Asymptote: the Vector Graphics Language

For version 2.90
This file documents Asymptote, version 2.90.

https://asymptote.sourceforge.io


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

1 Description ................................................................. 1

2 Installation ................................................................. 3
   2.1 UNIX binary distributions ........................................ 3
   2.2 MacOS X binary distributions ................................... 3
   2.3 Microsoft Windows ................................................. 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 ............................................................................ 8
   2.9 Building the documentation ....................................... 8
   2.10 Uninstall .............................................................. 8

3 Tutorial ................................................................. 10
   3.1 Drawing in batch mode ............................................. 10
   3.2 Drawing in interactive mode ....................................... 10
   3.3 Figure size .................................................................. 11
   3.4 Labels ........................................................................ 12
   3.5 Paths .......................................................................... 12

4 Drawing commands .......................................................... 15
   4.1 draw .......................................................................... 15
   4.2 fill ............................................................................ 18
   4.3 clip ............................................................................ 20
   4.4 label .......................................................................... 20

5 Bezier curves ................................................................. 24

6 Programming ................................................................. 26
   6.1 Data types ............................................................... 26
   6.2 Paths and guides ......................................................... 33
   6.3 Pens ........................................................................... 40
   6.4 Transforms ............................................................... 50
   6.5 Frames and pictures ................................................... 51
   6.6 Deferred drawing ....................................................... 57
   6.7 Files ........................................................................... 58
   6.8 Variable initializers ..................................................... 61
   6.9 Structures ................................................................. 62
   6.10 Operators ................................................................. 65
      6.10.1 Arithmetic & logical operators ............................... 65
9 Command-line options ....................... 171

10 Interactive mode ............................ 176

11 Graphical User Interface .................... 178
   11.1 GUI installation .......................... 178
   11.2 GUI usage ................................ 178

12 Command-Line Interface .................... 179

13 Language server protocol ................... 180

14 PostScript to Asymptote ..................... 181

15 Help .......................................... 182

16 Debugger ...................................... 183

17 Acknowledgments .............................. 184

Index ............................................ 185
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-quality level of typesetting that \LaTeX provides for scientific text. By default it produces PostScript output, but it can also generate OpenGL, PDF, SVG, WebGL, V3D, and 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

http://asymptote.ualberta.ca

It is also possible to send remote commands to this server via the curl utility (see Chapter 12 [Command-Line Interface], page 179).

A major advantage of Asymptote over other graphics packages is that it is a high-level programming language, as opposed to just a graphics program: it can therefore exploit the best features of the script (command-driven) and graphical-user-interface (GUI) methods for producing figures. The rudimentary GUI xasy included with the package allows one to move script-generated objects around. To make Asymptote accessible to the average user, this GUI is currently being developed into a full-fledged interface that can generate objects directly. However, the script portion of the language is now ready for general use by users who are willing to learn a few simple Asymptote graphics commands (see Chapter 4 [Drawing commands], page 15).

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

The Asymptote vector graphics language provides:

- a standard for typesetting mathematical figures, just as \TeX/LaTeX is the de-facto standard for typesetting equations.
- \LaTeX typesetting of labels, for overall document consistency;
- 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 the stock version of Asymptote is designed for mathematics typesetting needs, one can write Asymptote modules that tailor it to specific applications; for example, a scientific graphing module is available (see Section 8.27 [graph], page 103). 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/local/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 (e.g. Ubuntu, Mepis, Linspire) follow the instructions for compiling from UNIX source (see Section 2.6 [Compiling from UNIX source], page 6). Alternatively, Debian users can install one of Hubert Chan's prebuilt Asymptote binaries from

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 TÎ£X implementation (we recommend https://www.tug.org/texlive or http://www.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 174). The Python 3 interpreter from https://www.python.org is only required if you wish to try out the graphical user interface (see Chapter 11 [GUI], page 178).

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 174). For example, the setting pdfviewer specifies the location of the PDF viewer. Here are the default values of several important configuration variables under UNIX:

```asy
import settings;
pdfviewer="acroread";
htmlviewer="google-chrome";
psviewer="evince";
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/local/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

```asy
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
texpth
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/local/share/asymptote under UNIX and C:\Program Files\Asymptote under MSDOS).
5. The Asymptote examples directory (by default, /usr/local/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.tar.gz in the subdirectory x.xx under

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

execute the commands:

```
gunzip asymptote-x.xx.src.tar.gz
tar -xf asymptote-x.xx.src.tar
```

```
cd asymptote-x.xx
```

By default the system version of the Boehm garbage collector will be used; if it is old we recommend first putting https://github.com/ivmai/bdwgc/releases/download/v8.0.4/gc-8.0.4.tar.gz https://www.ivmaisoft.com/_bin/atomic_ops/libatomic_ops-7.6.10.tar.gz in the Asymptote source directory.

On UNIX platforms (other than MacOS X), we recommend using version 3.2.1 of the freeglut library. To compile freeglut, download

```
https://prdownloads.sourceforge.net/freeglut/freeglut-3.2.1.tar.gz
```

and type (as the root user):

```
gunzip freeglut-3.2.1.tar.gz
tar -xf freeglut-3.2.1.tar
cd freeglut-3.2.1
```

```
cmake -DCMAKE_INSTALL_PREFIX=/usr -DCMAKE_C_FLAGS=-fcommon .
make
make install
```

Then compile Asymptote with the commands

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

Be sure to use GNU make (on non-GNU systems this command may be called gmake). To build the documentation, you may need to install the texinfo-tex package. If you get errors from a broken texinfo or pdftex installation, simply put
Chapter 2: Installation

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 relatively recent version (e.g.
3.4.4 or later). 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, after enabling it by
putting the following lines in their .emacs initialization file, replacing ASYDIR with the
location of the Asymptote system directory (by default, /usr/local/share/asymptote or
C:\Program Files\Asymptote under MSDOS):
(add-to-list 'load-path "ASYDIR")
(autoload 'asy-mode "asy-mode.el" "Asymptote major mode." t)
(autoload 'lasy-mode "asy-mode.el" "hybrid Asymptote/LaTeX major mode." t)
(autoload 'asy-insinuate-latex "asy-mode.el" "Asymptote insinuate LaTeX." t)
(add-to-list 'auto-mode-alist '("\.asy$" . asy-mode))

Particularly useful key bindings in this mode are C-c C-c, which compiles and displays the
current buffer, and the key binding C-c ?, which shows the available function prototypes
for the command at the cursor. For full functionality you should also install the Apache
Software Foundation package two-mode-mode:
https://www.dedasys.com/freesoftware/files/two-mode-mode.el
Once installed, you can use the hybrid mode lazy-mode to edit a LaTeX file containing
embedded Asymptote code (see Chapter 7 [LaTeX usage], page 90). This mode can be en-
abled within latex-mode with the key sequence M-x lazy-mode <RET>. On UNIX systems,
additional keywords will be generated from all asy files in the space-separated list of direc-
tories specified by the environment variable ASYMETOTE_SITEDIR. Further documentation
of asy-mode is available within emacs by pressing the sequence keys C-h f asy-mode <RET>.
Fans of vim can customize vim for Asymptote with

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

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

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

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/local/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
make install
```

To compile without optimization, use the command `make CFLAGS=-g`. On Ubuntu systems, you may need to first install the required dependencies:

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

### 2.9 Building the documentation

Here are instructions for building the documentation:

```
cd doc
make # for both the PDF version doc/asymptote.pdf and the HTML version
cd png
make # for the HTML version only: doc/png/index.html
```

Note that the HTML version cannot be built without executing `make` from doc folder first.

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

### 2.10 Uninstall

To uninstall a Linux x86_64 binary distribution, use the commands

```
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:


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


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 10 [Interactive mode], page 176.

### 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 $(0,0)\rightarrow(1,1)$ to fit a rectangle of width 100.5 bp and height 100.5 bp (the extra 0.5bp accounts for the line width):

```asy
test.asy

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

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

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):

```asy
test.asy

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

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

```asy
test.asy

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 \texttt{unit square}, or equivalently, \texttt{box((0,0),(1,1))}.

To make the user coordinates represent multiples of exactly \(1\text{cm}\):

\begin{verbatim}
unitsize(1cm);
draw(unitsquare);
\end{verbatim}

### 3.4 Labels

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

\begin{verbatim}
size(3cm);
draw(unitsquare);
label("$A$",(0,0),SW);
label("$B$",(1,0),SE);
label("$C$",(1,1),NE);
label("$D$",(0,1),NW);
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{labels.png}
\caption{Labels on a square.}
\end{figure}

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

### 3.5 Paths

This example draws a path that approximates a quarter circle, terminated with an arrow-head:

\begin{verbatim}
size(100,0);
draw((1,0){up}..{left}(0,1),Arrow);
\end{verbatim}
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 24). 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 44). 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 8.29 [three], page 138):

```
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 8.27 [graph], page 103, (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://web.archive.org/web/20201130113133/http://www.piprime.fr/asympote.
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 \texttt{tex} commands; see [\texttt{tex}], page 56), the \texttt{layer} command may be used to start a new PostScript/\LaTeX{} layer:

\begin{verbatim}
definecolor{red}{RGB}{1,0,0}

void layer(picture pic=currentpicture);
\end{verbatim}

The \texttt{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 \texttt{tex} commands are always drawn after the PostScript objects in that layer.

A page break can be generated with the command

\begin{verbatim}
void newpage(picture pic=currentpicture);
\end{verbatim}

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

\begin{verbatim}
definecolor{red}{RGB}{1,0,0}

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);
\end{verbatim}

Draw the path \texttt{g} on the picture \texttt{pic} using pen \texttt{p} for drawing, with optional drawing attributes (Label \texttt{L}, explicit label alignment \texttt{align}, arrows and bars \texttt{arrow} and \texttt{bar}, margins \texttt{margin}, legend, and markers \texttt{marker}). Only one parameter, the path, is required. For convenience, the arguments \texttt{arrow} and \texttt{bar} may be specified in either order. The argument \texttt{legend} is a Label to use in constructing an optional legend entry.

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

The possible values of \texttt{arrow} are \texttt{None}, \texttt{Blank} (which draws no arrows or path), \texttt{BeginArrow}, \texttt{MidArrow}, \texttt{EndArrow} (or equivalently \texttt{Arrow}), and \texttt{Arrows} (which draws an arrow at both ends of the path).

There are also arrow versions with slightly modified default values of \texttt{size} and \texttt{angle} suitable for curved arrows: \texttt{BeginArcArrow}, \texttt{EndArcArrow} (or equivalently \texttt{ArcArrow}), \texttt{MidArcArrow}, and \texttt{ArcArrows}.
For example:
\begin{verbatim}
\draw((0,0)--(1,1),arrow=Arrows);
\end{verbatim}

All of the arrow specifiers except for \texttt{None} and \texttt{Blank} may be given optional arguments, for example:
\begin{verbatim}
\draw((0,0)--(1,1),arrow=Arrow(
arrowhead=HookHead,size=3mm,angle=20,filltype=Draw,position=0.9));
\end{verbatim}

The function \texttt{Arrow} has the signature
\begin{verbatim}
arrowbar Arrow(arrowhead arrowhead=DefaultHead,
    real size=0, real angle=arrowangle,
    filltype filltype=null, position position=EndPoint)
\end{verbatim}

Calling \texttt{Arrow()} returns \texttt{Arrow}, which is an \texttt{arrowbar} object. The parameters are:

- \texttt{arrowhead} can be one of the predefined arrowhead styles \texttt{DefaultHead}, \texttt{SimpleHead}, \texttt{HookHead}, \texttt{TeXHead}.

- \texttt{real size} is the arrowhead size in \texttt{PostScript} coordinates.

  The default arrowhead size when drawn with a pen \texttt{p} is \texttt{arrowsize(p)}.

- \texttt{real angle} is the arrowhead angle in degrees.

- \texttt{filltype filltype} (see \texttt{[filltype]}, page 53),

- (except for \texttt{MidArrow} and \texttt{Arrows}) \texttt{real position} (in the sense of \texttt{point(path p, real t)}) along the path where the tip of the arrow should be placed.

Margins \texttt{margin} can be used to shrink the visible portion of a path by \texttt{labelmargin(p)} to avoid overlap with other drawn objects.

  Typical values of \texttt{margin} are:

\begin{verbatim}
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 \texttt{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 \texttt{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 \texttt{dotfactor*linewidth(p)}.
\end{verbatim}
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 \texttt{sqtx01.asy}.

The legend for a picture \texttt{pic} can be fit and aligned to a frame with the routine:

\begin{verbatim}
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);
\end{verbatim}

Here \texttt{xmargin} and \texttt{ymargin} specify the surrounding \textit{x} and \textit{y} margins, \texttt{perline} specifies the number of entries per line (default 1; 0 means choose this number automatically), \texttt{linelength} specifies the length of the path lines, \texttt{hskip} and \texttt{vskip} specify the line skip (as a multiple of the legend entry size), \texttt{maxwidth} and \texttt{maxheight} specify optional upper limits on the width and height of the resulting legend (0 means unlimited), \texttt{hstretch} and \texttt{vstretch} allow the legend to stretch horizontally or vertically, and \texttt{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 \texttt{dest} using \texttt{add} or \texttt{attach} (see \cite[add about]{add about}, page 54).

To draw a dot, simply draw a path containing a single point. The \texttt{dot} command defined in the module \texttt{plain} draws a dot having a diameter equal to an explicit pen line width or the default line width magnified by \texttt{dotfactor} (6 by default), using the specified \texttt{filltype} (see \cite[filltype]{filltype}, page 53) or \texttt{dotfilltype} (\texttt{Fill} by default):

\begin{verbatim}
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);
\end{verbatim}

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

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

See [pathmarkers], page 113, and Section 8.10 [markers], page 97, 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:

```plaintext
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

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

```plaintext
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 here:

```plaintext
```

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 x 4` pen array `p` and internal control points in the `n x 4` array `z`, is implemented with

```plaintext
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:

```plaintext
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:

```plaintext
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 \cite{texengines}, page 174): the routine

```latex
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 `functionsshade.asy`.

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

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

### 4.3 clip

```latex
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 \cite{fillrule}, page 44). 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 \cite{LaTeXusage}, page 90.

### 4.4 label

```latex
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,

```latex
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

```latex
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 50). 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

```latex
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

```latex
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 44).

