a[i] >= a[j]`, or if `strict=true`, whether `i > j` implies implies `a[i] > a[j]`.

`int unique(real[] a, real x)`
if the sorted array `a` does not contain `x`, insert it sequentially, returning the index of `x` in the resulting array.

`bool lexorder(pair a, pair b)`
returns the strict lexicographical partial order of `a` and `b`.

`bool lexorder(triple a, triple b)`
returns the strict lexicographical partial order of `a` and `b`.

8.15 patterns

This module implements `PostScript` tiling patterns and includes several convenient pattern generation routines.

8.16 roundedpath

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

8.17 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,
                   arrowbar arrow=None);
```

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);
```

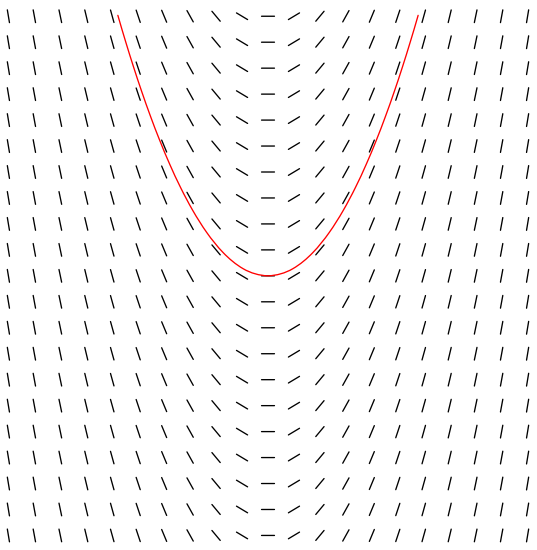
takes a point (`c`) and a slope field-defining function `f` and returns, as a path, the curve passing through that point. The points `a` and `b` represent the rectangular boundaries over which the curve is interpolated.

Both `slopefield` and `curve` alternatively accept a function `real f(real)` that depends on x only, as seen in this example:

```
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);
```



8.18 smoothcontour3

This module, written by Charles Staats, draws implicitly defined surfaces with smooth appearance. The purpose of this module is similar to that of `contour3`: given a real-valued function $f(x, y, z)$, construct the surface described by the equation $f(x, y, z) = 0$. The `smoothcontour3` module generally produces nicer results than `contour3`, but takes longer to compile. Additionally, the algorithm assumes that the function and the surface are both smooth; if they are not, then `contour3` may be a better choice.

To construct the null surface of a function `f(triple)` or `ff(real,real,real)` over `box(a,b)`, use the routine

```
surface implicitsurface(real f(triple)=null,
```

```
real ff(real,real,real)=null,
triple a,
triple b,
int n=nmesh,
bool keyword overlapedges=false,
int keyword nx=n,
int keyword ny=n,
int keyword nz=n,
int keyword maxdepth=8,
bool usetriangles=true);
```

The optional parameter `overlapedges` attempts to compensate for an artifact that can cause the renderer to "see through" the boundary between patches. Although it defaults to `false`, it should usually be set to `true`. The example `genustwo.asy` illustrates the use of this function. Additional examples, together with a more in-depth explanation of the module's usage and pitfalls, are available at `https://github.com/charlesstaats/smoothcontour3`.

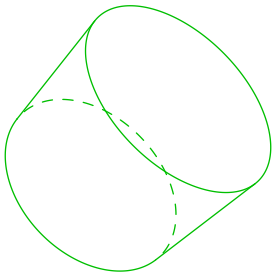
8.19 solids

This solid geometry module 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);
draw(r,heavygreen);
```



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

The structure `skeleton` contains the three-dimensional wireframe used to visualize a volume of revolution:

```
struct skeleton {
  struct curve {
    path3[] front;
    path3[] back;
```

```
  }
  // transverse skeleton (perpendicular to axis of revolution)
  curve transverse;
  // longitudinal skeleton (parallel to axis of revolution)
  curve longitudinal;
}
```

8.20 stats

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

8.21 tube

This module extends the `tube` surfaces constructed in `three_arrows.asy` to arbitrary cross sections, colors, and spine transformations. The routine

```
surface tube(path3 g, coloredpath section,
             transform T(real)=new transform(real t) {return identity();},
             real corner=1, real relstep=0);
```

draws a tube along `g` with cross section `section`, after applying the transformation `T(t)` at `point(g,t)`. The parameter `corner` controls the number of elementary tubes at the angular points of `g`. A nonzero value of `relstep` specifies a fixed relative time step (in the sense of `relpoint(g,t)`) to use in constructing elementary tubes along `g`. The type `coloredpath` is a generalization of `path` to which a `path` can be cast:

```
struct coloredpath
{
  path p;
  pen[] pens(real);
  int colortype=coloredSegments;
}
```

Here `p` defines the cross section and the method `pens(real t)` returns an array of pens (interpreted as a cyclic array) used for shading the tube patches at `relpoint(g,t)`. If `colortype=coloredSegments`, the tube patches are filled as if each segment of the section was colored with the pen returned by `pens(t)`, whereas if `colortype=coloredNodes`, the tube components are vertex shaded as if the nodes of the section were colored.

A `coloredpath` can be constructed with one of the routines:

```
coloredpath coloredpath(path p, pen[] pens(real),
                        int colortype=coloredSegments);
coloredpath coloredpath(path p, pen[] pens=new pen[] {currentpen},
                        int colortype=coloredSegments);
coloredpath coloredpath(path p, pen pen(real));
```

In the second case, the pens are independent of the relative time. In the third case, the array of pens contains only one pen, which depends of the relative time.

The casting of `path` to `coloredpath` allows the use of a `path` instead of a `coloredpath`; in this case the shading behavior is the default shading behavior for a surface.

An example of `tube` is provided in the file `trefoilknot.asy`. Further examples can be found at `https://web.archive.org/web/20201130113133/http://www.piprime.fr/files/asymptote/tube`.

9 Specialty modules

These modules are for specialized applications.

9.1 animation

This module allows one to generate animations, as illustrated by the files `wheel.asy`, `wavepacket.asy`, and `cube.asy` in the `animations` subdirectory of the examples directory. These animations use the `ImageMagick` `magick` program to merge multiple images into a GIF or MPEG movie.

The related `animate` module, derived from the `animation` module, generates higher-quality portable clickable PDF movies, with optional controls. This requires installing the module

`https://mirror.ctan.org/macros/latex/contrib/animate/animate.sty`

(version 2007/11/30 or later) in a new directory `animate` in the local LaTeX directory (for example, in `/usr/local/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`, illustrate the use of embedded PDF movies. The examples `inlinemovie.tex` and `inlinemovie3.tex` show how to generate and embed PDF movies directly within a LaTeX file (see Chapter 7 [LaTeX usage], page 112). The member function

```
string pdf(fit fit=NoBox, real delay=animationdelay, string options="",
           bool keep=settings.keep, bool multipage=true);
```

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

`https://mirror.ctan.org/macros/latex/contrib/animate/doc/animate.pdf`

9.2 animate

This module loads the TeX `animate` package before importing the `animation` module to generate portable clickable PDF movies with optional controls.

9.3 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.

9.4 babel

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

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

9.5 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`:

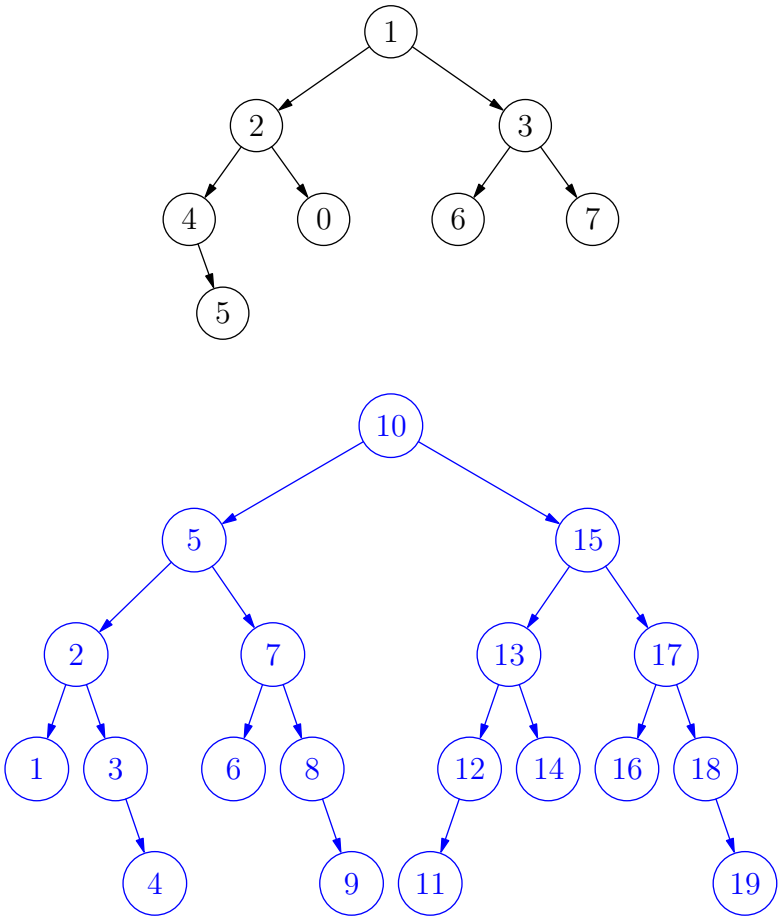
```
import binarytree;

picture pic,pic2;

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

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,condensed=true);

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



9.6 bsp

This module implements hidden surface removal of planar pictures using a binary space partition (BSP) and picture clipping. It is used for projecting three-dimensional scenes to two dimensions with proper occlusion.

9.7 CAD

This module, 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`.

9.8 contour3

This module 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`.

9.9 drawtree

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

9.10 embed

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

`https://mirror.ctan.org/macros/latex/contrib/media9`

for embedding movies, sounds, and 3D objects into a PDF document.

A more portable method for embedding movie files, which should work on any platform and does not require the `media9` package, is provided by using the `external` module instead of `embed`.

Examples of the above two interfaces is provided in the file `embeddedmovie.asy` in the `animations` subdirectory of the examples directory and in `externalmovie.asy`. For a higher quality embedded movie generated directly by `Asymptote`, use the `animate` module along with the `animate.sty` package to embed a portable PDF animation (see Section 9.2 [animate], page 193).

An example of embedding `U3D` code is provided in the file `embeddedu3d`.

9.11 external

This module provides a portable method for embedding movie files in PDF documents without requiring the `media9` package. It is an alternative to the `embed` module for better cross-platform compatibility.

9.12 feynman

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

9.13 flowchart

This module provides routines for 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, pair center=(0,0),
                pen headerpen=mediumgray, pen bodypen=invisible,
                pen drawpen=currentpen,
                real dx=3, real minheaderwidth=minblockwidth,
                real minheaderheight=minblockwidth,
                real minbodywidth=minblockheight,
                real minbodyheight=minblockheight);
block rectangle(object body, pair center=(0,0),
                pen fillpen=invisible, pen drawpen=currentpen,
                real dx=3, real minwidth=minblockwidth,
                real minheight=minblockheight);
```

parallelogram block:

```
block parallelogram(object body, pair center=(0,0),
                    pen fillpen=invisible, pen drawpen=currentpen,
                    real dx=3, real slope=2,
                    real minwidth=minblockwidth,
                    real minheight=minblockheight);
```

diamond-shaped block:

```
block diamond(object body, pair center=(0,0),
              pen fillpen=invisible, pen drawpen=currentpen,
              real ds=5, real dw=1,
              real height=20, real minwidth=minblockwidth,
              real minheight=minblockheight);
```

circular block:

```
block circle(object body, pair center=(0,0), pen fillpen=invisible,
             pen drawpen=currentpen, real dr=3,
             real mindiameter=mincirclediameter);
```

rectangular block with rounded corners:

```
block roundrectangle(object body, pair center=(0,0),
                     pen fillpen=invisible, pen drawpen=currentpen,
                     real ds=5, real dw=0, real minwidth=minblockwidth,
                     real minheight=minblockheight);
```

rectangular block with beveled edges:

```
block bevel(object body, pair center=(0,0), pen fillpen=invisible,
            pen drawpen=currentpen, real dh=5, real dw=5,
            real minwidth=minblockwidth, real minheight=minblockheight);
```

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

```
path path(pair point[], ... flowdir dir[]);
```

The entries in `dir` identify whether successive segments between the pairs specified by `point` should be drawn in the `Horizontal` or `Vertical` direction.

Here is a simple flowchart example (see also the example `controlsystem.asy`):

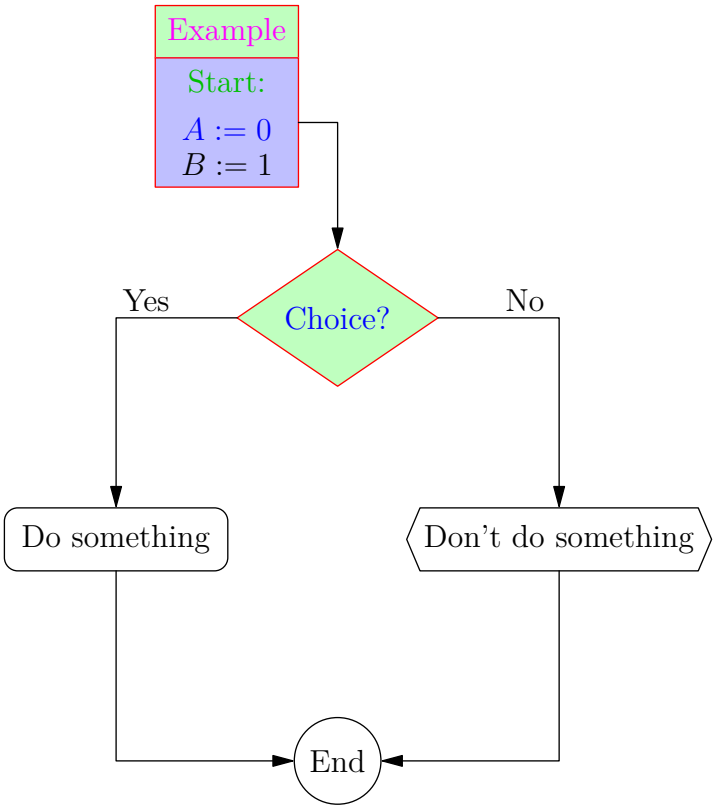
```
size(0,300);

import flowchart;

block block1=rectangle(Label("Example",magenta),
                       pack(Label("Start:",heavygreen),"",Label("$A:=0$",blue),
                            "$B:=1$"),(-0.5,3),palegreen,paleblue,red);
block block2=diamond(Label("Choice?",blue),(0,2),palegreen,red);
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(picture pic, transform t) {
    blockconnector operator --=blockconnector(pic,t);
    //    draw(pic,block1.right(t)--block2.top(t));
    block1--Right--Down--Arrow--block2;
    block2--Label("Yes",0.5,NW)--Left--Down--Arrow--block3;
    block2--Right--Label("No",0.5,NE)--Down--Arrow--block4;
    block4--Down--Left--Arrow--block5;
    block3--Down--Right--Arrow--block5;
  });
```



9.14 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 module requires the latex tex engine and inherits the limitations of the PSTricks \pstextpath macro.

9.15 labelpath3

This module, contributed by Jens Schwaiger, implements a 3D version of labelpath that does not require the PSTricks package. An example is provided in curvedlabel3.asy.

9.16 lmfit

This module provides least-squares fitting routines for data analysis. It can be used to fit mathematical models to experimental data.

9.17 MetaPost

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

Unlike MetaPost, Asymptote does not implicitly solve linear equations and therefore does not have the notion of a whatever unknown. The routine extension (see [extension], page 36) provides a useful replacement for a common use of whatever: finding the intersection point of the lines through P, Q and p, q. For less common occurrences of whatever, one can use the built-in explicit linear equation solver solve instead.

9.18 obj

This module allows one to construct surfaces from simple obj files, as illustrated in the example files galleon.asy and triceratops.asy.

9.19 ode

The ode module, illustrated in the example odetest.asy, implements a number of explicit numerical integration schemes for ordinary differential equations.

9.20 pstoedit

This module provides support for converting PostScript files to Asymptote code using the pstoedit utility with the Asymptote backend.

9.21 rational

This module implements rational number arithmetic for exact computations without floating-point errors. It provides a rational type with support for basic arithmetic operations.

9.22 rationalSimplex

This module provides the simplex method for linear programming using rational arithmetic. It is an exact version of the simplex module that avoids floating-point precision issues.

9.23 simplex

This module solves the general linear programming problem using the simplex method.

9.24 size10

This module provides a simple way to use the named LaTeX font sizes corresponding to \documentclass[10pt]{article}:

```
import size10;
pen small=fontcommand("\small");
label("M",0,W,red+small);
pen normal=fontsize(9pt);
label("M",0,E,normal);
```

9.25 size11

This module provides a simple way to use the named LaTeX font sizes corresponding to `\documentclass[11pt]{article}`. The usage is similar to `size10`.

9.26 slide

This module 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 `slidedemo.asy`.

9.27 syzygy

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

9.28 tree

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

9.29 trembling

This module, written by Philippe Ivaldi and illustrated in the example `floatingdisk.asy`, allows one to draw wavy lines, as if drawn by hand.

10 Developer modules

These modules are for `Asymptote` developers.

10.1 plain

This is the default `Asymptote` base file 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.

10.2 bezulate

This module provides the `bezulate` routine for subdivision surface generation. It decomposes (possibly nonsimply connected) meshes into a set of Bezier patches suitable for rendering. The module is used internally by `Asymptote` for creating smooth surfaces from mesh data.

10.3 simplex2

This module solves a special case of the two-variable linear programming problem used by the module `plain` for automatic sizing of pictures (see [deferred drawing], page 56).

10.4 v3d

This module provides support for the V3D 3D graphics format, which can be viewed using the Python `pyv3d` library, an `Okular` plugin (`https://github.com/vectorgraphics/Okular-v3d-Plugins`), or embedded within PDF documents. V3D content can be automatically generated by setting `settings.v3d=true`, or existing V3D files can be imported using the `importv3d` function:

```
import v3d;
importv3d("file.v3d");
```

11 Command-line options

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

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

Options (negate boolean options by replacing - with -no):

-GPUblockSize n        Compute shader block size [8]
-GPUcompress           Compress GPU transparent fragment counts [false]
-GPUindexing           Compute indexing partial sums on GPU [true]
-GPUinterlock          Use fragment shader interlock [true]
-GPUlocalSize n        Compute shader local size [256]
-V,-View               View output; command-line only
-absolute              Use absolute WebGL dimensions [false]
-a,-align C|B|T|Z      Center, Bottom, Top, or Zero page alignment [C]
-aligndir pair         Directional page alignment (overrides align) [(0,0)]
-antialias n           Antialiasing width for rasterized output [2]
-auto3D                Automatically activate 3D scene [true]
-autobillboard         3D labels always face viewer by default [true]
-autoimport str        Module to automatically import
-autoplain             Enable automatic importing of plain [true]
-autoplay              Autoplay 3D WebGL animations [true]
-autorotate            Enable automatic PDF page rotation [false]
-axes3                 Show 3D axes in PDF output [true]
-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
-cmyk                  Convert rgb colors to cmyk false
-c,-command str        Command to autoexecute
-compact               Conserve memory at the expense of speed false
-compress              Compress images in PDF output [true]
-convertOptions str    []
-curlAfterRead         Allow libcurl after reading local files via input() false
-d,-debug              Enable debugging messages and traceback false
-device str            Vulkan device, by index or name (see asy -vv)
-digits n              Default output file precision [7]
-divisor n             Garbage collect using purge(divisor=n) [2]
-dvipsOptions str      []
-dvisvgmMultipleFiles  dvisvgm supports multiple files [true]
-dvisvgmOptions str    [--optimize]
-embed                 Embed rendered preview image [true]
-e,-environment        Show summary of environment settings; command-line only
-exitonEOF             Exit interactive mode on EOF [true]
-fitscreen             Fit rendered image to screen [true]
-framerate frames/s    Animation speed [30]
-fxaa                  Enable FXAA. Multisampling is turned off if FXAA is enabled [false]
```

```
-glOptions str          []
-globalread             Allow read from other directory true
-globalwrite            Allow write to other directory false
-gray                   Convert all colors to grayscale false
-gsOptions str          []
-h,-help                Show summary of options; command-line only
-historylines n         Retain n lines of history [1000]
-htmlviewerOptions str
                        []
-hyperrefOptions str    [setpagesize=false,unicode,pdfborder=0 0 0]
-ibl                    Enable environment map image-based lighting [false]
-image str              Environment image name [snowyField]
-imageDir str           Environment image library directory [ibl]
-imageURL str           Environment image library URL [https://vectorgraphics.gitlab.io/asym
-inlineimage            Generate inline embedded image [false]
-inlinetex              Generate inline TeX code [false]
-inpipe n               Input pipe [-1]
-interactiveMask        Mask fpu exceptions in interactive mode [true]
-interactiveView        View output in interactive mode [true]
-interactiveWrite       Write expressions entered at the prompt to stdout [true]
-k,-keep                Keep intermediate files [false]
-keepaux                Keep intermediate LaTeX .aux files [false]
-keys                   Generate WebGL keys false
-level n                Postscript level [3]
-licenses               Show license and third-party attribution; command-line only
-l,-listvariables       List available global functions and variables [false]
-localhistory           Use a local interactive history file [false]
-loop                   Loop 3D animations [false]
-lossy                  Use single precision for V3D reals [false]
-lsp                    Interactive mode for the Language Server Protocol [false]
-m,-mask                Mask fpu exceptions; command-line only
-maxFramesInFlight n    Maximum frames queued to the GPU [3]
-maxtile pair           Maximum rendering tile size [(1024,768)]
-maxviewport pair       Maximum viewport size [(0,0)]
-multiline              Input code over multiple lines at the prompt [false]
-multipleView           View output from multiple batch-mode files [false]
-multisample n          Multisampling width for screen images [4]
-offline                Produce offline html files [false]
-offscreen              Use offscreen rendering [false]
-O,-offset pair         PostScript offset [(0,0)]
-f,-outformat format    Convert each output file to specified format
-o,-outname name        Alternative output directory/file prefix
-outpipe n              Output pipe [-1]
-paperheight bp         Default page height [0]
-paperwidth bp          Default page width [0]
-p,-parseonly           Parse file [false]
-pdfreload              Automatically reload document in pdfviewer [false]
```

```
-pdfreloadOptions str  []
-pdfreloaddelay usec   Delay before attempting initial pdf reload [750000]
-pdfviewerOptions str  []
-position pair         Initial 3D rendering screen position [(0,0)]
-prc                   Embed 3D PRC graphics in PDF output [false]
-prerender resolution  Prerender V3D objects (0 implies vector output) [0]
-prompt str            Prompt [> ]
-prompt2 str           Continuation prompt for multiline input  [..]
-psviewerOptions str   []
-q,-quiet              Suppress welcome text and noninteractive stdout [false]
-render n              Render 3D graphics using n pixels per bp [2]
-resizestep step       Resize step [1.2]
-reverse               reverse 3D animations [false]
-rgb                   Convert cmyk colors to rgb false
-safe                  Disable system call true
-scroll n              Scroll standard output n lines at a time [0]
-shiftHoldDistance n   WebGL touch screen distance limit for shift mode [20]
-shiftWaitTime ms      WebGL touch screen shift mode delay [200]
-spinstep deg/s        Spin speed [60]
-srgb                  Render 3D images in sRGB space [false]
-svgemulation          Emulate unimplemented SVG shading [true]
-tabcompletion         Interactive prompt auto-completion [true]
-tex engine            latex|pdflatex|xelatex|lualatex|tex|pdftex|luatex|context|none [late
-thick                 Render thick 3D lines [true]
-thin                  Render thin 3D lines [true]
-threads               Use POSIX threads for 3D rendering [true]
-toolbar               Show 3D toolbar in PDF output [true]
-s,-translate          Show translated virtual machine code [false]
-twice                 Run LaTeX twice (to resolve references) [false]
-twosided              Use two-sided 3D lighting model for rendering [true]
-u,-user str           General purpose user string
-v3d                   Embed 3D V3D graphics in PDF output [false]
-v,-verbose            Increase verbosity level (can specify multiple times) 0
-version               Show version; command-line only
-vibrateTime ms        WebGL shift mode vibrate duration [25]
-viewportmargin pair   Horizontal and vertical 3D viewport margin [(0.5,0.5)]
-vsync                 Vertically synchronize with monitor [false]
-vulkan                Use Vulkan renderer if available [true]
-wait                  Wait for child processes to finish before exiting [false]
-warn str              Enable warning; command-line only
-webgl2                Use webgl2 if available [false]
-where                 Show where listed variables are declared [false]
-wsl                   Run asy under the Windows Subsystem for Linux [false]
-xasy                  Interactive mode for xasy false
-zoomPinchCap limit    WebGL maximum zoom pinch [100]
-zoomPinchFactor n     WebGL zoom pinch sensitivity [10]
-zoomThreshold threshold
```

```
                        Zoom remesh threshold [0.02]
-zoomfactor factor      Zoom step factor [1.05]
-zoomstep step          Mouse motion zoom step [0.1]
```

All boolean options can be negated by prepending `no` to the option name.