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}\n\smallskip \textsc{Andy Hammerlindl, John Bowman, and Tom Prince}\nhttps://asymptote.sourceforge.io\n\"),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

\[
(1 - t)^3 z_0 + 3t(1 - t)^2 c_0 + 3t^2(1 - t)c_1 + t^3 z_1 \quad 0 \leq t \leq 1.
\]

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:

\[\text{draw}((0,0) \ldots \text{controls} (0,100) \text{ and } (100,100) \ldots (100,0));\]

However, it is usually more convenient to just use the \ldots operator, which tells \texttt{Asymptote} to choose its own control points using the algorithms described in Donald Knuth’s monograph, The MetaFontbook, Chapter 14. The user can still customize the guide (or path) by specifying direction, tension, and curl values.
The higher the tension, the straighter the curve is, and the more it approximates a straight line. One can change the spline tension from its default value of 1 to any real value greater than or equal to 0.75 (cf. John D. Hobby, Discrete and Computational Geometry 1, 1986):

\begin{verbatim}
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));
\end{verbatim}

In these examples there is a space between $2$ and \ldots 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):

\begin{verbatim}
draw((100,0){curl 0}..(100,100)..{curl 0}(0,100));
\end{verbatim}

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 71). For example, compare

\begin{verbatim}
draw((0,0){up}..(100,25){right}..(200,0){down});
\end{verbatim}

with

\begin{verbatim}
draw((0,0){up}::(100,25){right}::(200,0){down});
\end{verbatim}

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:

```plaintext
// 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 the Java-style shorthand for iterating over all elements of an array:

```plaintext
// 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:
  ```plaintext
  bool b=true;
  ```
  Defines a boolean variable b and initializes it to the value true. If no initializer is given:
  ```plaintext
  bool b;
  ```
  The value false is assumed.
bool3  an extended boolean type that can take on the values \texttt{true}, \texttt{default}, or \texttt{false}.
A bool3 type can be cast to or from a bool. The default initializer for bool3 is \texttt{default}.

int  an integer type; if no initializer is given, the implicit value 0 is assumed. The minimum allowed value of an integer is \texttt{intMin} and the maximum value is \texttt{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 \texttt{realEpsilon}, with \texttt{realDigits} significant digits. The smallest positive real number is \texttt{realMin} and the largest positive real number is \texttt{realMax}.
The variables \texttt{inf} and \texttt{nan}, along with the function \texttt{bool isnan(real x)} are useful when floating-point exceptions are masked with the \texttt{-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 \texttt{z.x} and \texttt{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:

\begin{verbatim}
pair z=(3,4);
z=(z.x,-z.y);
z=z.x-I*z.y;
z=conj(z);
\end{verbatim}
Here I is the pair (0,1). A number of built-in functions are defined for pairs:

\begin{verbatim}
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 \texttt{length(pair)} is \texttt{abs(pair)}.
The function \texttt{abs2(pair z)} returns \texttt{|z|^2};
real angle(pair z, bool warn=true) returns the angle of z in radians in the interval [-\pi,\pi] or 0 if \texttt{warn} is \texttt{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 \texttt{warn} is \texttt{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 \texttt{angle} measured in radians;
\end{verbatim}
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;

pairrealmult(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.
Equivalent, 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 draw-
ings. 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\ (e.g. for using the \texttt{babel} package, see Section 8.22 \texttt{[babel]}, page 102):

- \" 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 carriage return
- \0-\377 map to corresponding octal byte
- \x0-\xFF map to corresponding hexadecimal byte

The implicit initializer for strings is the empty string "". Strings may be concatenated with the + operator. In the following string functions, position 0 denotes the start of the string:

\begin{verbatim}
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);
\end{verbatim}
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();

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 strftime 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 strftime 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.

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` (see the example in Section 6.12 [Functions], page 68).

### 6.2 Paths and guides

A path is 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 13):

```cpp
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`:

```cpp
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

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

One may also specify the direction explicitly:

```cpp
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

```cpp
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:

```cpp
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

```cpp
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

\[
\text{path brace(pair a, pair b, real amplitude=bracedefaultratio*length(b-a));}
\]

This example illustrates the use of all five guide connectors discussed in Chapter 3 [Tutorial], page 10, and Chapter 5 [Bezier curves], page 24:

\[
\text{size}(300,0);
\]

\[
\text{pair[]} z=\text{new pair}[10];
\]

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

\[
\text{for(int n=3; n <= 9; ++n)}
\]

\[
z[n]=z[n-3]+(200,0);
\]

\[
\text{path p=z[0]..z[1]---z[2]}::\{\text{up}\}z[3]
\]

\[
&z[3]..z[4]--z[5]}::\{\text{up}\}z[6]
\]

\[
&z[6]::z[7]---z[8]..\{\text{up}\}z[9];
\]

\[
\text{draw(p, grey+linewidth(4mm));}
\]

\[
\text{dot(z);}
\]

Here are some useful functions for paths:

\[
\text{int length(path p);} \\
\text{This is the number of (linear or cubic) segments in path p. If p is cyclic, this is the same as the number of nodes in p.}
\]

\[
\text{int size(path p);} \\
\text{This is the number of nodes in the path p. If p is cyclic, this is the same as length(p).}
\]

\[
\text{bool cyclic(path p);} \\
\text{return true iff path p is cyclic.}
\]

\[
\text{bool straight(path p, int i);} \\
\text{return true iff the segment of path p between node i and node i+1 is straight.}
\]

\[
\text{bool piecewisestraight(path p)} \\
\text{return true iff the path p is piecewise straight.}
\]
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 24).
    If t lies outside the range \([0, \text{length}(p)]\), 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 24).

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);
    Return 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 77). 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);
    Return 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 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 ac-
    cording 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 44).

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 44), -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 44).

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(path p);
    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 15). They are used to specify the following drawing attributes:
color, line type, line width, line cap, line join, fill rule, text alignment, font, font size, pattern, overwrite mode, and calligraphic transforms on the pen nib. The default pen used by the drawing routines is called \texttt{currentpen}. This provides the same functionality as the \texttt{MetaPost} command \texttt{pickup}. The implicit initializer for pens is \texttt{defaultpen}.

Pens may be added together with the nonassociative binary operator \texttt{+}. 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 \texttt{dashed+red+green} or \texttt{red+green+dashed} or \texttt{red+dashed+green}. The binary operator \texttt{*} 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 colorspace:

  \begin{itemize}
  \item \texttt{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.
  \item \texttt{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 \([0,1]\).
  \item \texttt{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 \([0,255]\).
  \item \texttt{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 \([0,1]\).
  \item \texttt{pen invisible;}  
      This special pen writes in invisible ink, but adjusts the bounding box as if something had been drawn (like the \texttt{\phantom} command in \LaTeX{}). The function \texttt{bool invisible(pen)} can be used to test whether a pen is invisible.
\end{itemize}

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

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

- A pen can be converted to a hexadecimal string with \texttt{string hex(pen p)};

Various shades and mixtures of the grayscale primary colors \texttt{black} and \texttt{white}, RGB primary colors \texttt{red}, \texttt{green}, and \texttt{blue}, and RGB secondary colors \texttt{cyan}, \texttt{magenta}, and \texttt{yellow} are defined as named colors, along with the CMYK primary colors \texttt{Cyan}, \texttt{Magenta}, \texttt{Yellow}, and \texttt{Black}, in the module \texttt{plain}:
The standard 140 RGB X11 colors can be imported with the command
\texttt{import x11colors;}  
and the standard 68 CMYK \LaTeX{} colors can be imported with the command
\texttt{import texcolors;}  
Note that there is some overlap between these two standards and the definitions of some colors (e.g. \texttt{Green}) actually disagree.

\texttt{Asymptote} also comes with a \texttt{asycolors.sty} \LaTeX{} package that defines to \LaTeX{} CMYK versions of \texttt{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 8.20 [slide], page 102) it may be desirable to specify the color directly to \LaTeX{}. This file can be passed to \LaTeX{} with the \texttt{Asymptote} command
\texttt{usepackage("asycolors");}

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

\begin{verbatim}
pen p=hsv(180,0.5,0.75);
write(p); // ([default], red=0.375, green=0.75, blue=0.75)
hs v q=p;
write(q.h,q.s,q.v); // 180 0.5 0.75
\end{verbatim}

- Line types are specified with the function \texttt{pen linetype(real[] a, real offset=0, bool scale=true, bool adjust=true)}, where \texttt{a} is an array of real array numbers.
The optional parameter \texttt{offset} specifies where in the pattern to begin. The first number specifies how far (if \texttt{scale} is \texttt{true}, in units of the pen line width; otherwise in \texttt{PostScript} units) to draw with the pen on, the second number specifies how far to draw with the pen off, and so on. If \texttt{adjust} is \texttt{true}, these spacings are automatically adjusted by \texttt{Asymptote} to fit the arclength of the path. Here are the predefined line types:

\begin{verbatim}
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});
p pen Dotted(pen p=currentpen) {return linetype(new real[] {0,3})+2*linewidth(p);} 
p pen Dotted=Dotted();
\end{verbatim}

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

- The pen line width is specified in \texttt{PostScript} units with \texttt{pen linewidth(real)}. The default line width is 0.5 bp; this value may be changed with \texttt{defaultpen(pen)}. The line width of a pen is returned by \texttt{real linewidth(pen p=currentpen)}. For convenience, in the module \texttt{plain_pens} we define

\begin{verbatim}
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;}
\end{verbatim}

so that one may set the line width like this:

\begin{verbatim}
defaultpen(2);
p p=red+0.5;
\end{verbatim}

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

\begin{verbatim}
pen squarecap=linecap(0);
pen roundcap=linecap(1);
pen extendcap=linecap(2);
\end{verbatim}

The default line cap, \texttt{roundcap}, may be changed with \texttt{defaultpen(pen)}. The line cap of a pen is returned by \texttt{int linecap(pen p=currentpen)}.

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

\begin{verbatim}
pen miterjoin=linejoin(0);
\end{verbatim}
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 miter limit, 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 −π and −γ are aligned.

  import fontsize;
  import three;

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

  defaultpen(fontsize(100pt));
  dot(0);

  label("acg",0,align=N,basealign);
  label("ace",0,align=N,red);
  label("acg",0,align=S,basealign);
  label("ace",0,align=S,red);
  label("acg",0,align=E,basealign);
Another method for aligning baselines is provided by the `baseline` function (see [baseline], page 22).
The font size is specified in TeX points (1 pt = 1/72.27 inches) with the function \texttt{pen fontsize(real size, real lineskip=1.2*size)}. The default font size, 12pt, may be changed with \texttt{defaultpen(pen)}. Nonstandard font sizes may require inserting \texttt{import fontsize;} at the beginning of the file (this requires the \texttt{type1cm} package available from \url{http://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 \texttt{real fontsize(pen p=currentpen)} and \texttt{real lineskip(pen p=currentpen)}, respectively.

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

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

A nonstandard font command can be generated with \texttt{pen fontcommand(string)}. A convenient interface to the following standard PostScript fonts is also provided:

\begin{itemize}
  \item \texttt{pen AvantGarde(string series="m", string shape="n");}
  \item \texttt{pen Bookman(string series="m", string shape="n");}
  \item \texttt{pen Courier(string series="m", string shape="n");}
  \item \texttt{pen Helvetica(string series="m", string shape="n");}
  \item \texttt{pen NewCenturySchoolBook(string series="m", string shape="n");}
  \item \texttt{pen Palatino(string series="m", string shape="n");}
  \item \texttt{pen TimesRoman(string series="m", string shape="n");}
  \item \texttt{pen ZapfChancery(string series="m", string shape="n");}
  \item \texttt{pen Symbol(string series="m", string shape="n");}
  \item \texttt{pen ZapfDingbats(string series="m", string shape="n");}
\end{itemize}

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 \texttt{cp1251} and Russian language in Cyrillic script (font encoding T2A):

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

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

\url{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 "\texttt{\CJKfamily{kai}}" to the label string):

\begin{verbatim}
texpreamble("\usepackage{CJK}
\end{verbatim}
• The transparency of a pen can be changed with the command:

\begin{verbatim}
pen opacity(real opacity=1, string blend="Compatible");
\end{verbatim}

The opacity can be varied from 0 (fully transparent) to the default value of 1 (opaque), and \texttt{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 https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf. Since PostScript does not support transparency, this feature is only effective with the \texttt{-f pdf} output format option; other formats can be produced from the resulting PDF file with the \texttt{ImageMagick magick} program. Labels are always drawn with an opacity of 1. A simple example of transparent filling is provided in the example file \texttt{transparency.asy}.

• \texttt{PostScript} commands within a \texttt{picture} may be used to create a tiling pattern, identified by the string \texttt{name}, for \texttt{fill} and \texttt{draw} operations by adding it to the global \texttt{PostScript} frame \texttt{currentpatterns}, with optional left-bottom margin \texttt{lb} and right-top margin \texttt{rt}.

\begin{verbatim}
import patterns;
void add(string name, picture pic, pair lb=0, pair rt=0);
\end{verbatim}

To fill or draw using pattern \texttt{name}, use the pen \texttt{pattern("name")}. For example, rectangular tilings can be constructed using the routines \texttt{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 \texttt{patterns}:

\begin{verbatim}
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"));
\end{verbatim}
Hatch patterns can be generated with the routines \texttt{picture hatch(real H=5mm, pair dir=NE, pen p=currentpen)}, \texttt{picture crosshatch(real H=5mm, pen p=currentpen)}:

\begin{verbatim}
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"));
\end{verbatim}

You may need to turn off aliasing in your \texttt{PostScript} viewer for patterns to appear correctly. Custom patterns can easily be constructed, following the examples in module \texttt{patterns}. The tiled pattern can even incorporate shading (see [gradient shading], page 18), as illustrated in this example (not included in the manual because not all printers support \texttt{PostScript} 3):

\begin{verbatim}
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);
\end{verbatim}
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:

```latex
\begin{verbatim}
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);
\end{verbatim}
```

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:

<table>
<thead>
<tr>
<th>overwrite</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow</td>
<td>Allow labels to overwrite one another. This is the default behavior (unless overridden with <code>defaultpen(pen)</code>).</td>
</tr>
<tr>
<td>Suppress</td>
<td>Suppress, with a warning, each label that would overwrite another label.</td>
</tr>
<tr>
<td>SuppressQuiet</td>
<td>Suppress, without warning, each label that would overwrite another label.</td>
</tr>
<tr>
<td>Move</td>
<td>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.</td>
</tr>
</tbody>
</table>
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 \([t.xx \ t.xy \ t.yx \ t.yy \ t.x \ t.y]\).

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 33, 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 \(^\wedge\) 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;

transformyscale(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 \((x,y)\) \(\rightarrow\) \((x+s*y,y)\);

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, pictures are usually more convenient to work with. The implicit initializer for frames is `newframe`. The function `bool empty(frame f)` returns `true` only if the frame \(f\) is empty. 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

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

or prepended with

```c
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 20), is returned by

```c
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:

```c
picture pic;
```

Anonymous pictures can be made by the expression `new picture`. The `size` routine specifies the dimensions of the desired picture:

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

If the \(x\) and \(y\) sizes are both 0, user coordinates will be interpreted as PostScript coordinates. In this case, the transform mapping `pic` to the final output frame is `identity()`.

If exactly one of \(x\) or \(y\) is 0, no size restriction is imposed in that direction; it will be scaled the same as the other direction.

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

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

To make the user coordinates of picture `pic` represent multiples of \(x\) units in the \(x\) direction and \(y\) units in the \(y\) direction, use
void unitsize(picture pic=currentpicture, real x, real y=x);
When nonzero, these x and y values override the corresponding size parameters
of picture pic.
The routine
void size(picture pic=currentpicture, real xsize, real ysize,
    pair min, pair max);
forces the final picture scaling to map the user coordinates box(min,max) to a
region of width xsize and height ysize (when these parameters are nonzero).
Alternatively, calling the routine
transform fixedscaling(picture pic=currentpicture, pair min,
    pair max, pen p=nullpen, 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. The default page orienta-
tion 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 orienta-
tion 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 (e.g. 2D graphs) where only an approximate size estimate for
pic is available, the picture fitting routine
frame pic.scale(real xsize=this.xsize, real ysize=this.ysize,
bool keepAspect=this.keepAspect);

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

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

\[
\text{frame pad(picture pic=currentpicture, real xsize=pic.xsize, real ysize=pic.ysize, filltype filltype=NoFill);}
\]

The functions

\[
\text{pair min(picture pic, user=false);} \\
\text{pair max(picture pic, user=false);} \\
\text{pair size(picture pic, user=false);}
\]

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

The function

\[
\text{pair point(picture pic=currentpicture, pair dir, bool user=true);} \\
\]

is a convenient way of determining the point on the bounding box of \( \text{pic} \) in the direction \( \text{dir} \) relative to its center, ignoring the contributions from fixed-size objects (such as labels and arrowheads). If \text{user} is \text{true} the returned value is in user coordinates, otherwise it is in \text{PostScript} coordinates.

The function

\[
\text{pair truepoint(picture pic=currentpicture, pair dir, bool user=true);} \\
\]

is identical to \text{point}, except that it also accounts for fixed-size objects, using the scaling transform that picture \( \text{pic} \) would have if currently fit to a frame using its default size specification. If \text{user} is \text{true} the returned value is in user coordinates, otherwise it is in \text{PostScript} coordinates.

Sometimes it is useful to draw objects on separate pictures and add one picture to another using the \text{add} function:

\[
\text{void add(picture src, bool group=true, filltype filltype=NoFill, bool above=true);} \\
\text{void add(picture dest, picture src, bool group=true, filltype filltype=NoFill, bool above=true);} \\
\text{void add(picture dest=currentpicture, frame src, pair position=0, bool group=true, filltype filltype=NoFill, bool above=true);} \\
\text{void add(picture dest=currentpicture, frame src, pair position, pair align, bool group=true, filltype filltype=NoFill,}
\]

The first example adds \( \text{src} \) to \( \text{currentpicture} \); the second one adds \( \text{src} \) to \( \text{dest} \). The \text{group} option specifies whether or not the graphical user interface should treat all of the elements of \( \text{src} \) as a single entity (see Chapter 11 [GUI], page 178), \text{filltype} requests optional background filling or clipping, and \text{above} specifies whether to add \( \text{src} \) above or below existing objects.

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

\[
\text{void add(picture src, pair position, bool group=true, filltype filltype=NoFill, bool above=true);} \\
\text{void add(picture dest, picture src, pair position, bool group=true, filltype filltype=NoFill, bool above=true);} \\
\text{void add(picture dest=currentpicture, frame src, pair position=0, bool group=true, filltype filltype=NoFill, bool above=true);} \\
\text{void add(picture dest=currentpicture, frame src, pair position, pair align, bool group=true, filltype filltype=NoFill,}
\]
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 20). 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 114, and [image], page 134. If you want to align three or more subpictures, group them two at a time:

```plaintext
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 15), and labels to be drawn independent of the scaling.

However, because of Asymptote's automatic scaling feature (see Section 3.3 [Figure size], page 11), the translation between user coordinate and PostScript coordinate is not determined until shipout time:

```plaintext
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)`:

```plaintext
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

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

is added to `currentpicture` with

```plaintext
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:

```plaintext
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="", bool check=true, string comment="#", string mode="");

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

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

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

\begin{verbatim}
write(string \textit{s}="", T \textit{x}, suffix suffix=endl \ldots T[]);
write(file \textit{file}, string \textit{s}="", T \textit{x}, suffix suffix=none \ldots T[]);
write(file \textit{file}=stdout, string \textit{s}="", explicit T[] \textit{x} \ldots T[][]);
write(file \textit{file}=stdout, T[][],);
write(suffix suffix=endl);
write(file \textit{file}, suffix suffix=none);
\end{verbatim}

If \texttt{file} is not specified, \texttt{stdout} is used and terminated by default with a newline. If specified, the optional identifying string \textit{s} is written before the data \textit{x}. An arbitrary number of data values may be listed when writing scalars or one-dimensional arrays. The \texttt{suffix} may be one of the following: \texttt{none} (do nothing), \texttt{flush} (output buffered data), \texttt{endl} (terminate with a newline and flush), \texttt{newl} (terminate with a newline), \texttt{DOSendl} (terminate with a DOS newline and flush), \texttt{DOSnewl} (terminate with a DOS newline), \texttt{tab} (terminate with a tab), or \texttt{comma} (terminate with a comma). Here are some simple examples of data output:

\begin{verbatim}
file \textit{fout}=output("test.txt");
write(fout,1); \quad // Writes "1"
write(fout); \quad \quad \quad \quad \quad // Writes a new line
write(fout,"List: ",1,2,3); \quad // Writes "List: 1 2 3"
\end{verbatim}

A file may be opened with \texttt{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 (available on all UNIX platforms). Alternatively, a file may also be opened with \texttt{mode="binary"} to read or write double precision reals and single precision integers in the native (nonportable) machine binary format, or to read the entire file into a string. The virtual member functions \texttt{file singlereal(bool \textit{b}=true)} and \texttt{file singleint(bool \textit{b}=true)} be used to change the precision of real and integer I/O operations, respectively, for an XDR or binary file \texttt{f}. Similarly, the function \texttt{file signedint(bool \textit{b}=true)} can be used to modify the signedness of integer reads and writes for an XDR or binary file \texttt{f}.

The virtual members \texttt{name, mode, singlereal, singleint, and signedint} may be used to query the respective parameters for a given file.

One can test a file for end-of-file with the boolean function \texttt{eof(file)}, end-of-line with \texttt{eol(file)}, and for I/O errors with \texttt{error(file)}. One can flush the output buffers with \texttt{flush(file)}, clear a previous I/O error with \texttt{clear(file)}, and close the file with \texttt{close(file)}. The function \texttt{int precision(file \textit{file}=stdout, int \textit{digits}=0)} sets the number of digits of output precision for \texttt{file} to \texttt{digits}, provided \texttt{digits} is nonzero, and returns the previous precision setting. The function \texttt{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 (i.e. by one less than the current number of display lines). The default value, `settings.scroll=0`, specifies continuous scrolling.

The routines

- `string getstring(string name="", string default="", string prompt="", bool store=true);`
- `int getint(string name="", int default=0, string prompt="", bool store=true);`
- `real getreal(string name="", real default=0, string prompt="", bool store=true);`
- `pair getpair(string name="", pair default=0, string prompt="", bool store=true);`
- `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 176). 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, as in `int x=3;` where the variable `x` is assigned the value 3. As well as literal constants such as 3, arbitrary 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

```plaintext
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 `0=(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

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

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,

```plaintext
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 writable 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 class, 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
deinition of a structure $T$, a variable of type $T$ is initialized to a new instance ($\text{new } T$) by
default. During the definition of the structure, however, variables of type $T$ are initialized
to $\text{null}$ by default. This special behavior is to avoid infinite recursion of creating new
instances in code such as

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

The expression $\text{null}$ 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 $\text{bool alias}(T, T)$ checks to see if two structure references refer to the same
instance of the structure (or both to $\text{null}$). In the example at the beginning of this section,
$\text{alias}(\text{foo, new } T)$ would return false, as $\text{new } T$
creates a new instance of the structure $T$. The boolean operators $==$ and $!=$ are by default
equivalent to $\text{alias}$ and $!\text{alias}$ respectively, but may be overwritten for a particular type
(for example, to do a deep comparison).

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

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

```c
S s; // Initializes s with $\text{new } S$
write(s.f(2)); // Outputs 3
```

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

```c
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 $\text{Person}$ structure:

```c
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:

```cpp
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.

Much like in C++, casting (see Section 6.14 [Casts], page 83) provides for an elegant implementation of structure inheritance, including a virtual function v:

```cpp
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 \( T \text{ interp}(T \ a, T \ b, \text{real} \ t) \) 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 \( \text{int AND}(\text{int}, \text{int}) \), \( \text{int OR}(\text{int}, \text{int}) \), \( \text{int XOR}(\text{int}, \text{int}) \), \( \text{int NOT}(\text{int}) \), \( \text{int CLZ}(\text{int}) \) (count leading zeros), \( \text{int CTZ}(\text{int}) \) (count trailing zeros), \( \text{int popcount}(\text{int}) \) (count bits populated by ones), and \( \text{int bitreverse}(\text{int} \ a, \text{int} \ bits) \) (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,

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

is equivalent to the code

```plaintext
int i=1;
i=i+2;
i 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 \texttt{operator} to define or redefine operators on structures and built-in types:

\(- \ + \ * \ / \ \% \ ^\ \& \ | \ ^^ \ . \ :: \ --- \ ++
\<<< \ >> \ $ \ $$ \ @ \ @@ \ <>\n```

The operators on the second line have precedence one higher than the boolean operators \(<, >, \leq, \text{and} >\).

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

```plaintext
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);

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:

```plaintext
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:

```plaintext
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:

```plaintext
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.
```

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

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

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

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

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

g(); // writes "hi"

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

void f() {write("overloaded");};
f(); // writes "overloaded"
g(); // writes "bye"

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

void f(bool b);

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

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

g(true); // Writes true, then writes false.

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

Functions are allowed to call themselves recursively. As in C++, infinite nested recursion will generate a stack overflow (reported as a segmentation fault, unless a fully working version of the GNU library *libsigsegv* (e.g. 2.4 or later) is installed at configuration time).

### 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 83) one can often avoid ambiguities by ordering function arguments from the simplest to the most complicated. For example, given

```asymptote
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,

```asymptote
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,

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

is equivalent to the statements

```asymptote
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

```asymptote
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

```asymptote
int f(int x=3, int x()) {
  return x+x();
}
```
then \( f(x=\text{seven}) \) would be illegal, even though \( f(\text{seven}) \) obviously would be allowed.

### 6.12.3 Rest arguments

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

```plaintext
// 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
```

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

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

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

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

One can even combine normal arguments with rest arguments:

```plaintext
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:

```plaintext
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`.

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

The overloading resolution in \texttt{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.

\begin{verbatim}
int f(path g);
int f(guide g);
f((0,0)--(100,100)); // matches the second; the argument is a guide
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}

### 6.12.4 Mathematical functions

\texttt{Asymptote} has built-in versions of the standard \texttt{libm} mathematical real(real) functions \texttt{sin}, \texttt{cos}, \texttt{tan}, \texttt{asin}, \texttt{acos}, \texttt{atan}, \texttt{exp}, \texttt{log}, \texttt{pow10}, \texttt{log10}, \texttt{sinh}, \texttt{cosh}, \texttt{tanh}, \texttt{asinh}, \texttt{acosh}, \texttt{atanh}, \texttt{sqrt}, \texttt{cbrt}, \texttt{fabs}, \texttt{expm1}, \texttt{log1p}, as well as the identity function \texttt{identity}. \texttt{Asymptote} also defines the order \texttt{n} Bessel functions of the first kind \texttt{Jn(int n, real)} and second kind \texttt{Yn(int n, real)}, as well as the gamma function \texttt{gamma}, the error function \texttt{erf}, and the complementary error function \texttt{erfc}. The standard real(real, real) functions \texttt{atan2}, \texttt{hypot}, \texttt{fmod}, \texttt{remainder} are also included.

The functions \texttt{degrees(real radians)} and \texttt{radians(real degrees)} can be used to convert between radians and degrees. The function \texttt{Degrees(real radians)} returns the angle in degrees in the interval \([0,360)\). For convenience, \texttt{Asymptote} defines variants \texttt{Sin}, \texttt{Cos}, \texttt{Tan}, \texttt{aSin}, \texttt{aCos}, and \texttt{aTan} of the standard trigonometric functions that use degrees rather than radians. We also define complex versions of the \texttt{sqrt}, \texttt{sin}, \texttt{cos}, \texttt{exp}, \texttt{log}, and \texttt{gamma} functions.
The functions \texttt{floor}, \texttt{ceil}, and \texttt{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 \texttt{Floor}, \texttt{Ceil}, and \texttt{Round} are respectively similar, except that if the result cannot be converted to a valid int, they return \texttt{intMax} for positive arguments and \texttt{intMin} for negative arguments, rather than generating an integer overflow. We also define a function \texttt{sgn}, which returns the sign of its real argument as an integer (-1, 0, or 1).

There is an \texttt{abs(int)} function, as well as an \texttt{abs(real)} function (equivalent to \texttt{fabs(real)}), an \texttt{abs(pair)} function (equivalent to \texttt{length(pair)}).

Random numbers can be seeded with \texttt{srand(int)} and generated with the \texttt{int rand()} function, which returns a random integer between 0 and the integer \texttt{randMax}. The \texttt{unitrand()} function returns a random number uniformly distributed in the interval [0,1]. A Gaussian random number generator \texttt{Gaussrand} and a collection of statistics routines, including \texttt{histogram}, are provided in the module \texttt{stats}. The functions \texttt{factorial(int n)}, which returns \(n!\), and \texttt{choose(int n, int k)}, which returns \(n!/(k!(n-k)!}\), are also defined.

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

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

Asymptote also provides a few general purpose numerical routines:

```asymptote
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 \texttt{verbose=true}. If the iteration fails after the maximum allowed number of loops (\texttt{iterations}), \texttt{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 \texttt{verbose=true}. If the iteration fails after the maximum allowed number of loops (\texttt{iterations}), \texttt{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.
Internally, `operator init` implicitly defines a constructor function `Person(string, string)` as follows, where `args` is `string firstname, string lastname` in this case:

```cpp
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:

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

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

```cpp
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`.

```cpp
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

```cpp
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 Java-style shorthand exists for iterating over all elements of an array; see [array iteration], page 26.

The declaration

```cpp
real[] A;
```
initializes \texttt{A} to be an empty (zero-length) array. Empty arrays should be distinguished from null arrays. If we say

\begin{verbatim}
real[] A=null;
\end{verbatim}

then \texttt{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:

\begin{verbatim}
real[] A={0,1,2};
\end{verbatim}

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

Every array \texttt{A} of type \texttt{T[]} has the virtual members

- \texttt{int length},
- \texttt{bool cyclic},
- \texttt{int[]} \texttt{keys},
- \texttt{T push(T x)},
- \texttt{void append(T[] a)},
- \texttt{T pop()},
- \texttt{void insert(int i ... T[] x)},
- \texttt{void delete(int i, int j=i)},
- \texttt{void delete()}, and
- \texttt{bool initialized(int n)}.

The member \texttt{A.length} evaluates to the length of the array. Setting \texttt{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 \texttt{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 \texttt{n} with all entries initialized, \texttt{A.keys} evaluates to \{0,1,\ldots,n-1\}. A new keys array is produced each time \texttt{A.keys} is evaluated.

The functions \texttt{A.push} and \texttt{A.append} append their arguments onto the end of the array, while \texttt{A.insert(int i ... T[] x)} inserts \texttt{x} into the array at index \texttt{i}. For convenience \texttt{A.push} returns the pushed item. The function \texttt{A.pop()} pops and returns the last element, while \texttt{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, \texttt{A.delete()} provides a convenient way of deleting all elements of \texttt{A}. The routine \texttt{A.initialized(int n)} can be used to examine whether the element at index \texttt{n} is initialized. Like all \texttt{Asymptote} functions, \texttt{push}, \texttt{append}, \texttt{pop}, \texttt{insert}, \texttt{delete}, and \texttt{initialized} can be "pulled off" of the array and used on their own. For example,

\begin{verbatim}
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;
\end{verbatim}
f(3); // A now contains {1,2,1,2,3}.
int g() = A.pop;
write(g()); // Outputs 3.
A.delete(0); // A now contains {2,1,2}.
A.delete(0,1); // A now contains {2}.
A.insert(1,3); // A now contains {2,3}.
A.insert(1 .. A); // A now contains {2,2,3,3}.
A.insert(2,4,5); // A now contains {2,2,4,5,3,3}.

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

real a, A[];

This declares a to be real and implicitly declares A to be of type real[].