If no arguments are given, `Asymptote` runs in interactive mode (see Chapter 12 [Interactive mode], page 208). 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, one or more modules are automatically imported for each run, immediately after loading module `plain`. Multiple modules may be specified as a comma-separated list, for example `-autoimport three,graph`. Path syntax is also accepted (for example, to autoimport a personal module kept in a subdirectory, `-autoimport mydir/common`; the module is imported under its last path component name, here `common`). Note that templated modules (such as `collections/hashmap`) cannot be autoimported because they require type parameters.

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 (see Section 2.5 [Search paths], page 6). 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`, `hyperrefOptions`, `convertOptions`, `gsOptions`, `htmlviewerOptions`, `psviewerOptions`, `pdfviewerOptions`, `pdfreloadOptions`, `glOptions`, and `dvisvgmOptions` allow specialized options to be passed as a string to the respective applications or libraries. The default value of `hyperrefOptions` is `setpagesize=false,unicode,pdfborder=0 0 0`.

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` and `tex` tex engine and PDF for the `pdflatex`, `xelatex`, `context`, `luatex`, and `lualatex` tex engines. Alternative output formats may be produced using the `-f` option (or `outformat` setting).

To produce SVG output, you will need `dvisvgm` (version 3.2.1 or later) from `https://dvisvgm.de`, which can display SVG output (used by the `xasy` editor) for embedded EPS,

PDF, PNG, and JPEG images included with the `graphic()` function. The generated output is optimized with the default setting `settings.dvisvgmOptions="--optimize"`.

`Asymptote` can also produce any output format supported by the `ImageMagick` `magick` program (version 7 or later). The optional setting `-render n` requests an output resolution of `n` pixels per `bp`. Antialiasing is controlled by the parameter `antialias`, which by default specifies a sampling width of 2 pixels. To give other options to `magick`, use the `convertOptions` setting or call `magick convert` manually. This example emulates how `Asymptote` produces antialiased `tiff` output at one pixel per `bp`:

```
asy -o - venn | magick convert -alpha Off -density 144x144 -geometry 50%x eps:- venn.tiff
```

If the option `-nosafe` is given, `Asymptote` runs in unsafe mode. This enables the `int system(string s)` and `int system(string[] s)` calls, 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 pair `aligndir` specifies an optional direction on the boundary of the page (mapped to the rectangle [-1,1]×[-1,1]) to which the picture should be aligned; the default value `(0,0)` specifies center alignment.

The `-c` (`command`) option may be used to execute arbitrary `Asymptote` code on the command line as a string. It is not necessary to terminate the string with a semicolon. Multiple `-c` options are executed in the order they are given. For example

```
asy -c 2+2 -c "sin(1)" -c "size(100); draw(unitsquare)"
```

produces the output

```
4
0.841470984807897
```

and draws a unitsquare of size `100`.

The `-u` (`user`) option may be used to specify arbitrary `Asymptote` settings 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` (`listvariables`) option is used with file arguments, only global functions and variables defined in the specified file(s) are listed.

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.

Type `asy -licenses` to see the license and third-party attribution information for `Asymptote` and its bundled components. Type `asy -licenses=full` or `asy --licenses=full` to print the full license texts (read from the installed `licenses/` directory).

12 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`. If the expression is non-writable, its type signature will be printed out instead. In either case, the expression can be referred to using the symbol `%` in the next line input at the prompt. For example:

```
> 2+3
5
> %*4
20
> 1/%
0.05
> sin(%)
0.0499791692706783
> currentpicture
<picture currentpicture>
> %.size(200,0)
>
```

The `%` symbol, when used as a variable, is shorthand for the identifier `operator answer`, which is set by the prompt after each written expression evaluation.

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

`help` view the manual;

`erase` erase `currentpicture`;

`reset` reset the `Asymptote` environment to its initial state, except for changes to the settings module (see [settings], page 205), the current directory (see [cd], page 58), and breakpoints (see Chapter 18 [Debugger], page 216);

`input FILE`
does an interactive reset, followed by the command `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: `https://tiswww.case.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 3` 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")
```

13 Graphical User Interface

In the event that adjustments to the final figure are required, the preliminary Graphical User Interface (GUI) `xasy` included with `Asymptote` allows you to move graphical objects and draw new ones. The modified figure can then be saved as a normal `Asymptote` file.

13.1 GUI installation

As `xasy` is written in the interactive scripting language `Python/Qt`, it requires `Python` (`https://www.python.org`), along with the `Python` packages `PySide6`, `cson`, and `numpy`:

```
pip3 install cson numpy PySide6
```

Pictures are deconstructed into the SVG image format. Since `Qt5` does not support `SVG` clipping, you will need the `rsvg-convert` utility, which is part of the `librsvg2-tools` package on `UNIX` systems and the `librsvg` package on `macOS X`; under `Microsoft Windows`, it is available as

`https://sourceforge.net/projects/tumagcc/files/rsvg-convert-2.40.20.7z`

Ensure that `rsvg-convert` is available in `PATH`, or specify the location of `rsvg-convert` in `rsvgConverterPath` option within `xasy`'s settings.

Deconstruction of a picture into its components is fastest when using the LaTeX TeX engine. The default setting `dvisvgmMultipleFiles=true` speeds up deconstruction under PDF TeX engines.

13.2 GUI usage

The arrow keys (or mouse wheel) are convenient for temporarily raising and lowering objects within `xasy`, allowing an object to be selected. Pressing the arrow keys will pan while the shift key is held and zoom while the control key is held. The mouse wheel will pan while the alt or shift keys is held and zoom while the control key is held. In translate mode, an object can be dragged coarsely with the mouse or positioned finely with the arrow keys while holding down the mouse button.

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 54).

14 Command-Line Interface

`Asymptote` code may be sent to the `https://asymptote.ualberta.ca` server directly from the command line. The public endpoint is

```
https://asymptote.ualberta.ca/cli
```

Send your `.asy` source as the POST body (for example with `curl --data-binary`) and choose the output format with the `f` (or `outformat`) query argument; additional `asy` command-line options are passed as further query arguments. A GET to the endpoint returns the server's `asy -version` string.

- SVG output:

  ```
  curl --data-binary 'import venn;' 'https://asymptote.ualberta.ca/cli?f=svg'
  | display -
  ```

- PDF output:

  ```
  curl --data-binary 'import venn;' 'https://asymptote.ualberta.ca/cli?f=pdf'
  -o venn.pdf
  ```

- HTML output:

  ```
  curl --data-binary @/usr/local/share/doc/asymptote/examples/Klein.asy
  'https://asymptote.ualberta.ca/cli' -o Klein.html
  ```

- V3D output:

  ```
  curl --data-binary 'import teapot;' 'https://asymptote.ualberta.ca/cli?f=v3d'
  -o teapot.v3d
  ```

- PDF output with embedded V3D content and rendered bitmap at 2 pixels per bp:

  ```
  curl --data-binary 'import teapot;' 'https://asymptote.ualberta.ca/cli?f=pdf&v3d'
  -o teapot.pdf
  ```

- PDF output with rendered bitmap at 2 pixels per bp:

  ```
  curl --data-binary 'import teapot;' 'https://asymptote.ualberta.ca/cli?f=pdf'
  -o teapot.pdf
  ```

- PDF output with rendered bitmap at 4 pixels per bp:

  ```
  curl --data-binary 'import teapot;' 'https://asymptote.ualberta.ca/cli?f=pdf&render=4'
  -o teapot.pdf
  ```

- PRC output (legacy format):

  ```
  curl --data-binary 'import teapot;' 'https://asymptote.ualberta.ca/cli?f=prc'
  -o teapot.prc
  ```

- PDF output with embedded PRC content:

  ```
  curl --data-binary 'import teapot;' 'https://asymptote.ualberta.ca/cli?f=pdf&prc'
  -o teapot.pdf
  ```

- PDF output with embedded PRC content and rendered preview bitmap at 4 pixels per bp:

  ```
  curl --data-binary 'import teapot;' 'https://asymptote.ualberta.ca/cli?f=pdf&prc&render=4
  -o teapot.pdf
  ```

The same requests are also accepted over plain HTTP directly on port `10007` (for example `curl --data-binary 'import venn;' 'http://asymptote.ualberta.ca:10007?f=svg'`). The HTTPS endpoint above is preferred; the direct port will eventually be discontinued.

The source code for the command-line interface is available at `https://github.com/vectorgraphics/asymptote-http-server`.

15 Language server protocol

Under UNIX and macOS X, Asymptote supports features of the Language Server Protocol (LSP) (https://en.wikipedia.org/wiki/Language_Server_Protocol), including function signature and variable matching. Under MSWindows, Asymptote currently supports LSP only when compiled within the Windows Subsystem for Linux.

Emacs users can enable the Asymptote language server protocol by installing lsp-mode using the following procedure:

- Add to the .emacs initialization file:

```
(require 'package)
(add-to-list 'package-archives '("melpa" . "https://melpa.org/packages/") t)
(package-initialize)
```

- Launch emacs and execute

```
M-x package-refresh-contents
M-x package-install
```

  and select lsp-mode.

- Add to the .emacs initialization file:

```
(require 'lsp-mode)
(add-to-list 'lsp-language-id-configuration '(asy-mode . "asymptote"))

(lsp-register-client
 (make-lsp-client :new-connection (lsp-stdio-connection '("asy" "-lsp"))
                  :activation-fn (lsp-activate-on "asymptote")
                  :major-modes '(asy-mode)
                  :server-id 'asyls
                  )
 )
```

- Launch emacs and execute

```
M-x lsp
```

To view all overloads of a function, execute \(lsp-describe-thing-at-point) (via M-x) which opens a help window, or set \(setq lsp-eldoc-render-all t) in your initialization file to show them in the echo area.

16 PostScript to Asymptote

The excellent PostScript editor pstoedit (version 3.50 or later; available from https://sourceforge.net/projects/pstoedit/) can convert PostScript to Asymptote code. This driver includes native clipping, even-odd fill rule, PostScript subpath, and full image support. Here is an example:

```
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.

17 Help

A list of frequently asked questions (FAQ) is maintained at

`https://asymptote.sourceforge.io/FAQ`

Questions on installing and using `Asymptote` that are not addressed in the FAQ should be sent to the `Asymptote` forum:

`https://sourceforge.net/p/asymptote/discussion/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:

`https://sourceforge.net/p/asymptote/patches`

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

`https://sourceforge.net/projects/asymptote/`

If you find a bug in `Asymptote`, please check (if possible) whether the bug is still present in the latest `git` developmental code (see Section 2.8 [Git], page 7) before submitting a bug report. New bugs can be reported at

`https://github.com/vectorgraphics/asymptote/issues`

To see if the bug has already been fixed, check bugs with Status `Closed` and recent lines in

`https://asymptote.sourceforge.io/ChangeLog`

`Asymptote` can be configured with the optional GNU library `libsigsegv`, available from `https://www.gnu.org/software/libsigsegv/`, which allows one to distinguish user-generated `Asymptote` stack overflows (see [stack overflow], page 71) 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).