In the following list of built-in array functions, T represents a generic type. Note that the internal functions alias, 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);

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

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

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,\ldots,0} (otherwise returns a null array);

int[] complement(int[] a, int n)
returns the complement of the integer array a in \{0,1,2,\ldots,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 k,
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 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);
\[
f = \text{fft}(g, 1);
\text{write}();
\text{write}(f/n);
\]

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

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

\[
\text{realschur schur}(\text{real}[][] \text{ a})
\text{ returns a struct } \text{realschur}
\text{ containing a unitary matrix } \text{U}
\text{ and a quasitriangular matrix } \text{T}
\text{ such that } \text{a} = \text{U} \times \text{T} \times \text{transpose(U)};
\]

\[
\text{schur schur}(\text{pair}[][] \text{ a})
\text{ returns a struct } \text{schur}
\text{ containing a unitary matrix } \text{U}
\text{ and a triangular matrix } \text{T}
\text{ such that } \text{a} = \text{U} \times \text{T} \times \text{conj(transpose(U))};
\]

\[
\text{real dot}(\text{real}[] \text{ a}, \text{real}[] \text{ b})
\text{ returns the dot product of the vectors } \text{a}
\text{ and } \text{b};
\]

\[
\text{pair dot}(\text{pair}[] \text{ a}, \text{pair}[] \text{ b})
\text{ returns the complex dot product } \text{sum(a*conj(b))}
\text{ of the vectors } \text{a}
\text{ and } \text{b};
\]

\[
\text{real}[] \text{ tridiagonal}(\text{real}[] \text{ a}, \text{real}[] \text{ b}, \text{real}[] \text{ c}, \text{real}[] \text{ f});
\text{ Solve the periodic tridiagonal problem } \text{Lx} = \text{f}
\text{ and return the solution } \text{x}, \text{ where }\]
\text{ f is an } n \times n \text{ matrix} \text{ and } \text{L is the } n \times n \text{ matrix}
\[
\begin{bmatrix}
\text{b[0]} & \text{c[0]} & \text{a[0]} \\
\text{a[1]} & \text{b[1]} & \text{c[1]} \\
\text{a[2]} & \text{b[2]} & \text{c[2]} \\
\vdots & \ \\
\text{c[n-1]} & \text{a[n-1]} & \text{b[n-1]}
\end{bmatrix}
\]
\text{ For Dirichlet boundary conditions (denoted here by } u[-1] \text{ and } u[n], \text{ replace}
\text{ f[0] by } f[0]-a[0]u[-1] \text{ and } f[n-1]-c[n-1]u[n]; \text{ then set } a[0]=c[n-1]=0;
\]

\[
\text{real}[] \text{ solve}(\text{real}[] [] \text{ a}, \text{real}[] \text{ b}, \text{bool } \text{warn=true})
\text{ Solve the linear equation } \text{ax} = \text{b}
\text{ by LU decomposition and return the solution } \text{x}, \text{ where } \text{a}
\text{ is an } n \times n \text{ matrix and } \text{b} \text{ is an array of length } n. \text{ For example:}
\]
import math;
\text{real}[] [] \text{ a}={1,-2,3,0},{4,-5,6,2},{-7,-8,10,5},{1,50,1,-2};
\text{real}[] \text{ b}={7,19,33,3};
\text{real}[] \text{ x}=\text{solve} (\text{a}, \text{b});
\text{write}(\text{a}); \text{write}();
\text{write}(\text{b}); \text{write}();
\text{write}(\text{x}); \text{write}();
\text{write}(\text{a*x});
\text{If } \text{a} \text{ is a singular matrix and } \text{warn is false, return an empty array. If the matrix}
\text{ a is tridiagonal, the routine } \text{tridiagonal}
\text{ provides a more efficient algorithm}
\text{ (see [tridiagonal], page 79);}
real[] solve(real[] a, real[] b, bool warn=true)
Solve the linear equation $ax = b$ 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.

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 members dimension, line, csv, word, and read of a file are useful for reading arrays. For example, if line mode is set with file line(bool b=true), then reading will stop once the end of the line is reached instead:

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:

```cpp
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:

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

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

```cpp
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:

```cpp
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:

```cpp
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,

```cpp
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:

```cpp
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:

```java
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:

```java
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 $x[m:n]=\text{copy}(x)$. 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:

```cpp
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:

```cpp
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 \texttt{bool initialized(T)}:

```cpp
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:

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

```cpp
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

```cpp
real operator cast(int x) {return x*100;}
```
is equivalent to an infinite recursion, since the result $x \times 100$ needs itself to be cast from an integer to a real. Instead, we want to use the standard conversion of int to real:

```cpp
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

```cpp
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

```cpp
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:

```cpp
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:

```cpp
from graph access *;
```

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

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

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

Alternatively, one can unravel selective fields:

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

The command

```cpp
import graph;
```
is a convenient abbreviation for the commands

```plaintext
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`:

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

The command

```plaintext
import graph as graph2d;
```

is a convenient abbreviation for the commands

```plaintext
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 name contains nonalphanumeric characters, enclose it with quotation marks:

```plaintext
access "/usr/local/share/asymptote/graph.asy" as graph;
from "/usr/local/share/asymptote/graph.asy" access axes;
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:

```plaintext
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:

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

To conditionally execute an array of asy files, use

```plaintext
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:

```cpp
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 174).

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

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

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

```cpp
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:

```cpp
include graph;
include "/usr/local/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

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

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

### 6.15.1 Templated imports

**Warning:** This feature is experimental: it has known issues and its behavior may change in the future.

In *Asymptote*, it is possible to create modules that must have one or more types 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,

```cpp
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

```cpp
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

```plaintext
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 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

```plaintext
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

```plaintext
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,

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

creates one object `z` for each object of type `A` created.
In this example,

```c
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:

```c
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:

```c
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:

```c
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.
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

dflatex latexusage
asy latexusage-*.asy
dflatex latexusage

To switch between using inline Asymptote code with latex and dflatex 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

http://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 \texttt{latex} \TeX{} engine is used with the \texttt{inline} option, labels might not show up in DVI viewers that cannot handle raw PostScript code. One can use \texttt{dvips/dvipdf} to produce PostScript/PDF output (we recommend using the modified version of \texttt{dvipdf} in the \texttt{Asymptote} patches directory, which accepts the \texttt{dvips -z hyperdvi} option).

Here now is \texttt{latexusage.tex}:

\begin{verbatim}
\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.
settings.prc=true;
import three;
usepackage("bm");
texpreamble("\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;
\end{asy}
\end{center}
\end{figure}
\end{verbatim}
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);

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("$B$",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{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 \verb+\LaTeX\ \verb+inline=true+, 3D figures should specify
\verb+\LaTeX\ \verb+attach=true+ option as well as the
Chapter 7: \LaTeX{} usage

\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 \texttt{latexmk}, 3D figures should specify \texttt{inline=true}. It is sometimes desirable to embed 3D files as annotated attachments; this requires the \texttt{attach=true} option as well as the \texttt{attachfile2} \LaTeX{} package.

One can also scale the figure to the full line width:
8 Base modules

Asymptote currently ships with the following base modules:

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

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

8.2 simplex
This module solves the general linear programming problem using the simplex method.

8.3 simplex2
This module solves a special case of the two-variable linear programming problem used by the module plain for automatic sizing of pictures.

8.4 math
This module extends Asymptote’s mathematical capabilities with useful functions such as

void drawline(picture pic=currentpicture, pair P, pair Q, pen p=currentpen);

draw the visible portion of the (infinite) line going through P and Q, without altering the size of picture pic, using pen p.

real intersect(triple P, triple Q, triple n, triple Z);

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 nth 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 nth 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 nth 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 \(y=z.y\).

\[\text{int}[	ext{int}][	ext{int}] \text{segment(bool}][	ext{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 \(\text{strict=false}\), whether \(i > j\) implies \(a[i] >= a[j]\), or if \(\text{strict=true}\), whether \(i > j\) 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 \(b\).

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

8.6 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:


8.7 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.
8.8 stats
This module implements a Gaussian random number generator and a collection of statistics routines, including histogram and leastsquares.

8.9 patterns
This module implements PostScript tiling patterns and includes several convenient pattern generation routines.

8.10 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 114) 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 $AOB$:

```plaintext
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`. 
8.11 map
This module creates a struct parameterized by the types specified in strings key and value, mapping keys to values with a specified default:

```plaintext
import map;

mapTemplate(name="map",key="string",value="int",default="-1");

map M;

M.add("z",2);
M.add("a",3);
M.add("d",4);
write(M.lookup("a"));
write(M.lookup("y"));
```

8.12 tree
This module implements an example of a dynamic binary search tree.

8.13 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`:

```plaintext
import binarytree;

picture pic,pic2;

binarytree bt=binarytree(1,2,4,nil,5,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);

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

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

8.16 feynman
This module, contributed by Martin Wiebusch, is useful for drawing Feynman diagrams, as illustrated by the examples eetomumu.asy and fermi.asy.
8.17 roundedpath

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

8.18 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

http://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 90). The member function

```latex
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:


8.19 embed

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

http://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 [animate], page 101).

An example of embedding U3D code is provided in the file `embeddedu3d`.
8.20 **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`.

8.21 **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 37) 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.

8.22 **babel**

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

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

8.23 **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

```asymptote
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.

8.24 **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`.

8.25 **annotate**

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

```asymptote
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.
8.26 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.

8.27 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:

- \texttt{guide\ graph(picture pic=\texttt{currentpicture}, real f(real), real a, real b, int n=n\texttt{graph}, real T(real)=identity, \texttt{interpolate join=}operator --);} \\
  \texttt{guide[] graph(picture pic=\texttt{currentpicture}, real f(real), real a, real b, int n=n\texttt{graph}, real T(real)=identity, bool3 cond(real), \texttt{interpolate join=}operator --);} \\

Returns a graph using the scaling information for picture \texttt{pic} (see [automatic scaling], page 120) of the function \texttt{f} on the interval \([T(a),\texttt{T(b)}]\), sampling at \(n\) points evenly spaced in \([a,b]\), optionally restricted by the bool3 function \texttt{cond} on \([a,b]\). If \texttt{cond} is:

- \texttt{true}, the point is added to the existing guide;
- \texttt{default}, the point is added to a new guide;
- \texttt{false}, the point is omitted and a new guide is begun.

The points are connected using the interpolation specified by \texttt{join}:

- \texttt{operator --} (linear interpolation; the abbreviation \texttt{Straight} is also accepted);
- \texttt{operator ..} (piecewise Bezier cubic spline interpolation; the abbreviation \texttt{Spline} is also accepted);
- \texttt{linear} (linear interpolation),
- \texttt{Hermite} (standard cubic spline interpolation using boundary condition \texttt{notaknot}, \texttt{natural}, \texttt{periodic}, \texttt{clamped(real slopea, real slopeb)}), or \texttt{monotonic}. The abbreviation \texttt{Hermite} is equivalent to \texttt{Hermite(notaknot)} for nonperiodic data and \texttt{Hermite(periodic)} for periodic data).

- \texttt{guide\ graph(picture pic=\texttt{currentpicture}, real x(real), real y(real), real a, real b, int n=n\texttt{graph}, real T(real)=identity, \texttt{interpolate join=}operator --);} \\
  \texttt{guide[] graph(picture pic=\texttt{currentpicture}, real x(real), real y(real), real a, real b, int n=n\texttt{graph}, real T(real)=identity, bool3 cond(real), \texttt{interpolate join=}operator --);} \\

Returns a graph using the scaling information for picture \texttt{pic} of the parametrized function \((x(t),y(t))\) for \(t\) in the interval \([T(a),\texttt{T(b)}]\), sampling at \(n\) points evenly spaced in \([a,b]\), optionally restricted by the bool3 function \texttt{cond} on \([a,b]\), using the given interpolation type.
Chapter 8: Base modules

- \texttt{guide graph(picture pic=currentpicture, pair z(real), real a, real b,}
  \texttt{ int n=ngraph, real T(real)=identity,}
  \texttt{ interpolate join=operator --);}
- \texttt{guide[] graph(picture pic=currentpicture, pair z(real), real a, real b,}
  \texttt{ int n=ngraph, real T(real)=identity, bool3 cond(real),}
  \texttt{ interpolate join=operator --);}\

Returns a graph using the scaling information for picture \texttt{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 \texttt{bool3} function \texttt{cond} on \([a, b]\), using the given interpolation type.

- \texttt{guide graph(picture pic=currentpicture, pair[] z,}
  \texttt{ interpolate join=operator --);}
- \texttt{guide[] graph(picture pic=currentpicture, pair[] z, bool3[] cond,}
  \texttt{ interpolate join=operator --);}\

Returns a graph using the scaling information for picture \texttt{pic} of the elements of the array \(z\), optionally restricted to those indices for which the elements of the boolean array \texttt{cond} are \texttt{true}, using the given interpolation type.

- \texttt{guide graph(picture pic=currentpicture, real[] x, real[] y,}
  \texttt{ interpolate join=operator --);}
- \texttt{guide[] graph(picture pic=currentpicture, real[] x, real[] y,}
  \texttt{ bool3[] cond, interpolate join=operator --);}\

Returns a graph using the scaling information for picture \texttt{pic} of the elements of the arrays \((x,y)\), optionally restricted to those indices for which the elements of the boolean array \texttt{cond} are \texttt{true}, using the given interpolation type.

- \texttt{guide polargraph(picture pic=currentpicture, real f(real), real a,}
  \texttt{ real b, int n=ngraph, interpolate join=operator --);}\

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

- \texttt{guide polargraph(picture pic=currentpicture, real[] r, real[] theta,}
  \texttt{ interpolate join=operator--);}\

Returns a polar-coordinate graph using the scaling information for picture \texttt{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:

- \texttt{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 \( \text{pic} \) from \( x=\text{xmin} \) to \( x=\text{xmax} \) using pen \( p \), optionally labelling it with Label \( \text{L} \). The relative label location along the axis (a real number from \([0,1]\)) defaults to 1 (see [Label], page 20), so that the label is drawn at the end of the axis. An infinite value of \( \text{xmin} \) or \( \text{xmax} \) specifies that the corresponding axis limit will be automatically determined from the picture limits. The optional \( \text{arrow} \) argument takes the same values as in the \text{draw} command (see [arrows], page 15). The axis is drawn before any existing objects in \( \text{pic} \) unless \( \text{above}=true \). The axis placement is determined by one of the following \text{axis} types:

\text{YZero}(\text{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 \text{extend}=false.

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

\text{Bottom}(\text{bool extend}=false)
  
Request a bottom axis.

\text{Top}(\text{bool extend}=false)
  
Request a top axis.

\text{BottomTop}(\text{bool extend}=false)
  
Request a bottom and top axis.

Custom axis types can be created by following the examples in the module \text{graph.asy}. One can easily override the default values for the standard axis types:

\text{import graph;}

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

\text{YZero=YZero();}

The default tick option is \text{NoTicks}. The options \text{LeftTicks}, \text{RightTicks}, or \text{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 \text{LeftTicks}(\text{Label format}="", \text{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 \hfill31. 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 120), 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 20). An infinite value of `ymin` or `ymax` specifies that the corresponding axis limit will be automatically determined from the picture limits. The optional `arrow` argument takes the same values as in the `draw` command (see [arrows], page 15). 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=false)**
- 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 11 [GUI], page 178). If an \texttt{UnFill(real xmargin=0, real ymargin=xmargin)} or \texttt{Fill(pen)} option is specified to \texttt{add}, the legend will obscure any underlying objects. Here we illustrate how to clip the portion of the picture covered by a label:

```plaintext
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`:

```latex
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:

```latex
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));
```
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.

```plaintext
class graph {
    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.

```plaintext
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

```plaintext
marker marker(path g, markroutine markroutine=marknodes,
pen p=currentpen, filltype filltype=NoFill,
bool above=true);
```
using the path `unitcircle` (see [filltype], page 53), are drawn below each node. Any
frame can be converted to a marker, using

```latex
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:

```latex
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:

```latex
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.

```latex
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:

```plaintext
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);
```
Chapter 8: Base modules

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

```plaintext
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). Ex-
isting 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 \( f_{\text{prime}} \):

\[
\text{unitsize}(2\text{cm});
\text{import}\ \text{graph};
\text{pair } F(\text{real } t) \{
    \text{return } (1.3*t,-4.5*t^2+3.0*t+1.0);
\}
\text{pair } F\text{prime}(\text{real } t) \{
    \text{return } (1.3,-9.0*t+3.0);
\}
\text{path } g=\text{graphwithderiv}(F,F\text{prime},0,0.9,4);
\text{dot}(g,\text{red});
\text{draw}(g,\text{arrow=Arrow(TeXHead)});
\]

The next example illustrates how one can extract a common axis scaling factor.

\[
\text{import}\ \text{graph};
\text{axiscoverage}=0.9;
\text{size}(200,\text{IgnoreAspect});
\text{real[] } x=\{-1e-11,1e-11\};
\text{real[] } y=\{0,1e6\};
\text{real } xscale=\text{round}(\text{log10}(\text{max}(x)));
\text{real } yscale=\text{round}(\text{log10}(\text{max}(y)))-1;
\text{draw}(\text{graph}(x*10^{-xscale},y*10^{-yscale}),\text{red});
\text{xaxis("x/10\{-+(string) \text{xscale+}\}",BottomTop,LeftTicks);}
\text{yaxis("y/10\{-+(string) \text{yscale+}\}",LeftRight,RightTicks(\text{trailingzero})});
\]
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 (e.g. for a depth axis, Linear(-1) requests axis reversal).

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

import graph;

size(200,200,IgnoreAspect);

real f(real t) {return 1/t;}
scale(Log,Log);

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

//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:

```plaintext
import graph;

size(300,175,IgnoreAspect);
scale(Log,Log);
draw(graph(identity,5,20));
xlims(5,20);
ylim(1,100);
xaxis("$M/M_\odot$",BottomTop,LeftTicks(DefaultFormat, new real[] {6,10,12,14,16,18}));
yaxis("$\nu_{\text{upp}}$ [Hz]",LeftRight,RightTicks(DefaultFormat));
```

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

```plaintext
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\{\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);
  ylims(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 \texttt{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);
11. Here is an example of reading column data in from a file and a least-squares fit, using the \texttt{stats} module.

\begin{verbatim}
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];

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;
\end{verbatim}
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.

```plaintext
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

```plaintext
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:

```plaintext
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.
Achnanthes minutissima Kuetzing
Anomoeoneis vitrea (Grunow) Ross
Asterionella forbesii Hassall
Tabellaria flocculosa (Rehb) Kuetzing
Fragilaria cf. tenera
Chaetoceros muelleri/elmorei cysts
Aulacoseira spp.
Fragilaria capucina var. vaucheriae (Kuetzing)
Achnanthes minutissima Kuetzing
0
100
200
300
400 sediment depth (cm)
1961 1950 1940 1930 1920 1915 1910 1888
1763
1726

0
1
2
3
4
5
6
7
8
9
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
10
M2
20
M1
30
40
Initial no. of mosquitoes per bird ($S_{Mo}/N_{Bo}$)

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
8.28 palette

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

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

- `pen[] Rainbow(int NColors=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`:

```asy```
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 120). If `divs > 1`, the palette is quantized to `divs − 1` values. A `bounds` structure for the function values is returned:

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

This information can be used for generating an optional palette bar. The palette color space corresponds to a range of values specified by the argument `range`, which can be `Full`, `Automatic`, or an explicit range `Range(real min, real max)`. Here `Full` specifies a range varying from the minimum to maximum values of the function over the sampling interval, while `Automatic` selects "nice" limits. The examples `fillcontour.asy` and `imagecontour.asy` illustrate how level sets (contour lines) can be drawn on a color density plot (see Section 8.36 [contour], page 162).
A color density plot can also be generated from an explicit real[][], array data:

```plaintext
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

```plaintext
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

```plaintext
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 105) that takes the following arguments:

```plaintext
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:

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

```plaintext
import graph;
import palette;
```

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

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

```plaintext
pen[] Palette=BWRainbow();
```
Here is an example that uses logarithmic scaling of the function values:

```plaintext
import graph;
import palette;

def f(x, y):
    return 0.9 * 10^(2 * sin(x/5 + 2*y^0.25)) + 0.1 * (1 + cos(10*log(y)));

scale(Linear, Log, Log);

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

xaxis("$x$", BottomTop, LeftTicks, above=true);
yaxis("$y$", LeftRight, RightTicks, above=true);
```
One can also draw an image directly from a two-dimensional pen array or a function `pen f(int, int)`: 

```plaintext
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:

```plaintext
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 \texttt{palette} also defines functions that may be used to construct a pen array from a given function and palette:

\begin{verbatim}
pen[] palette(real[] f, pen[] palette);
pen[][] palette(real[][] f, pen[] palette);
\end{verbatim}

\section*{8.29 three}

This module fully extends the notion of guides and paths in \texttt{Asymptote} to three dimensions. It introduces the new types \texttt{guide3}, \texttt{path3}, and \texttt{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 \texttt{Asymptote}, \texttt{MetaPost}, and \texttt{MetaFont} [cf. J. C. Bowman, Proceedings in Applied Mathematics and Mechanics, 7:1, 2010021-2010022 (2007)].

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

\begin{verbatim}
import three;

size(100);

path3 g=(1,0,0)..(0,1,0)..(-1,0,0)..(0,-1,0)..cycle;
draw(g);
draw(0--Z,red+dashed,Arrow3);
\end{verbatim}
and then distorted into a saddle:

```asy```
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:

```asy```
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 be drawn with one of the routines:

```asy```
void draw(picture pic=currentpicture, surface s, int nu=1, int nv=1,
          material[] surfacenpen=currentpen, pen meshpen=nullpen,
          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[] surfacenpen, 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[] surfacenpen, pen meshpen=linewidth(0),
          light light=currentlight, light meshlight=nolight, string name="",
          render render=defaultrender);
```

```asy```
```
The parameters \texttt{nu} and \texttt{nv} specify the number of subdivisions for drawing optional mesh lines for each Bezier patch. The optional \texttt{name} parameter is used as a prefix for naming the surface patches in the PRC model tree. Here material is a structure defined in \texttt{three\_light.asy}:

```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 \texttt{plain\_prethree.asy} and \texttt{three\_light.asy}:

```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 \texttt{currentlight.background} (or \texttt{background} member of the specified \texttt{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 \texttt{currentlight.background=black+opacity(0.0)};

Asymptote also supports image-based lighting with the setting \texttt{settings.ibl=true}. This uses pre-rendered EXR images from the directory specified by \texttt{-imageDir} (which defaults to ibl) or, for WebGL rendering, the URL specified by \texttt{-imageURL} (which defaults to \url{https://vectorgraphics.gitlab.io/asymptote/ibl}). Additional rendered images can be generated on an NVIDIA GPU using the \texttt{reflect} program in the \texttt{cudareflect} subdirectory of the \texttt{Asymptote} source directory.

Sample Bezier surfaces are contained in the example files \texttt{BezierSurface.asy}, \texttt{teapot.asy}, \texttt{teapotIBL.asy}, and \texttt{parametricsurface.asy}. 
The structure `render` contains specialized rendering options documented at the beginning of module `three`.

The examples `elevation.asy` and `sphericalharmonic.asy` illustrate how to draw a surface with patch-dependent colors. The examples `vertexshading.asy` and `smoothelevation.asy` illustrate vertex-dependent colors, which are supported by Asymptote’s native OpenGL/WebGL renderers and the two-dimensional vector output format (settings.render=0). Since the PRC output format does not currently support vertex shading of Bezier surfaces, PRC patches are shaded with the mean of the four vertex colors.

A surface can be constructed from a cyclic `path3` with the constructor

```asy
surface surface(path3 external, triple[] internal=new triple[],
    pen[] colors=new pen[], bool3 planar=default);
```

and then filled:

```asy
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

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

and then filled:

```asy
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:

```asy
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

```asy
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`. The ratio of physical to logical screen pixels can be specified with the setting `devicepixelratio`. 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

   ```asy```
   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:

```html
<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).

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. A negative value of \( n \) is interpreted as \( |2n| \) for EPS and PDF formats and \( |n| \) for other formats. The default value of render is -1. 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 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 174), or specified in the script before module three (or graph3) is imported. The media9 LaTeX package is also required (see Section 8.19 [embed], page 101). 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 \texttt{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 90, 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:
\begin{verbatim}
\include{file.v3d}
\end{verbatim}
An online Javascript-based V3D-aware PDF viewer is available at https://github.com/vectorgraphics/pdfv3dReader.

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
\begin{verbatim}
asy -V file.v3d
\end{verbatim}
or using the v3d module and importv3d function in interactive mode (or within an Asymptote file):
\begin{verbatim}
import v3d;
importv3d("file.v3d");
\end{verbatim}

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 151).

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
\begin{verbatim}
void size3(picture pic=currentpicture, real x, real y=x, real z=y,
        bool keepAspect=pic.keepAspect);
\end{verbatim}
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 155, 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 \texttt{obliqueZ} is a synonym for \texttt{oblique}.

obliqueX
  \texttt{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
  \texttt{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=O, real zoom=1, pair viewportshift=0, bool showtarget=true, bool center=false)
  This projects from three to two dimensions using the view as seen at a point infinitely far away in the direction \texttt{unit(camera)}, orienting the camera so that, if possible, the vector \texttt{up} points upwards. Parallel lines are projected to parallel lines. The bounding volume is expanded to include \texttt{target} if \texttt{showtarget=true}. If \texttt{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=O, real zoom=1, pair viewportshift=0, bool showtarget=true, bool center=false)
  This is equivalent to
  \texttt{orthographic}((x,y,z),up,target,zoom,viewportshift,showtarget,center)
  The routine
  \texttt{triple camera(real alpha, real beta)};
  can be used to compute the camera position with the \(x\) axis below the horizontal at angle \texttt{alpha}, the \(y\) axis below the horizontal at angle \texttt{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=false)
  This projects from three to two dimensions, taking account of perspective, as seen from the location \texttt{camera} looking at \texttt{target}, orienting the camera so that, if possible, the vector \texttt{up} points upwards. If \texttt{autoadjust=true}, the camera will automatically be adjusted to lie outside the bounding volume for all possible interactive rotations about \texttt{target}. If \texttt{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 autoadjust=false, bool center=false)
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

\[
\text{triple invert(pair } z, \text{ projection } P=\text{currentprojection});
\]

A pair direction \( \text{dir} \) on the projection plane can be inverted to a triple direction relative to a point \( v \) with the routine

\[
\text{triple invert(pair } \text{dir}, \text{ triple } v, \text{ projection } P=\text{currentprojection}).
\]

Three-dimensional objects may be transformed with one of the following built-in transform3 types (the identity transformation is \text{identity4}):

\[
\text{shift(triple } v) \\
\text{translates by the triple } v;
\]

\[
\text{xscale3(real } x) \\
scales by } x \text{ in the } x \text{ direction;}
\]

\[
\text{yscale3(real } y) \\
scales by } y \text{ in the } y \text{ direction;}
\]

\[
\text{zscale3(real } z) \\
scales by } z \text{ in the } z \text{ direction;}
\]

\[
\text{scale3(real } s) \\
scales by } s \text{ in the } x, y, \text{ and } z \text{ directions;}
\]

\[
\text{scale(real } x, \text{ real } y, \text{ real } z) \\
scales by } x \text{ in the } x \text{ direction, by } y \text{ in the } y \text{ direction, and by } z \text{ in the } z \text{ direction;}
\]

\[
\text{rotate(real } \text{angle}, \text{ triple } v) \\
\text{rotates by } \text{angle} \text{ in degrees about the axis } 0--v;
\]

\[
\text{rotate(real } \text{angle}, \text{ triple } u, \text{ triple } v) \\
\text{rotates by } \text{angle} \text{ in degrees about the axis } u--v;
\]

\[
\text{reflect(triple } u, \text{ triple } v, \text{ triple } w) \\
\text{reflects about the plane through } u, v, \text{ and } w.
\]

When not multiplied on the left by a \text{transform3}, three-dimensional \LaTeX{} Labels are drawn as Bezier surfaces directly on the projection plane:

\[
\text{void label(picture pic=\text{currentpicture}, Label } L, \text{ triple position,} \\
\text{align align=\text{NoAlign}, pen } p=\text{\text{currentpen}}, \\
\text{light light=\text{nolight}, string name=""}, \\
\text{render render=\text{\text{defaultrender}}, interaction } interaction= \\
\text{settings.autobillboard ? \text{Billboard} : \text{Embedded})}
\]

The optional \text{name} parameter is used as a prefix for naming the label patches in the \text{PRC} model tree. The default interaction is \text{Billboard}, which means that labels are rotated interactively so that they always face the camera. The interaction \text{Embedded} means that the label interacts as a normal 3D surface, as illustrated in the example \text{billboard.asy}. Alternatively, a label can be transformed from the \text{XY} plane by an explicit \text{transform3} or mapped to a specified two-dimensional plane with the predefined \text{transform3} types \text{XY}, \text{YZ}, \text{ZX}, \text{YX}, \text{ZY}, \text{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

```cpp
transform3 planeproject(triple n, triple O=O, triple dir=n);
```

One can use

```cpp
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

```cpp
transform3 planeproject(path3 p, triple dir=normal(p));
```

The functions

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

\[
\begin{vmatrix}
a.x & a.y & a.z & 1 \\
b.x & b.y & b.z & 1 \\
c.x & c.y & c.z & 1 \\
d.x & d.y & d.z & 1 \\
\end{vmatrix}
\]

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

```cpp
real insphere(triple a, triple b, triple c, triple d, triple e);
```
returns a positive (negative) value if \( \mathbf{e} \) lies inside (outside) the sphere passing through points \( \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \) oriented so that \( \text{dot}(\text{cross}(\mathbf{a}-\mathbf{d}, \mathbf{b}-\mathbf{d}), \mathbf{c}-\mathbf{d}) \) is positive, or zero if all five points are cospherical. The value returned is the determinant

\[
\begin{vmatrix}
    a.x & a.y & a.z & a.x^2+a.y^2+a.z^2 & 1 \\
    b.x & b.y & b.z & b.x^2+b.y^2+b.z^2 & 1 \\
    c.x & c.y & c.z & c.x^2+c.y^2+c.z^2 & 1 \\
    d.x & d.y & d.z & d.x^2+d.y^2+d.z^2 & 1 \\
    e.x & e.y & e.z & e.x^2+e.y^2+e.z^2 & 1
\end{vmatrix}
\]

Here is an example showing all five guide3 connectors:

```python
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 0=0, 
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;
8.30 obj
This module allows one to construct surfaces from simple obj files, as illustrated in the example files galleon.asy and triceratops.asy.