18 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{});
```

19 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 Orest Shardt, Jesse Frohlich, Michail Vidiassov, Charles Staats, Philippe Ivaldi, Olivier Guibé, Radoslav Marinov, Jeff Samuelson, Chris Savage, Jacques Pienaar, Mark Henning, Steve Melenchuk, Martin Wiebusch, Stefan Knorr, Supakorn "Jamie" Rassameemasmuang, Jacob Skitsko, Joseph Chaumont, and Oliver Cheng. Pedram Emami developed the `Asymptote Web Application` hosted at `https://asymptote.ualberta.ca`:

`https://github.com/vectorgraphics/asymptoteWebApplication`

General Index

!

! 66
!= 63, 66

#

..... 66

%

% 66, 208
%= 67

&

& 24, 66
&& 66

*

* 40, 66
** 66
*= 67

+

+ 40, 66
++ 67
+= 67

—

- 66
-- 12, 67
--- 24
-= 67

.

.. (two dots) 12
.asy 6

/

/ 66
/= 67

:

: 66
:: 24

<

< 66
<= 66

=

== 63, 66

>

> 66
>= 66

?

? 66

[

[=] 68
[] 68
Map.operator [=] 101
Map.operator [] 101
string.operator [] 29, 68

^

^ 66
^= 67
^^ 12

|

| 66
|| 66

2

2D graphs 117

3

3D graphs 161
3D grids 184
3D PostScript 160

A

a4 ... 5
abort ... 31
abs ... 26, 27, 74
abs2 ... 26, 27
absolute ... 4, 152
accel ... 34, 158
access ... 86
acknowledgments ... 217
acos ... 74
aCos ... 74
acosh ... 74
add ... 54, 154
  Map.add ... 102
  Set.add ... 105
addViews ... 156
adjust ... 42
advance ... 89, 97
after,
  SortedSet.after ... 107
Ai ... 75
Ai_deriv ... 75
Airy ... 75
alias ... 63, 79
Align ... 19
aligndir ... 206
all ... 82
Allow ... 49
and ... 24
AND ... 66
angle ... 26
animate ... 4, 61, 193
animation ... 193
annotate ... 193
antialias ... 152, 205
append ... 58, 77
Arc ... 33
arc ... 32, 154
ArcArrow ... 14
ArcArrow3 ... 159
ArcArrows ... 14
ArcArrows3 ... 159
arclength ... 35, 158
arcpoint ... 35
arctime ... 35, 158
arguments ... 71
arithmetic operators ... 66
array ... 30, 78
array iteration ... 25
arrays ... 76
Arrow ... 14
arrow ... 14, 20
arrow keys ... 9, 210
Arrow3 ... 159
arrowbar ... 14
arrowhead ... 15
arrows ... 14
Arrows ... 14
Arrows3 ... 159
as ... 87
ascii ... 30
asin ... 74
aSin ... 74
asinh ... 74
Aspect ... 51
assert ... 31
assignment ... 25
associative array, *See* Map_K_V
asy ... 31, 88
asy-mode ... 7
asy.vim ... 7
asygl ... 5
asyinclude ... 112
Asymptote Web Application ... 1
asymptote.sty ... 112
asymptote.xml ... 7
ASYMPTOTE_CONFIG ... 205
aTan ... 74
atan ... 74
atan2 ... 74
atanh ... 74
atleast ... 24
atOrAfter,
  SortedSet.atOrAfter ... 107
atOrBefore,
  SortedSet.atOrBefore ... 107
attach ... 55, 112, 125
attribution ... 207
autoadjust ... 155
autoimport ... 205
automatic scaling ... 134
autounravel ... 93
axialshade ... 17
axis ... 140, 143, 163, 164
azimuth ... 28

B

babel ... 193
background ... 148, 149
background color ... 53
BackView ... 156
Bar ... 14
Bar3 ... 159
Bars ... 14
Bars3 ... 159
barsize ... 14
basealign ... 43
baseline ... 21
batch mode ... 9
beep ... 32
before,
  SortedSet.before ... 107
BeginArcArrow ... 14
BeginArcArrow3 ... 159
BeginArrow ... 14

BeginArrow3 ... 159
BeginBar ... 14
BeginBar3 ... 159
BeginDotMargin ... 15
BeginDotMargin3 ... 160
BeginMargin ... 15
BeginMargin3 ... 160
BeginPenMargin ... 15
BeginPenMargin2 ... 160
BeginPenMargin3 ... 160
BeginPoint ... 20
Bessel ... 75
bevel ... 197
beveljoin ... 43
Bezier curves ... 23
Bezier patch ... 150
Bezier triangle ... 150
bezulate ... 150, 201
Bi ... 75
Bi_deriv ... 75
Billboard ... 157
binary ... 59
binary format ... 59
binary operators ... 66
binarytree ... 194
black stripes ... 152
Blank ... 14
block.bottom ... 196
block.bottomleft ... 196
block.bottomright ... 196
block.center ... 196
block.draw ... 196
block.left ... 196
block.position ... 196
block.right ... 196
block.top ... 196
block.topleft ... 196
block.topright ... 196
bool ... 25
bool3 ... 26
boolean operators ... 66
Bottom ... 119
BottomTop ... 119
BottomView ... 156
bounding box ... 53
Bounds ... 162
bounds ... 168
box ... 52, 154
bp ... 9
brace ... 33
bracket operators ... 68
break ... 25
breakpoints ... 216
brick ... 46
broken axis ... 137
bsp ... 195
BTreeMap_K_V ... 100
BTreeSet_T ... 107
bug reports ... 215
buildcycle ... 37
Button-1 ... 210
Button-2 ... 210
BWRainbow ... 168
BWRainbow2 ... 168

C

C string ... 29
CAD ... 195
camera ... 155
casts ... 85
cbrt ... 74
cd ... 58
ceil ... 74
center ... 155
Center ... 20
checker ... 46
checkerboard ... 175
Chinese ... 46
choose ... 75
Ci ... 75
Circle ... 32
circle ... 32, 154, 197
circlebarframe ... 185
CJK ... 46
clamped ... 117
clang ... 6
clear ... 60, 216
clip ... 19
CLZ ... 66
cm ... 10
cmd ... 4
cmyk ... 40
colatitude ... 28
collections ... 95
collections.btreemap ... 100
collections.enumerate ... 99
collections.genericpair ... 96
collections.hashmap ... 100
collections.iter ... 89, 96
collections.map ... 100
collections.zip ... 98
collections.zip2 ... 97
color ... 40, 165
coloredNodes ... 191
coloredpath ... 191
coloredSegments ... 191
coloring surfaces ... 149, 173
colorless ... 41
colormap ... 176
colors ... 40
comma ... 59
comma-separated-value mode ... 83
command-line interface ... 211
command-line options ... 5, 202
comment character ... 58

compass directions 11
Compiling from UNIX source 6
complement 78
complexcubicroots 82
concat 79
conditional 25, 66
config 5, 205
configuration file 4, 205
configuring 4
conj 26
constructors 64
containers 95
contains,
  Map.contains 101
  Set.contains 104
context 205
continue 25, 216
contour 177
contour3 195
controls 23, 147
controlSpecifier 39
convert 61
convertOptions 205
Coons shading 18
copy 79
cornerPen 149, 174
Cos 74
cos 74
cosh 74
cputime 76
crop 131
cropping graphs 131
cross 27, 28, 127
crossframe 186
crosshatch 47
csv 82, 83
CTZ 66
cubicroots 82
curl 24, 147
curlSpecifier 39
currentlight 148
currentpen 40
currentprojection 156
curve 189
custom axis types 119
custom mark routine 130
custom tick locations 121
cut 36
cycle 10, 12, 147
cyclic 34, 39, 77, 158
Cyrillic 46

D

dashdotted 42
dashed 42
data types 25
date 31
Debian 3
debugger 216
declaration 25
deconstruct 210
default arguments 71
defaultformat 120
DefaultHead 15
DefaultHead3 159
defaultpen 40, 42, 43, 45, 49
defaultrender 148
deferred drawing 56, 201
Degrees 74
degrees 26, 74
delete 60, 77
  Map.delete 101
  Set.delete 105
description 1
developer modules 201
diagonal 81
diamond 196
dictionary, *See* Map_K_V
diffuse 148
diffusepen 148
dimension 82, 83
dir 6, 27, 28, 34, 158
direction specifier 23
directory 58
dirSpecifier 39
dirtime 35
display 4
do 25
DOSendl 59
DOSnewl 59
dot 16, 27, 28, 81
DotMargin 15
DotMargin3 160
DotMargins 16
DotMargins3 160
dotted 42
double deferred drawing 153
double precision 59
Draw 53
draw 14, 17, 150
drawer 57
drawing commands 14
drawline 187
drawtree 195
dvips 5
dvipsOptions 205
dvisvgm 5, 205
dvisvgmMultipleFiles 210
dvisvgmOptions 205

E

Editing modes ... 7
Ei ... 75
ellipse ... 33
elliptic functions ... 75
else ... 25
emacs ... 7
embed ... 195
Embedded ... 157
emissivepen ... 148
empty ... 50
  Map.empty ... 101
  Set.empty ... 104
EndArcArrow ... 14
EndArcArrow3 ... 159
EndArrow ... 14
EndArrow3 ... 159
EndBar ... 14
EndBar3 ... 159
EndDotMargin ... 16
EndDotMargin3 ... 160
endl ... 59
EndMargin ... 15
EndMargin3 ... 160
EndPenMargin ... 15
EndPenMargin2 ... 160
EndPenMargin3 ... 160
EndPoint ... 20
enumerate ... 99
envelope ... 21
environment variables ... 5
eof ... 60, 82
eol ... 60, 82
EPS ... 20, 205
erase ... 9, 29, 50, 55
erf ... 74
erfc ... 74
error ... 58, 60
error bars ... 128
errorbars ... 127
eval ... 88
evenodd ... 12, 43
exit ... 31, 208, 216
exp ... 74