8.31 graph3
This module implements three-dimensional versions of the functions in graph.asy. To draw an x axis in three dimensions, use the routine

```cpp
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, bool above=false);
```

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:

```cpp
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, arrowbar3 arrow=none);
```

The predefined three-dimensional axis types are

- `axis YZEquals(real y, real z, triple align=O, bool extend=false);`
- `axis XZEquals(real x, real z, triple align=O, bool extend=false);`
- `axis XYEquals(real x, real y, triple align=O, bool extend=false);`
- `axis YZZero(triple align=O, bool extend=false);`
- `axis XZZero(triple align=O, bool extend=false);`
- `axis XYZero(triple align=O, bool extend=false);`
- `axis Bounds(int type=Both, int type2=Both, triple align=O, 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(5,2,2);

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=0--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:

\[
\text{surface surface}(\text{real[][] } f, \text{pair a, pair b, bool[][] cond={}});
\]
\[
\text{surface surface}(\text{real[][] } f, \text{pair a, pair b, splinetype xsplinetype},
\text{splinetype ysplinetype=xsplinetype, bool[][] cond={}});
\]
\[
\text{surface surface}(\text{real[][] } f, \text{real[] x, real[] y},
\text{splinetype xsplinetype=null, splinetype ysplinetype=xsplinetype,}
\text{bool[][] cond={}})
\]
\[
\text{surface surface}(\text{triple[][] } f, \text{bool[][] cond={}});
\]
\[
\text{surface surface}(\text{real } f(\text{pair z}), \text{pair a, pair b, int nx=nmesh, int ny=nx,}
\text{bool cond(pair z)=null});
\]
\[
\text{surface surface}(\text{real } f(\text{pair z}), \text{pair a, pair b, int nx=nmesh, int ny=nx,}
\text{splinetype xsplinetype, splinetype ysplinetype=xsplinetype,}
\text{bool cond(pair z)=null});
\]
\[
\text{surface surface}(\text{triple } f(\text{pair z}), \text{real[] u, real[] v},
\text{splinetype[] usplinetype, splinetype[] vsplinetype=Spline,}
\text{bool cond(pair z)=null});
\]
\[
\text{surface surface}(\text{triple } f(\text{pair z}), \text{pair a, pair b, int nu=nmesh, int nv=nu,}
\text{bool cond(pair z)=null});
\]
\[
\text{surface surface}(\text{triple } f(\text{pair z}), \text{pair a, pair b, int nu=nmesh, int nv=nu,}
\text{splinetype[] usplinetype, splinetype[] vsplinetype=Spline,}
\text{bool cond(pair z)=null});
\]

The final two versions draw parametric surfaces for a function \( f(u,v) \) over the parameter space box(a,b), as illustrated in the example \texttt{parametricsurface.asy}. An optional splinetype \texttt{Spline} may be specified. The boolean array or function \texttt{cond} can be used to control which surface mesh cells are actually drawn (by default all mesh cells over box(a,b) are drawn).

One can also construct the surface generated by rotating a path \( g \) between \texttt{angle1} to \texttt{angle2} (in degrees) sampled \( n \) times about the line \( c--c+axis \):

\[
\text{surface surface}(\text{triple c, path3 g, triple axis, int n=nslice,}
\text{real angle1=0, real angle2=360, pen color(int i, real j)=null});
\]

The optional argument \texttt{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).

Surface lighting is illustrated in the example files \texttt{parametricsurface.asy} and \texttt{sinc.asy}. Lighting can be disabled by setting \texttt{light=nolight}, as in this example of a Gaussian surface:

\[
\text{import graph3;}
\]
A mesh can be drawn without surface filling by specifying `nullpen` for the surfacepen.

A vector field of $\text{nu} \times \text{nv}$ arrows on a parametric surface $f$ over box($a$, $b$) can be drawn with the routine

\[\text{picture vectorfield(path3 vector(pair v), triple f(pair z), pair a, pair b, int nu=nmesh, int nv=nu, bool truesize=false, real maxlength=truesize ? 0 : \text{maxlength}(f,a,b,nu,nv), bool cond(pair z)=null, pen p=currentpen, arrowbar3 arrow=Arrow3, margin3 margin=PenMargin3)}\]

as illustrated in the examples `vectorfield3.asy` and `vectorfieldsphere.asy`. 
8.32 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/):

```asy
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));
```

![3D Grid Example](image-url)
8.33 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:

```plaintext
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:

```plaintext
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;
}
```

8.34 tube

This module extends the `tube` surfaces constructed in `three_arrows.asy` to arbitrary cross sections, colors, and spine transformations. The routine

```plaintext
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 $\text{relpoint}(g,t)$ to use in constructing elementary tubes along $g$. The type \textit{coloredpath} is a generalization of \textit{path} to which a \textit{path} can be cast:

```plaintext
struct coloredpath
{
    path p;
    pen[] pens(real);
    int colortype=coloredSegments;
}
```

Here $p$ defines the cross section and the method \textit{pens(real t)} returns an array of pens (interpreted as a cyclic array) used for shading the tube patches at $\text{relpoint}(g,t)$. If \textit{colortype=coloredSegments}, the tube patches are filled as if each segment of the section was colored with the pen returned by \textit{pens(t)}, whereas if \textit{colortype=coloredNodes}, the tube components are vertex shaded as if the nodes of the section were colored.

A \textit{coloredpath} can be constructed with one of the routines:

```plaintext
coloredpath coloredpath(path p, pen[] pens(real),
    int colortype=coloredSegments);
```

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 on the relative time.

The casting of \textit{path} to \textit{coloredpath} allows the use of a \textit{path} instead of a \textit{coloredpath}; in this case the shading behavior is the default shading behavior for a surface.


### 8.35 flowchart

This module provides routines for drawing flowcharts. The primary structure is a \textit{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 $t$:

```plaintext
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:

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

The center of the block in user coordinates is stored in \textit{block.center} and the block size in PostScript coordinates is given by \textit{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):**

```plaintext
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);
```

```plaintext
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:**

```plaintext
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:**

```plaintext
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:**

```plaintext
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:**

```plaintext
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:**

```plaintext
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:

```plaintext
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`):

```plaintext
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;
});
```
8.36 contour

This module draws contour lines. To construct contours corresponding to the values in a real array \( c \) for a function \( f \) on \( \text{box}(a,b) \), use the routine

\[
guide[] \text{ contour}(\text{real } f(\text{real, real}), \text{pair } a, \text{pair } b, \\
\quad \text{real[] } c, \text{ int } nx=\text{ngraph}, \text{ int } ny=nx, \\
\quad \text{interpolate join}=\text{operator } --, \text{ int } subsample=1); 
\]

The integers \( nx \) and \( ny \) define the resolution. The default resolution, \( \text{ngraph} \times \text{ngraph} \) (here \( \text{ngraph} \) defaults to 100) can be increased for greater accuracy. The default interpolation operator is \( \text{operator } -- \) (linear). Spline interpolation (\( \text{operator } .. \)) may produce smoother contours but it can also lead to overshooting. The \( \text{subsample} \) parameter indicates the number of interior points that should be used to sample contours within each \( 1 \times 1 \) 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 \( \text{box}(a,b) \), use

\[
guide[] \text{ contour}(\text{real[]}[] f, \text{pair } a, \text{pair } b, \text{real[]}[] c, \\
\quad \text{interpolate join}=\text{operator } --, \text{ int } subsample=1); 
\]

To construct contours for an array of data values on a nonoverlapping regular mesh specified by the two-dimensional array \( z \),

\[
guide[] \text{ contour}(\text{pair[]}[] z, \text{real[]}[] f, \text{real[]}[] c, \\
\quad \text{interpolate join}=\text{operator } --, \text{ int } subsample=1); 
\]
To construct contours for an array of values $f$ specified at irregularly positioned points $z$, use the routine

$$\text{guide}[] \text{ contour(pair[]} z, \text{ real}[] f, \text{ real}[] c, \text{ interpolate join=operator --});$$

The contours themselves can be drawn with one of the routines

$$\text{void draw(picture pic=currentpicture, Label[]} L=\text{new Label}[],$$
$$\text{ guide[]} g, \text{ pen p=currentpen});$$

$$\text{void draw(picture pic=currentpicture, Label[]} L=\text{new Label}[],$$
$$\text{ guide[]} g, \text{ pen[]} p);$$

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

```plaintext
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 $100 \times 100$, using a dashed pen for negative contours and a solid pen for positive (and zero) contours:

```plaintext
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);
```
The next examples illustrates how contour lines can be drawn on color density images, with and without palette quantization:

```plaintext
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);

real[] Cvals=uniform(range.min,range.max,Divs);
draw(contour(f,a,b,Cvals,N,operator --),Tickpen+squarecap+beveljoin);
```
// 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:

```plaintext
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:

```plaintext
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.

8.37 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.

8.38 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 \( f(\text{triple}) \) or \( ff(\text{real,real,real}) \) over box(a,b), use the routine

```latex
surface implicitsurface(real f(\text{triple})=null, real ff(\text{real,real,real})=null, \text{triple} a, \text{triple} b, int n=nmesh, bool \text{keyword} overlapedges=false, int \text{keyword} nx=n, int \text{keyword} ny=n, int \text{keyword} nz=n, int \text{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.

### 8.39 slopefield

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

```latex
picture slopefield(real f(\text{real,real}), \text{pair} a, \text{pair} b, int nx=nmesh, int ny=nx, real tickfactor=0.5, \text{pen} p=\text{currentpen}, \text{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, \( \text{tickfactor} \) 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 \text{currentpicture} via the \text{add(picture)} command.

The function

```latex
path curve(\text{pair} c, real f(\text{real,real}), \text{pair} a, \text{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 \( \text{real} f(\text{real}) \) that depends on \( x \) only, as seen in this example:

```latex
import slopefield;
```
The \texttt{ode} module, illustrated in the example \texttt{odetest.asy}, implements a number of explicit numerical integration schemes for ordinary differential equations.
9 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)]
- `-animating` [false]
- `-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 animations [false]
- `-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` []
- `-d,-debug` Enable debugging messages [false]
- `-devicepixelratio n` Ratio of physical to logical pixels [1]
- `-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]
- `-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]
-htlmviewerOptions str []
-hyperrefOptions str [setpagesize=false,unicode,pdfborder=0 0 0]
-ibl Enable environment map image-based lighting [false]
-iconify Iconify rendering window [false]
-image str Environment image name [snowyField]
-imageDir str Environment image library directory [ibl]
-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]
-interrupt [false]
-k,-keep Keep intermediate files [false]
-keepaux Keep intermediate LaTeX .aux files [false]
-level n Postscript level [3]
-1,-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
-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]
-0,-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)]
Chapter 9: Command-line options

- **-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 (-1=auto) [-1]
- **-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]
- **-svgemulation** Emulate unimplemented SVG shading [true]
- **-tabcompletion** Interactive prompt auto-completion [true]
- **-tex engine** latex|pdflatex|xelatex|lualatex|tex|pdftex|luatex|context|none [latex]
- **-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]]
- **-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]
- **-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 10 [Interactive mode], page 176). In this case, the default output file is **out.eps**.

If - is given as the file argument, **Asymptote** reads from standard input.
If multiple files are specified, they are treated as separate `Asymptote` runs.

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

Default option values may be entered as `Asymptote` code in a configuration file named `config.asy` (or the file specified by the environment variable `ASYMPTOTE_CONFIG` or `-config` option). `Asymptote` will look for this file in its usual search path (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:

```asymptote
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`, `pdfreloaddOptions`, `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

```asymptote
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 anti-aliased `tiff` output at one pixel per `bp`:

```bash
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 `-0` option:

```bash
asy -0 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 \([-1,1] \times [-1,1]\)) 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

```bash
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:

```asymptote
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.
10 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:

```plaintext
> 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 174), the current directory (see [cd], page 58), and breakpoints (see Chapter 16 [Debugger], page 183);

- **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 \texttt{-localhistory} was specified, in which case the history will be stored in the file \texttt{.asy\_history} in the current directory).

Typing \texttt{ctrl-C} interrupts the execution of \texttt{Asymptote} code and returns control to the interactive prompt.

Interactive mode is implemented with the GNU \texttt{readline} library, with command history and auto-completion. To customize the key bindings, see: \url{https://tiswww.case.edu/php/chet/readline/readline.html}

The file \texttt{asymptote.py} in the \texttt{Asymptote} system directory provides an alternative way of entering \texttt{Asymptote} commands interactively, coupled with the full power of \texttt{Python}. Copy this file to your \texttt{Python path} and then execute from within \texttt{Python 3} the commands

\begin{verbatim}
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")
\end{verbatim}
11 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.

11.1 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 pyqt5, cson, and numpy:

```
pip3 install cson numpy pyqt5 PyQt5.sip
```

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
```

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.

11.2 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).
12 Command-Line Interface

Asymptote code may be sent to the http://asymptote.ualberta.ca server directly from the command line, specifying any options directly in the URL:

- SVG output:
  ```
curl --data-binary 'import venn;' 'asymptote.ualberta.ca:10007?f=svg' | display -
  ```

- HTML output:
  ```
  ```

- V3D output:
  ```
curl --data-binary 'import teapot;' 'asymptote.ualberta.ca:10007?f=v3d' -o teapot.v3d
  ```

- PDF output with rendered bitmap at 2 pixels per bp:
  ```
  ```

- PDF output with rendered bitmap at 4 pixels per bp:
  ```
  ```

- PRC output:
  ```
  ```

- PRC output with rendered preview bitmap at 4 pixels per bp:
  ```
  ```

The source code for the command-line interface is available at https://github.com/vectorgraphics/asymptote-http-server.
13 Language server protocol


Emacs users can enable the Asymptote language server protocol by installing lsp-mode using the following procedure:

- Add to the .emacs initialization file:
  
  ```lisp
  (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:
  
  ```lisp
  (require 'lsp-mode)
  (add-to-list 'lsp-language-id-configuration '(asy-mode . "asymptote"))
  ```

  ```lisp
  (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
14 PostScript to Asymptote

The excellent PostScript editor pstoedit (version 3.50 or later; available from https://sourceforge.net/projects/pstoedit/) includes an Asymptote backend. Unlike virtually all other pstoedit backends, this driver includes native clipping, even-odd fill rule, PostScript subpath, and full image support. Here is an example:

asy -V /usr/local/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.
15 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 8) 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 69) 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).
16 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:

```cpp
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`):

```asymptote
stop("test",2,quote{ignore=(++count <= 10);});
```

To set a break point in file `file` at the first line containing the string `text`, use:

```cpp
void stop(string file, string text, code s=quote{});
```

To list all breakpoints, use:

```cpp
void breakpoints();
```

To clear a breakpoint, use:

```cpp
void clear(string file, int line);
```

To clear all breakpoints, use:

```cpp
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:

```cpp
void breakpoint(code s=quote{});
```
17 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 http://asymptote.ualberta.ca:

https://github.com/vectorgraphics/asymptoteWebApplication
## Index

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>66, 63, 65</td>
</tr>
<tr>
<td>!=</td>
<td>63, 65</td>
</tr>
<tr>
<td>#</td>
<td>65</td>
</tr>
<tr>
<td>#=</td>
<td>65, 176, 66</td>
</tr>
<tr>
<td>%</td>
<td>65</td>
</tr>
<tr>
<td>%=</td>
<td>66</td>
</tr>
<tr>
<td>&amp;</td>
<td>25, 65, 65</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>65</td>
</tr>
<tr>
<td>*</td>
<td>41, 65, 65, 66</td>
</tr>
<tr>
<td>+</td>
<td>41, 65, 66</td>
</tr>
<tr>
<td>+=</td>
<td>66</td>
</tr>
<tr>
<td>-</td>
<td>65, 13, 66, 25, 66</td>
</tr>
<tr>
<td>-=</td>
<td>66</td>
</tr>
<tr>
<td>--</td>
<td>175</td>
</tr>
<tr>
<td>-=c</td>
<td>175</td>
</tr>
<tr>
<td>-=u</td>
<td>175</td>
</tr>
<tr>
<td>-v</td>
<td>4, 10</td>
</tr>
<tr>
<td>.</td>
<td>13, 6</td>
</tr>
<tr>
<td>.asy</td>
<td>6</td>
</tr>
<tr>
<td>/</td>
<td>65, 66</td>
</tr>
<tr>
<td>/=</td>
<td>66</td>
</tr>
<tr>
<td>:</td>
<td>66</td>
</tr>
<tr>
<td>::</td>
<td>25</td>
</tr>
<tr>
<td>&lt;</td>
<td>65</td>
</tr>
<tr>
<td>&lt;=</td>
<td>65</td>
</tr>
<tr>
<td>=</td>
<td>63, 65</td>
</tr>
<tr>
<td>==</td>
<td>63, 65</td>
</tr>
<tr>
<td>&gt;</td>
<td>65</td>
</tr>
<tr>
<td>&gt;=</td>
<td>65</td>
</tr>
<tr>
<td>?</td>
<td>66</td>
</tr>
<tr>
<td>^</td>
<td>65, 66, 13</td>
</tr>
<tr>
<td>^=</td>
<td>66</td>
</tr>
<tr>
<td>^^</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>103</td>
</tr>
<tr>
<td>3</td>
<td>152</td>
</tr>
<tr>
<td>3D graphs</td>
<td>152</td>
</tr>
<tr>
<td>3D grids</td>
<td>157</td>
</tr>
<tr>
<td>3D PostScript</td>
<td>151</td>
</tr>
</tbody>
</table>
Index

A

a4 ............................................. 5
abort ....................................... 32
abs ....................................... 27, 28, 73
abs2 ....................................... 27, 28
absolute .................................. 4, 143
accel .................................... 35, 149
access .................................. 84
acknowledgments ....................... 184
acCos ..................................... 72
acos ....................................... 72
acosh ..................................... 72
add ........................................ 54, 145
addViews ................................ 147
adjust .................................... 43
A ........................................... 73
A1 ........................................ 73
Airy ....................................... 73
alias .................................... 63, 77
Align ....................................... 20
aligndir ................................ 174
all .......................................... 80
Allow ...................................... 49
and ......................................... 25
AND ........................................ 66
angle ....................................... 27
animate .................................. 4, 60, 101
animation .................................. 101
annotate .................................. 102
antialias .................................. 143, 174
append .................................... 58, 75
Arc ......................................... 33
arc ......................................... 33, 145
ArcArrow .................................. 15
ArcArrow3 ................................ 150
ArcArrows ................................ 15
ArcArrows3 ................................ 150
arc length ................................ 35, 149
arc point ................................ 36
arctime .................................. 36, 149
arguments .................................. 69
arithmetic operators ..................... 65
array ........................................ 31, 76
array iteration ........................... 26
arrays ...................................... 74
Arrow ....................................... 15
arrow ....................................... 15, 21
arrow keys .................................. 10, 178
Arrow3 ..................................... 150
arrowbar .................................. 15
arrowhead ................................ 16
Arrows ..................................... 15
arrows .................................... 15
Arrows3 .................................. 150
as ........................................... 85
ascii ....................................... 31
aSin ....................................... 72
asin ........................................ 72

B

babel ....................................... 102
background ................................ 140
background color ........................... 53
BackView .................................. 147
Bar .......................................... 15
Bar3 ........................................ 150
Bars ......................................... 15
Bars3 ....................................... 150
barsize ...................................... 15
base modules ................................ 95
basealign .................................. 44
baseline .................................... 22
batch mode .................................. 10
beep ......................................... 33
BeginArcArrow ............................. 15
BeginArcArrow3 .......................... 150
BeginArrow ................................ 15
BeginArrow3 ................................ 150
BeginBar .................................... 15
BeginBar3 .................................. 150
BeginDotMargin ........................... 16
BeginDotMargin3 ......................... 151
BeginMargin ................................ 16
BeginMargin3 ................................ 151
BeginPenMargin ........................... 16
BeginPenMargin2 ......................... 151
BeginPenMargin3 .......................... 151
BeginPoint ................................ 21
Bessel ...................................... 73
bevel ....................................... 160
| beveljoin                      | 43  |
| Bezier curves                 | 24  |
| Bezier patch                  | 141 |
| Bezier triangle               | 141 |
| bezulate                      | 141 |
| Bi                            | 73  |
| Bi_deriv                      | 73  |
| Billboard                     | 148 |
| binary                        | 59  |
| binary format                 | 59  |
| binary operators              | 65  |
| binarytree                    | 99  |
| black stripes                 | 143 |
| Blank                         | 15  |
| block.bottom                  | 159 |
| block.bottomleft              | 159 |
| block.bottomright             | 159 |
| block.center                  | 159 |
| block.draw                    | 160 |
| block.left                    | 159 |
| block.position                | 159 |
| block.right                   | 159 |
| block.top                     | 159 |
| block.topleft                 | 159 |
| block.topright                | 159 |
| bool                          | 26  |
| bool3                         | 27  |
| boolean operators             | 65  |
| Bottom                        | 105 |
| BottomTop                     | 105 |
| BottomView                    | 147 |
| bounding box                 | 53  |
| Bounds                        | 152 |
| bounds                        | 133 |
| box                           | 134, 145 |
| bp                            | 10  |
| brace                         | 34  |
| break                         | 26  |
| breakpoints                   | 183 |
| brick                         | 47  |
| broken axis                   | 123 |
| bug reports                   | 182 |
| buildcycle                    | 37  |
| Button-1                      | 178 |
| Button-2                      | 178 |
| BWRainbow                     | 133 |
| BWRainbow2                    | 133 |
| Center                        | 21  |
| checker                       | 47  |
| Chinese                       | 46  |
| choose                        | 73  |
| Ci                            | 13  |
| Circle                        | 33  |
| circle                        | 33, 145, 160 |
| circleframe                   | 97  |
| CJK                           | 46  |
| clamped                       | 103 |
| clang                         | 7   |
| clear                         | 183 |
| clip                          | 20  |
| CLZ                           | 66  |
| cm                            | 11  |
| cmd                           | 4   |
| cmyk                          | 41  |
| colatitude                    | 29  |
| color                         | 41, 155 |
| coloredNodes                  | 159 |
| coloredpath                   | 159 |
| coloredSegments               | 159 |
| colorless                     | 41  |
| colors                        | 41  |
| comma                         | 59  |
| comma-separated-value mode     | 81  |
| command-line interface        | 179 |
| command-line options          | 5, 171 |
| comment character             | 58  |
| compass directions            | 12  |
| Compiling from UNIX source    | 6   |
| complement                    | 77  |
| concat                        | 77  |
| conditional                   | 26, 66 |
| config                        | 5, 174 |
| configuration file            | 4, 174 |
| configuring                   | 4   |
| conj                          | 27  |
| constructors                  | 63  |
| context                       | 174 |
| continue                      | 26, 183 |
| contour                       | 162 |
| contour3                      | 168 |
| controls                      | 24, 138 |
| controlSpecifier              | 40  |
| convert                       | 60  |
| convertOptions                | 174 |
| Coons shading                 | 19  |
| copy                          | 77  |
| cos                           | 72  |
| Cos                           | 72  |
| cosh                          | 72  |
| cputime                       | 74  |
| crop                          | 117 |
| cropping graphs               | 117 |
| cross                         | 28, 29, 113 |
| crossframe                    | 97  |
| crosshatch                    | 48  |

C

C string                                    | 30  |

CAD                                      | 103 |
camera                                    | 146 |
casts                                      | 83  |
cbrt                                       | 72  |
cd                                         | 58  |
ceil                                        | 72  |
center                                     | 146 |
<table>
<thead>
<tr>
<th>Index</th>
<th>188</th>
</tr>
</thead>
<tbody>
<tr>
<td>csv</td>
<td>81</td>
</tr>
<tr>
<td>CTZ</td>
<td>66</td>
</tr>
<tr>
<td>cubicroots</td>
<td>80</td>
</tr>
<tr>
<td>curl</td>
<td>25, 138</td>
</tr>
<tr>
<td>curlSpecifier</td>
<td>40</td>
</tr>
<tr>
<td>currentlight</td>
<td>140</td>
</tr>
<tr>
<td>currentpen</td>
<td>40</td>
</tr>
<tr>
<td>currentprojection</td>
<td>147</td>
</tr>
<tr>
<td>curve</td>
<td>169</td>
</tr>
<tr>
<td>custom axis types</td>
<td>105</td>
</tr>
<tr>
<td>custom mark routine</td>
<td>116</td>
</tr>
<tr>
<td>custom tick locations</td>
<td>107</td>
</tr>
<tr>
<td>cut</td>
<td>37</td>
</tr>
<tr>
<td>cycle</td>
<td>11, 13, 138</td>
</tr>
<tr>
<td>cyclic</td>
<td>34, 40, 75, 149</td>
</tr>
<tr>
<td>Cyrillic</td>
<td>46</td>
</tr>
<tr>
<td>csv</td>
<td>81</td>
</tr>
<tr>
<td>CTZ</td>
<td>66</td>
</tr>
<tr>
<td>cubicroots</td>
<td>80</td>
</tr>
<tr>
<td>curl</td>
<td>25, 138</td>
</tr>
<tr>
<td>curlSpecifier</td>
<td>40</td>
</tr>
<tr>
<td>currentlight</td>
<td>140</td>
</tr>
<tr>
<td>currentpen</td>
<td>40</td>
</tr>
<tr>
<td>currentprojection</td>
<td>147</td>
</tr>
<tr>
<td>curve</td>
<td>169</td>
</tr>
<tr>
<td>custom axis types</td>
<td>105</td>
</tr>
<tr>
<td>custom mark routine</td>
<td>116</td>
</tr>
<tr>
<td>custom tick locations</td>
<td>107</td>
</tr>
<tr>
<td>cut</td>
<td>37</td>
</tr>
<tr>
<td>cycle</td>
<td>11, 13, 138</td>
</tr>
<tr>
<td>cyclic</td>
<td>34, 40, 75, 149</td>
</tr>
<tr>
<td>Cyrillic</td>
<td>46</td>
</tr>
</tbody>
</table>

**D**

| Dasheddotted | 42 |
| Dashed       | 42 |
| Data types   | 26 |
| Date          | 31 |
| Debian        | 3 |
| Debugger      | 183 |
| Declaration   | 26 |
| Deconstruct   | 178 |
| Default arguments | 69 |
| Defaultformat | 106 |
| DefaultHead   | 16 |
| DefaultHead3  | 150 |
| DefaultPen    | 41, 43, 45, 49, 50 |
| Defaultrender | 139 |
| Deferreddrawing | 95 |
| Degrees       | 72 |
| Degrees       | 27, 72 |
| Delete        | 60, 75 |
| Description   | 1 |
| Devicepixelratio | 142 |
| Diagonal      | 80 |
| Diamond       | 160 |
| Diffuse       | 140 |
| Diffusepen    | 140 |
| Dimension     | 81 |
| Dir            | 6, 28, 29, 35, 149 |
| Direction specifier | 24 |
| Directory     | 58 |
| Dirspecifier  | 40 |
| Dirtime       | 36 |
| Display       | 4 |
| Do             | 26 |
| DDSendl       | 59 |
| DDSnswl       | 59 |
| Dot            | 17, 28, 29, 79 |
| DotMargin     | 16 |
| DotMargin3    | 151 |
| DotMargins    | 17 |
| DotMargins3   | 151 |
| Dotted        | 42 |
| Double deferred drawing | 144 |
| Double precision | 59 |
| Draw           | 53 |
| Draw           | 15, 18, 141 |
| Drawer         | 57 |
| Drawing commands | 15 |
| Drawline       | 95 |
| Drawtree       | 100 |
| Dvips          | 5 |
| DvipsOptions   | 174 |
| Dvisgm         | 5, 174 |
| DvisgmMultipleFiles | 178 |
| DvisgmOptions  | 174 |

**E**

| E                        | 12, 73 |
| Ee                        | 2 |
| Edit modes                | 7 |
| Ellipse                   | 33 |
| Elliptic functions        | 73 |
| Else                      | 26 |
| Emacs                     | 7 |
| Embed                      | 101 |
| Embedded                  | 148 |
| Emisivepen                | 140 |
| Empty                     | 51 |
| EndArrow                  | 15 |
| EndArrow3                 | 150 |
| EndArrow3                 | 15 |
| EndBar                    | 15 |
| Endbar                    | 150 |
| EndDotMargin              | 17 |
| EndDotMargin3             | 151 |
| End1                      | 59 |
| EndMargin                 | 16 |
| EndMargin3                | 151 |
| EndPenMargin              | 16 |
| EndPenMargin2             | 151 |
| EndPenMargin3             | 151 |
| EndPoint                  | 21 |
| Envelope                  | 22 |
| Environment variables     | 5 |
| Eof                       | 59, 80 |
| Eol                       | 59, 80 |
| Erp                       | 72 |
| Erfc                      | 72 |
| Error                     | 58, 59 |
| Error bars                | 114 |
| Errorbars                 | 113 |
| Eval                      | 85, 86 |
| Evenodd                   | 13, 44 |
| Exit                      | 32, 176, 183 |
| Exp                       | 72 |
Index

I
i_scaled ........................................... 73
ibl .................................................. 140
iconify ............................................. 143
identity ........................................... 50, 72, 80
identity4 ........................................... 148
if ....................................................... 26
IgnoreAspect ...................................... 51
image ................................................. 133, 134
image-based lighting .............................. 140
ImageMagick ....................................... 5, 101, 174
images ............................................... 133
implicit casts ........................................ 83
implicit linear solver ............................ 102
implicit scaling ..................................... 67
implicitSurface .................................... 169
import ............................................... 84
import3d ........................................... 144
inches ............................................... 11
incircle ............................................. 28
include ............................................... 86
including images ................................... 21
increasing .......................................... 96
inf ..................................................... 27
inheritance .......................................... 64
initialized .......................................... 75
initializers ......................................... 61
inline ............................................... 90
I .......................................................... 73
InOutTicks .......................................... 152
input .................................................. 58, 176
input encoding ..................................... 46
insert ............................................... 30, 75
inside ............................................... 38
insphere ............................................ 149
int ..................................................... 183
installation ......................................... 3
int ..................................................... 27
integer division ..................................... 65
integral .............................................. 73
integrate ........................................... 73
interactive mode ................................... 10, 176
interior ............................................. 38
interp ............................................... 66
interpolate ......................................... 96
intersect ........................................... 36, 95, 149
intersection ........................................ 37, 95, 149
intersectionpoint ................................... 37, 149
intersections ....................................... 36, 149
IntTicks ............................................. 152
intMax ............................................... 27
intMin ............................................... 27
inverse ............................................. 50, 80
invert ............................................... 147
invisible ............................................ 41

J
J .......................................................... 72, 73
Japanese ............................................ 46

K
k_scaled ............................................. 73
Kate ................................................... 8
KDE editor ........................................... 8
keepAspect .......................................... 51, 90
keyboard bindings .................................. 142
keys .................................................... 75
keyword .............................................. 70
keyword-only ........................................ 70
keywords ............................................. 70
K ......................................................... 73
Korean ................................................ 46