expi ... 26, 28
explicit ... 85
explicit casts ... 85
expm1 ... 74
exponential integral ... 75
extendcap ... 43
extension ... 36, 199
external ... 195
extract,
  Set.extract ... 106
extrude ... 158
E ... 11, 75

F

fabs ... 74
face ... 160
factorial ... 75
Fedora ... 3
feynman ... 195
fft ... 80
FFTW ... 7
file ... 58, 216
Fill ... 53
fill ... 17
FillDraw ... 52
filldraw ... 17
filloutside ... 17
fillrule ... 43
filltype ... 52
find ... 29, 79
findall ... 79
firstcut ... 37
fit3 ... 154
fitColors ... 176
fixedscaling ... 51
floor ... 74
flowchart ... 196
flush ... 59, 60
fmod ... 74
font ... 45, 46
font encoding ... 46
fontcommand ... 45
fontsize ... 45, 177
for ... 25
  range-based ... 89
format ... 30, 205
forum ... 215
frame ... 50
freshnel0 ... 148
from ... 86
FrontView ... 156
function declarations ... 71
function shading ... 18
Function shading ... 18
functions ... 69, 74
functionshade ... 18
F ... 75

G

gamma ... 74
Gaussrand ... 75
geometry ... 184
get ... 89, 97
  Set.get ... 105
getc ... 58
getint ... 60
getpair ... 60
getreal ... 60
getstring ... 60
gettriple ... 60

git ... 7
globalwrite ... 58, 60
glOptions ... 152, 205
GNU Scientific Library ... 75
gouraudshade ... 18
Gradient ... 168
gradient shading ... 17
graph ... 117
graph3 ... 161
graphic ... 20, 205
graphical user interface ... 210
graphwithderiv ... 132
gray ... 40
grayscale ... 40
Grayscale ... 168
grid ... 46, 135
grid3 ... 184
gs ... 4
gsl ... 75
GSL ... 7
gsOptions ... 205
GUI ... 210
GUI installation ... 210
GUI usage ... 210
guide ... 37
guide3 ... 147

H

hash ... 94
hash function ... 94
HashMap_K_V ... 100
hatch ... 47
Headlamp ... 148
height ... 112
help ... 208, 215, 216
Hermite ... 117
Hermite(splinetype splinetype ... 117
hex ... 30, 41
hexadecimal ... 30, 41
hidden surface removal ... 160
histogram ... 75
history ... 60, 208
historylines ... 208
HookHead ... 15
HookHead3 ... 159
Horizontal ... 197
HTML5 ... 152
htmlviewer ... 4
htmlviewerOptions ... 205
hyperrefOptions ... 205
hypot ... 74

I

i_scaled ... 75
ibl ... 149
iconify ... 152
identity ... 50, 74, 81
identity4 ... 157
if ... 25
IgnoreAspect ... 51
image ... 168, 169
image-based lighting ... 149
ImageMagick ... 5, 193, 205
images ... 168
implicit casts ... 85
implicit linear solver ... 199
implicit scaling ... 69
implicitsurface ... 189
import ... 86
importv3d ... 153
inches ... 10
incircle ... 27
include ... 88
including images ... 20
increasing ... 188
inf ... 26
inheritance ... 65
initialized ... 77
initializers ... 61
inline ... 112
InOutTicks ... 162
input ... 58, 208
input encoding ... 46
insert ... 29, 77
inside ... 37
insphere ... 159
inst ... 216
installation ... 3
int ... 26
integer division ... 66
integral ... 75
integrate ... 75
interactive mode ... 9, 208
interior ... 37
interp ... 66
interpolate ... 185
intersect ... 35, 158, 187
intersection,
  Set ... 106
intersectionpoint ... 36, 158, 187
intersectionpoints ... 36, 158
intersections ... 36, 158
InTicks ... 162
intMax ... 26
intMin ... 26
inverse ... 49, 82
invert ... 157
invisible ... 40
isnan ... 26
isNullValue ... 100, 101

I 75
Iter_array_T 98
Iter_T 89, 96, 97
Iterable 97
Iterable_array_T 98
Iterable_T 96
iterators 89

J

J 74, 75
Japanese 46

K

k_scaled 75
Kate 7
KDE editor 7
keepAspect 51, 112
keyboard bindings: 151
keys 77
  Map.keys 102
keyword 72
keyword-only 72
keywords 71
K 75
Korean 46

L

Label 16, 19, 122
label 11, 19, 157
labelmargin 19
labelpath 198
labelpath3 198
labelx 122
labely 122
Landscape 52
language context 46
language server protocol 213
lastcut 37
lasy-mode 7
latex 205
LaTeX NFSS fonts 45
LaTeX usage 112
latexmk 112
latitude 28
latticeshade 17
layer 14
leastsquares 141, 191
Left 121
LeftRight 121
LeftSide 20
LeftTicks 119, 121
LeftView 156
legend 14, 16, 125
Legendre 75
length 26, 27, 29, 33, 39, 77, 158
letter 5
lexorder 188
libcurl 87
libm routines 74
libsigsegv 71, 215
licenses 207
light 148
limits 131
line 82
line mode 82
linear 117
Linear 134
linecap 43
linejoin 43
lineskip 45
linetype 42
linewidth 42
lmfit 198
locale 30
Log 134
log 74
log-log graph 134
log10 74
log1p 74
log2 graph 136
logarithmic graph 134
logical operators 66
longdashdotted 42
longdashed 42
longitude 28
loop 25
LSP 213
lualatex 205
luatex 205

M

macOS X binary distributions 3
macOS X configuration 6
magick 5, 61, 193, 205
makePair 96
makepen 48
makeQueue 109
map 78
Map 99
Map_K_V 99, 100
  .add 102
  .contains 101
  .delete 101
  .empty 101
  .keys 102
  .operator [=] 101
  .operator [] 101
  .operator iter 101
  .pairs 102
  .size 101
  cast to Iterable 102
  iterating over 101, 102

Margin ... 15
Margin3 ... 160
Margins ... 15
margins ... 154
Margins3 ... 160
mark ... 127
markangle ... 186
marker ... 127
markers ... 185
marknodes ... 127
markuniform ... 127
mask ... 26
material ... 148
math ... 187
mathematical functions ... 74
max ... 37, 50, 80, 158
  SortedSet.max ... 107
maxbound ... 27, 28
maxtile ... 152
maxtimes ... 36
maxviewport ... 152
mesh cell ... 175
metallic ... 148
MetaPost ... 199
MetaPost 24
MetaPost cutafter ... 37
MetaPost cutbefore ... 37
MetaPost pickup ... 40
MetaPost whatever ... 199
Microsoft Windows ... 3
MidArcArrow ... 14
MidArcArrow3 ... 159
MidArrow ... 14
MidArrow3 ... 159
MidPoint ... 20
midpoint ... 35
min ... 37, 50, 80, 158
  SortedSet.min ... 107
minbound ... 27, 28
minipage ... 21
mintimes ... 36
miterjoin ... 43
miterlimit ... 43
mktemp ... 58
mm ... 10
mobile browser ... 152
mode ... 59
monotonic ... 117
mouse ... 210
mouse bindings ... 151
mouse wheel ... 210
Move ... 49
MoveQuiet ... 49
multisample ... 151

N

name ... 59
named arguments ... 71
nan ... 26
natural ... 117
new ... 62, 78
newEmpty,
  Set.newEmpty ... 105
newframe ... 50
newl ... 59
newpage ... 14
newton ... 75
next ... 216
nobasealign ... 43
NoFill ... 53
noglobalread ... 58
nolight ... 148, 174
NoMargin ... 15
NoMargin3 ... 160
none ... 59
None ... 14
normal ... 158
nosafe ... 206
notaknot ... 117
NOT ... 66
NoTicks ... 119
NoTicks3 ... 162
null ... 62
nullpen ... 19, 52, 53
nullValue ... 100, 101
N ... 11

O

obj ... 199
object ... 21
oblique ... 155
obliqueX ... 155
obliqueY ... 155
obliqueZ ... 155
ode ... 199
offset ... 42, 206
OmitTick ... 121
OmitTickInterval ... 121
OmitTickIntervals ... 121
opacity ... 46, 148
open ... 58
OpenGL ... 151
operator ... 67
operator +(...string[] a) ... 30
operator -- ... 117
operator 117
operator == ... 95
operator [=] ... 68
  Map.operator [=] ... 101
operator [] ... 68
  Map.operator [] ... 101
  string.operator [] ... 29, 68

operator answer ... 208
operator cast ... 85
operator ecast ... 86
operator init ... 61, 64
operator iter ... 89, 97
  Map.operator iter ... 101
  Queue_T.operator iter ... 109
  Set.operator iter ... 105
operators ... 65
options ... 202
orient ... 27, 158
orientation ... 52
orthographic ... 155
OR ... 66
outformat ... 151
outprefix ... 52
output ... 58, 205
OutTicks ... 162
overloading functions ... 70
overwrite ... 49
O ... 154

P

pack ... 21
packing ... 73
pad ... 53
pair ... 10, 26
Pair_K_V ... 96
pairs ... 80
  Map.pairs ... 102
palette ... 168, 175
paperheight ... 5
papertype ... 5
paperwidth ... 5
parallelogram ... 196
parametric coordinates ... 165, 174
parametric surface ... 164
parametrized curve ... 131
partialsum ... 188
patch-dependent colors ... 149, 174
path ... 11, 32, 147, 197
path markers ... 127
path[] ... 12
path3 ... 147
patterns ... 46, 188
PBR ... 148
pdflatex ... 205
pdfreloadOptions ... 205
pdfviewer ... 4
pdfviewerOptions ... 205
PDF ... 205
peek,
  Queue_T.peek ... 109