L
label .................................................. 12, 20, 148
Label ................................................ 17, 20, 108
labelmargin ........................................... 20
labelpath .......................................... 102
labelpath3 .......................................... 102
labelx ............................................... 108
labely ............................................... 108
Landscape .......................................... 52
language context .................................... 46
language server protocol .......................... 180
lastcut ............................................. 37
lazy-mode .......................................... 7
latex .................................................. 174
LaTeX NFSS fonts ................................... 46
LaTeX usage ......................................... 90
latexmk ............................................. 90
latitude ............................................. 29
latticeshade ........................................ 18
layer ................................................... 15
left ..................................................... 15
least squares ....................................... 97, 127
Left ................................................... 107
LeftRight ............................................ 107
LeftSide ............................................. 21
LeftTicks ........................................... 105, 107
LeftView ............................................ 147
legend ............................................... 15, 17, 111
Legendre ............................................ 73
length ................................................ 27, 28, 30, 34, 39, 75, 149
letter .................................................. 5
lexorder ............................................. 96
libcurl .............................................. 85
libm routines ....................................... 72
libsigsegv .......................................... 69, 182
light ............................................... 140
limits ............................................... 117
line .......................................................... 80
line mode ................................................. 80
Linear ....................................................... 120
linecap ....................................................... 103
linejoin ....................................................... 43
lineskip ....................................................... 45
linetype ....................................................... 43
linewidth ..................................................... 43
locale ......................................................... 31
Log .......................................................... 120
log ........................................................... 72
log-log graph ................................................. 120
log10 ........................................................ 72
log1p ........................................................ 72
log2 ........................................................... 122
logarithmic graph ........................................... 120
logical operators .......................................... 65
longdashed ................................................. 42
longdashed ............................................... 122
longitudinal ............................................... 29
loop .......................................................... 26
LISP .......................................................... 180
luatex ......................................................... 174
luatex ......................................................... 174
MetaPost pickup ............................................. 40
MetaPost whatever ......................................... 102
Microsoft Windows ......................................... 3
MidArrow ....................................................... 15
MidArrow3 ..................................................... 150
MidArrow ....................................................... 15
MidArrow3 ..................................................... 150
midpoint ....................................................... 36
min ........................................................... 38, 51, 78, 149
minbound ..................................................... 28, 29
minipage ....................................................... 22
mintimes ....................................................... 37
miter ........................................................ 43
miterlimit .................................................... 44
mktemp ......................................................... 59
mm ........................................................... 11
mobile browser ............................................. 143
mode .......................................................... 59
monotonic ..................................................... 103
mouse ......................................................... 178
mouse bindings .............................................. 142
mouse wheel .................................................. 178
Move .......................................................... 49
MoveQuiet ..................................................... 49
multisample .................................................. 142

M
MacOS X binary distributions 3
MacOS X configuration 7
magick 5, 60, 101, 174
makepen ...................................................... 49
map .......................................................... 76, 99
Margin ......................................................... 16
Margin3 ....................................................... 151
Margins ....................................................... 16
margins ......................................................... 144
Margins3 ....................................................... 151
mark .......................................................... 113
markangle ..................................................... 98
marker ......................................................... 113
markers ....................................................... 97
marknodes ..................................................... 113
markuniform .................................................. 113
mask .......................................................... 27
material ....................................................... 140
math .......................................................... 95
mathematical functions .................................... 72
max 38, 51, 78, 149
maxbound ..................................................... 28, 29
maxtile ......................................................... 143
maxtimes ....................................................... 37
maxviewport ................................................... 143
metallic ....................................................... 140
MetaPost ....................................................... 102
MetaPost ....................................................... 25
MetaPost cutafter .......................................... 37
MetaPost cutbefore ......................................... 37

N
name .......................................................... 59
named arguments ........................................... 70
nan ........................................................... 27
natural ......................................................... 103
clear .......................................................... 28
new .......................................................... 62, 76
newframe ...................................................... 51
newl ........................................................... 59
newpage ......................................................... 15
newton ......................................................... 73
next ........................................................... 183
noalign ......................................................... 44
node .......................................................... 12
NoFill ......................................................... 53
noglobalread ................................................. 58
noind ......................................................... 140
NoMargin ..................................................... 16
NoMargin3 .................................................... 151
None ........................................................ 15
none .......................................................... 59
normal ......................................................... 149
nosafe ......................................................... 174
notaknot ....................................................... 103
NUT ........................................................... 66
NoTics ......................................................... 105
NoTics3 ...................................................... 152
null ............................................................ 62
nullpen ....................................................... 20, 53
NURBS ......................................................... 146
<table>
<thead>
<tr>
<th>O</th>
<th>192</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>152</td>
</tr>
<tr>
<td>object</td>
<td>22</td>
</tr>
<tr>
<td>oblique</td>
<td>146</td>
</tr>
<tr>
<td>obliqueX</td>
<td>146</td>
</tr>
<tr>
<td>obliqueY</td>
<td>146</td>
</tr>
<tr>
<td>obliqueZ</td>
<td>146</td>
</tr>
<tr>
<td>ode</td>
<td>170</td>
</tr>
<tr>
<td>offset</td>
<td>43</td>
</tr>
<tr>
<td>OmitTick</td>
<td>107</td>
</tr>
<tr>
<td>OmitTickInterval</td>
<td>107</td>
</tr>
<tr>
<td>OmitTickIntervals</td>
<td>107</td>
</tr>
<tr>
<td>opacity</td>
<td>47</td>
</tr>
<tr>
<td>open</td>
<td>58</td>
</tr>
<tr>
<td>OpenGLE</td>
<td>142</td>
</tr>
<tr>
<td>operator</td>
<td>66</td>
</tr>
<tr>
<td>operator *</td>
<td>31</td>
</tr>
<tr>
<td>operator --</td>
<td>103</td>
</tr>
<tr>
<td>operator</td>
<td>103</td>
</tr>
<tr>
<td>operator answer</td>
<td>176</td>
</tr>
<tr>
<td>operator cast</td>
<td>83</td>
</tr>
<tr>
<td>operator ecast</td>
<td>84</td>
</tr>
<tr>
<td>operator init</td>
<td>61, 64</td>
</tr>
<tr>
<td>operators</td>
<td>65</td>
</tr>
<tr>
<td>options</td>
<td>171</td>
</tr>
<tr>
<td>orient</td>
<td>28</td>
</tr>
<tr>
<td>orientation</td>
<td>52</td>
</tr>
<tr>
<td>orthographic</td>
<td>146</td>
</tr>
<tr>
<td>0</td>
<td>145</td>
</tr>
<tr>
<td>OR</td>
<td>66</td>
</tr>
<tr>
<td>outformat</td>
<td>142</td>
</tr>
<tr>
<td>outprefix</td>
<td>52</td>
</tr>
<tr>
<td>output</td>
<td>58</td>
</tr>
<tr>
<td>OutTicks</td>
<td>152</td>
</tr>
<tr>
<td>overloading functions</td>
<td>68</td>
</tr>
<tr>
<td>overwrite</td>
<td>49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>192</th>
</tr>
</thead>
<tbody>
<tr>
<td>pack</td>
<td>22</td>
</tr>
<tr>
<td>packing</td>
<td>71</td>
</tr>
<tr>
<td>pad</td>
<td>53</td>
</tr>
<tr>
<td>pair</td>
<td>11</td>
</tr>
<tr>
<td>pairs</td>
<td>78</td>
</tr>
<tr>
<td>paperheight</td>
<td>5</td>
</tr>
<tr>
<td>papertype</td>
<td>5</td>
</tr>
<tr>
<td>paperwidth</td>
<td>5</td>
</tr>
<tr>
<td>parallelogram</td>
<td>160</td>
</tr>
<tr>
<td>parametric surface</td>
<td>155</td>
</tr>
<tr>
<td>parametrized curve</td>
<td>117</td>
</tr>
<tr>
<td>partialism</td>
<td>96</td>
</tr>
<tr>
<td>path-dependent colors</td>
<td>141</td>
</tr>
<tr>
<td>path</td>
<td>12</td>
</tr>
<tr>
<td>path markers</td>
<td>113</td>
</tr>
<tr>
<td>path[]</td>
<td>13</td>
</tr>
<tr>
<td>path3</td>
<td>138</td>
</tr>
<tr>
<td>patterns</td>
<td>47</td>
</tr>
<tr>
<td>PBR</td>
<td>140</td>
</tr>
<tr>
<td>pdflatex</td>
<td>174</td>
</tr>
<tr>
<td>pdfloadOptions</td>
<td>174</td>
</tr>
<tr>
<td>pdfviewer</td>
<td>4</td>
</tr>
<tr>
<td>pdfviewerOptions</td>
<td>174</td>
</tr>
<tr>
<td>PDF</td>
<td>174</td>
</tr>
<tr>
<td>pen</td>
<td>40</td>
</tr>
<tr>
<td>PenMargin</td>
<td>16</td>
</tr>
<tr>
<td>PenMargin2</td>
<td>151</td>
</tr>
<tr>
<td>PenMargin3</td>
<td>151</td>
</tr>
<tr>
<td>PenMargins</td>
<td>16</td>
</tr>
<tr>
<td>PenMargins2</td>
<td>151</td>
</tr>
<tr>
<td>PenMargins3</td>
<td>151</td>
</tr>
<tr>
<td>periodic</td>
<td>103</td>
</tr>
<tr>
<td>perl</td>
<td>90</td>
</tr>
<tr>
<td>perpendicular</td>
<td>96</td>
</tr>
<tr>
<td>perspective</td>
<td>146</td>
</tr>
<tr>
<td>physically based rendering</td>
<td>140</td>
</tr>
<tr>
<td>picture</td>
<td>51</td>
</tr>
<tr>
<td>picture alignment</td>
<td>54</td>
</tr>
<tr>
<td>picture.add</td>
<td>57</td>
</tr>
<tr>
<td>picture.addPoint</td>
<td>57</td>
</tr>
<tr>
<td>picture.calculateTransform</td>
<td>52</td>
</tr>
<tr>
<td>picture.fit</td>
<td>52</td>
</tr>
<tr>
<td>picture.scale</td>
<td>52</td>
</tr>
<tr>
<td>piecewisestraight</td>
<td>34</td>
</tr>
<tr>
<td>pixel</td>
<td>151</td>
</tr>
<tr>
<td>P1</td>
<td>73</td>
</tr>
<tr>
<td>plain</td>
<td>95</td>
</tr>
<tr>
<td>planar</td>
<td>141</td>
</tr>
<tr>
<td>plane</td>
<td>145</td>
</tr>
<tr>
<td>planeproject</td>
<td>149</td>
</tr>
<tr>
<td>point</td>
<td>34, 40, 149</td>
</tr>
<tr>
<td>polar</td>
<td>29</td>
</tr>
<tr>
<td>polargraph</td>
<td>104</td>
</tr>
<tr>
<td>polygon</td>
<td>113</td>
</tr>
<tr>
<td>pop</td>
<td>75</td>
</tr>
<tr>
<td>Portrait</td>
<td>52</td>
</tr>
<tr>
<td>position</td>
<td>140, 142</td>
</tr>
<tr>
<td>postcontrol</td>
<td>35, 149</td>
</tr>
<tr>
<td>postfix operators</td>
<td>66</td>
</tr>
<tr>
<td>postscript</td>
<td>56</td>
</tr>
<tr>
<td>PostScript fonts</td>
<td>46</td>
</tr>
<tr>
<td>PostScript subpath</td>
<td>13</td>
</tr>
<tr>
<td>pstoedit</td>
<td>181</td>
</tr>
<tr>
<td>pviewer</td>
<td>4</td>
</tr>
<tr>
<td>pviewerOptions</td>
<td>174</td>
</tr>
<tr>
<td>pt</td>
<td>11</td>
</tr>
<tr>
<td>public</td>
<td>62</td>
</tr>
<tr>
<td>push</td>
<td>75</td>
</tr>
<tr>
<td>P</td>
<td>73</td>
</tr>
<tr>
<td>Python usage</td>
<td>177</td>
</tr>
</tbody>
</table>
Q
quadraticroots ........................... 80
quarticroots ............................ 95
quick reference ......................... 2
quit ................................ 10, 176, 183
quote .................................. 86
quotient ................................ 65

R
radialshade .............................. 19
RadialShade ............................. 53
RadialShadeDraw ......................... 53
radians .................................. 72
radius ................................... 35, 149
Rainbow ................................ 133
rand ...................................... 73
randMax .................................. 73
read ...................................... 81
reading ................................... 58
reading string arrays .................... 80
readline ................................. 60
real ........................................ 27
realDigits ................................. 27
realEpsilon ............................... 27
realMax ................................... 27
realMin ................................... 27
realmult ................................... 28
realschur ................................. 79
rectangle ................................. 160
recursion ................................ 69
reference ................................ 2
reflect .................................... 50
Relative .................................. 21
relpoint ................................... 36
reltime ................................... 36
remainder ................................ 72
rename .................................... 60
render .................................... 139, 142, 174
replace ................................... 31
resetdefaultpen ........................... 50
rest arguments ............................ 71
restore .................................... 56
restricted ................................ 62
return ..................................... 183
reverse ................................. 30, 36, 40, 77, 149
rewind .................................... 59
rfind ...................................... 30
rgb ......................................... 41
Riemann zeta function .................. 73
Right ....................................... 107
RightSide ................................ 21
RightTicks ................................. 105, 107
RightView ................................ 147
Rotate .................................... 21
rotate ...................................... 148
Rotate(pair z) ............................. 21
round ...................................... 72
roundcap .................................. 43
roundedpath ............................... 101
roundjoin ................................ 43
roundrectangle ......................... 160
RPM ........................................ 3
runtime imports ......................... 85
Russian .................................... 46

S
safe ........................................ 174
save ........................................ 56
saveline ................................. 60
Scale ...................................... 21, 120
cscale ................................... 43, 50, 120, 148
cscale3 .................................... 148
scaled graph ............................... 119
schur ....................................... 79
scientific graph .......................... 110
scroll ...................................... 60
search .................................... 77
search paths .............................. 6
Seascape .................................. 52
secondary axis ......................... 124
secondaryX ............................... 124
secondaryY ............................... 124
seconds ................................... 32
seek ....................................... 59
seekeof .................................... 59
segment ................................... 96
segmentation fault ...................... 182
self operators ............................ 66
sequence .................................. 76
settings ................................... 4, 174
sgn .......................................... 72
shading .................................... 18
Shift ....................................... 21
shift ....................................... 50, 51, 148
shiftless .................................. 51
shininess .................................. 140
shipout .................................... 52
showtarget ............................... 146
Si .......................................... 73
signedint .................................. 59
SimpleHead ................................ 16
simplex .................................... 95
simplex2 .................................. 95
simpson .................................... 73
sin .......................................... 72
Sin .......................................... 72
single precision ......................... 59
gsingleint .................................. 59
gsinglereal ................................ 59
gsinh ....................................... 72
SixViews .................................. 147
SixViewsFR ................................ 147
SixViewsUS ................................ 147
size ...................................... 11, 34, 39, 51, 52, 149, 174
Z

zaxis3 ........................................ 152
zero_Ai ..................................... 73
zero_Ai_deriv ............................... 73
zero_Bi ..................................... 73
zero_Bi_deriv ............................... 73
zero_J ....................................... 73
zerotransform .............................. 50
zerowinding ............................... 44
zeta .......................................... 73
zpart ......................................... 29
zscale3 ...................................... 148
Z .................................................. 145
ZX ............................................... 148
ZY ............................................... 148