pen ... 40
PenMargin ... 15
PenMargin2 ... 160
PenMargin3 ... 160
PenMargins ... 15
PenMargins2 ... 160
PenMargins3 ... 160
periodic ... 117
periodicOffset ... 117
perl ... 112
perpendicular ... 184
perspective ... 155
physically based rendering ... 148
picture ... 51
picture alignment ... 54
picture.add ... 57
picture.addPoint ... 57
picture.calculateTransform ... 52
picture.fit ... 52
picture.scale ... 52
piecewisestraight ... 34
pixel ... 160
Pl ... 75
plain ... 201
planar ... 150
plane ... 154
planeproject ... 158
point ... 34, 39, 158
polar ... 28
polargraph ... 118
polygon ... 127
pop ... 77
  Queue_T.pop ... 109
popMax,
  SortedSet.popMax ... 107
popMin,
  SortedSet.popMin ... 107
Portrait ... 52
position ... 148, 151
postcontrol ... 35, 158
postfix operators ... 67
P ... 75
PostScript ... 55
PostScript fonts ... 45
PostScript subpath ... 12
pow10 ... 74
prc ... 153
precision ... 60
precontrol ... 34, 158
prefix operators ... 67
private ... 62
programming ... 25
pstoedit ... 199, 214
psviewer ... 4
psviewerOptions ... 205
pt ... 10
public ... 62
push ... 77
  Queue_T.push ... 109
  Set.push ... 105
Python usage ... 209

Q

quadraticroots 82
quarticroots 187
Queue_T,
  .operator iter 109
  .peek 109
  .pop 109
  .push 109
  .size 109
  cast to Iterable 110
  iterating over 109
quick reference 2
quit 9, 208, 216
quote 88
quotient 66

R

RadialShade 53
radialshade 18
RadialShadeDraw 53
radians 74
radius 34, 158
Rainbow 168
rand 75
randMax 75
randString 75
range 91, 97
rational 199
rationalSimplex 199
read 82, 83
reading 58
reading string arrays 82
readline 60
real 26
realDigits 26
realEpsilon 26
realMax 26
realMin 26
realmult 27
realschur 81
rectangle 196
recursion 71
reference 2
reflect 50
Relative 20
relpoint 35
reltime 35
remainder 74
rename 61
render 148, 151, 205
replace 30
resetdefaultpen 49
rest arguments 72
restore 55
restricted 62
return 216
reverse 30, 35, 39, 78, 158
rewind 60
rfind 29
rgb 40, 41
Riemann zeta function 75
Right 121
RightSide 20
RightTicks 119, 121
RightView 156
Rotate 20
rotate 50, 157
Rotate(pair z) 20
round 74
roundcap 43
roundedpath 188
roundjoin 43
roundrectangle 197
RPM 3
runtime imports 88
Russian 46

S

safe 206
sameElementsInOrder 106
save 55
saveline 60
scale 42, 50, 134, 157
Scale 20, 134
scale3 157
scaled graph 133
schur 81
scientific graph 124
scroll 60
search 79
search paths 6
Seascape 52
secondary axis 138
secondaryX 138
secondaryY 138
seconds 31
seek 60
seekeof 60
segment 188
segmentation fault 215
self operators 67
sequence 78
Set,
  != 106
  & 106
  + ... 106
  - 106
  .add 105
  .contains 104
  .delete 105
  .empty 104
  .extract 106
  .get 105
  .newEmpty 105

.operator iter ... 105
.push ... 105
.size ... 104
<= ... 106
== ... 106
>= ... 106
^ ... 106
difference ... 106
intersection ... 106
iterating over ... 105
sameElementsInOrder ... 106
symmetric difference ... 106
union ... 106
settings ... 4, 205
sgn ... 74
shading ... 17
Shift ... 20
shift ... 50, 157
shiftless ... 50
shininess ... 148
shipout ... 52
showtarget ... 155
Si ... 75
signedint ... 59
SimpleHead ... 15
simplex ... 199
simplex2 ... 201
simpson ... 75
Sin ... 74
sin ... 74
single precision ... 59
singleint ... 59
singlereal ... 59
sinh ... 74
SixViews ... 156
SixViewsFR ... 156
SixViewsUS ... 156
size ... 10, 33, 39, 51, 158, 205
Map.size ... 101
Queue_T.size ... 109
Set.size ... 104
size10 ... 199
size11 ... 200
size3 ... 153
Slant ... 20
slant ... 50
sleep ... 32
slice ... 36, 37
slices ... 83
slide ... 200
slope ... 188
slopefield ... 188
smoothcontour3 ... 189
sncndn ... 75
solid ... 42
solids ... 190
solve ... 81
sort ... 79
S ... 11
SortedSet ... 106
.after ... 107
.atOrAfter ... 107
.atOrBefore ... 107
.before ... 107
.max ... 107
.min ... 107
.popMax ... 107
.popMin ... 107
iterating over ... 107
SortedSet_T ... 106
specialty modules ... 193
specular ... 148
specularfactor ... 148
specularpen ... 148
Spline ... 117, 164
split ... 30
sqrt ... 74
squarecap ... 43
srand ... 75
stack overflow ... 71, 215
static ... 91
stats ... 191
stdin ... 58
stdout ... 58
step ... 216
stickframe ... 185
stop ... 216
straight ... 34, 158
Straight ... 117
strftime ... 31
string ... 28, 30
stroke ... 17, 19
strokepath ... 37
strptime ... 31
struct ... 62
structured surface ... 165
structures ... 62
subpath ... 35, 158
subpictures ... 52
substr ... 30
sum ... 80
superpath ... 12
Suppress ... 49
SuppressQuiet ... 49
surface ... 148, 149, 150, 164, 165
surface coloring ... 149, 165, 173
surface parametrization ... 165
surface.domain ... 166
surface.palette ... 175
surface.paramCoords ... 166
surface.paramNormal ... 165
surface.paramPoint ... 165
surfaceVertex ... 173
SVG ... 205
symmetric difference ... 106
system ... 31, 206

syzygy ... 200

T

tab ... 59
tab completion ... 9
tan ... 74
Tan ... 74
tanh ... 74
target ... 155
tell ... 60
template ... 88
tension ... 24, 147
tensionSpecifier ... 39
tensor product shading ... 18
tensorshade ... 18
tessellation ... 150
tex ... 56, 205
TEX fonts ... 45
TEX string ... 28
texcolors ... 177
texcommand ... 5
TeXHead ... 15
TeXHead3 ... 159
texpath ... 5, 21
texpreamble ... 56
texreset ... 56
textbook graph ... 123
tgz ... 3
thick ... 150
thin ... 150
third-party licenses ... 207
this ... 62
three ... 147
ThreeViews ... 156
ThreeViewsFR ... 156
ThreeViewsUS ... 156
tick ... 122
ticks ... 119
Ticks ... 119, 121
tildeframe ... 186
tile ... 46
tilings ... 46
time ... 31, 187
times ... 36
Top ... 119
TopView ... 156
trace ... 216
trailingzero ... 120
transform ... 49, 158
transform3 ... 157
transparency ... 46
transparent ... 149
transpose ... 80
tree ... 200
trembling ... 200
triangle ... 184
triangles ... 150
triangulate ... 183
tridiagonal ... 81
trigonometric integrals ... 75
triple ... 27
TrueMargin ... 16
TrueMargin3 ... 160
tube ... 150, 191
tutorial ... 9
type1cm ... 45
typedef ... 32, 70

U

U3D ... 195
ucyclic ... 166
undefined ... 37
UnFill ... 53
unfill ... 19
uniform ... 79
uninstall ... 8
union ... 106
unique ... 188
unit ... 26, 28
unitbox ... 12, 154
unitcircle ... 12, 154
unitrand ... 75
unitsize ... 11, 51
UNIX binary distributions ... 3
unpacking ... 73
unravel ... 86
unstructured surface ... 165
up ... 155
update ... 58
UpsideDown ... 52
URL ... 87
usepackage ... 56
user coordinates ... 11
user modules ... 117
user-defined operators ... 67
using ... 32, 70
usleep ... 32

V

v3d ... 153, 201
valid ... 89, 97
value ... 187
var ... 62
variable initializers ... 61
vcyclic ... 166
vectorfield ... 143, 144
vectorfield3 ... 167
vectorization ... 82
verbatim ... 55
vertex shading ... 149, 173
vertex-dependent colors ... 149, 173
vertexPen ... 149, 173
Vertical ... 197

Viewport 148
viewportheight 112
viewportmargin 154
viewportsize 154
viewportwidth 112
views 153
vim 7
virtual functions 65
void 25

W

warn 5
W 11
WebGL 152
whatever 36
Wheel 168
wheel mouse 210
while 25
White 148
white-space string delimiter mode 82
width 112
windingnumber 37
word 82
write 58, 83

X

x11colors 177
xasy 210
xaxis3 161
xdr 59
xelatex 205
xequals 121
XEquals 121
xlimits 131
XOR 66
xpart 27, 28
xscale 50
xscale3 157
xtick 122
X 154
XY 157, 158
XYEquals 162
XYZero 162
XZEquals 162
XZero 121
XZZero 162

Y

yaxis3 161
yequals 121
YEquals 119
ylimits 131
ypart 27, 28
yscale 50
yscale3 157
ytick 122
Y 74, 75, 154
YX 158
YZ 158
YZEquals 162
YZero 119
YZZero 162

Z

zaxis3 161
zero_Ai 75
zero_Ai_deriv 75
zero_Bi 75
zero_Bi_deriv 75
zero_J 75
zeroTransform 50
zerowinding 43
zeta 75
zip 97, 98
zpart 28
zscale3 157
Z 154
ZX 158
ZY 158