

Asymptote: the Vector Graphics Language

For version 2.84

This file documents **Asymptote**, version 2.84.

<https://asymptote.sourceforge.io>

Copyright © 2004-22 Andy Hammerlindl, John Bowman, and Tom Prince.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Lesser General Public License (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	Uninstall	8
3	Tutorial	9
3.1	Drawing in batch mode	9
3.2	Drawing in interactive mode	9
3.3	Figure size	10
3.4	Labels	10
3.5	Paths	11
4	Drawing commands	13
4.1	draw	13
4.2	fill	15
4.3	clip	17
4.4	label	17
5	Bezier curves	21
6	Programming	23
6.1	Data types	23
6.2	Paths and guides	30
6.3	Pens	37
6.4	Transforms	44
6.5	Frames and pictures	45
6.6	Files	51
6.7	Variable initializers	53
6.8	Structures	54
6.9	Operators	58
6.9.1	Arithmetic & logical operators	5

6.10	Implicit scaling.....	61
6.11	Functions.....	61
6.11.1	Default arguments.....	63
6.11.2	Named arguments.....	63
6.11.3	Rest arguments.....	64
6.11.4	Mathematical functions.....	66
6.12	Arrays.....	67
6.12.1	Slices.....	74
6.13	Casts.....	76
6.14	Import.....	77
6.15	Static.....	79
7	LaTeX usage.....	82
8	Base modules.....	86
8.1	plain.....	86
8.2	simplex.....	86
8.3	math.....	86
8.4	interpolate.....	87
8.5	geometry.....	87
8.6	trembling.....	87
8.7	stats.....	87
8.8	patterns.....	88
8.9	markers.....	88
8.10	map.....	88
8.11	tree.....	89
8.12	binarytree.....	89
8.13	drawtree.....	89
8.14	syzygy.....	89
8.15	feynman.....	89
8.16	roundedpath.....	90
8.17	animation.....	90
8.18	embed.....	90
8.19	slide.....	91
8.20	MetaPost.....	91
8.21	babel.....	91

8.33	tube.....	136
8.34	flowchart.....	137
8.35	contour.....	139
8.36	contour3.....	144
8.37	smoothcontour3.....	144
8.38	slopefield.....	144
8.39	ode.....	145
9	Command-line options.....	146
10	Interactive mode.....	151
11	Graphical User Interface.....	153
11.1	GUI installation.....	153
11.2	GUI usage.....	153
12	Command-Line Interface.....	154
13	Language server protocol.....	155
14	PostScript to Asymptote.....	156
15	Help.....	157
16	Debugger.....	158
17	Acknowledgments.....	159
	Index.....	160

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 154).

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 13).

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 **T_EX**/**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

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 8.26 [graph], page 92). Examples of **Asymptote** code and output, including animations, are available at

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

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

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

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

A quick reference card for **Asymptote** is available at

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

2 Installation

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

We recommend subscribing to new release announcements at

<https://sourceforge.net/projects/asymptote>

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

<https://sourceforge.net/p/asymptote/discussion/409349>

2.1 UNIX binary distributions

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

```
tar -C / -zxf asymptote-x.xx.x86_64.tgz
texhash
```

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

Fedora users can easily install a recent version of **Asymptote** with the command

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

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

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

2.2 MacOS X binary distributions

MacOS X users can either compile the **UNIX** source code (see Section 2.6

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 [convert], page 149). 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 153).

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

```
import settings;
pdfviewer="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

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

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

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

By default, `Asymptote` attempts to center the figure on the page, assuming that the paper type is `letter`. The default paper type may be changed to `a4` with the configuration variable `papertype`. Alignment to other paper sizes can be obtained by setting the configuration variables `paperwidth` and `paperheight`.

These additional configuration variables normally do not require adjustment:

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

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

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

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

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

```
asy -psviewer=evince -V venn
```

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

```
ASYMPTOTE_PAPERTYPE="a4";
```

under Microsoft Windows XP:

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

2.5 Search paths

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

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

2.6 Compiling from UNIX source

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

<https://sourceforge.net/projects/asymptote/files/>
execute the commands:

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

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
```

<https://asymptote.sourceforge.io/asymptote.pdf>

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

For a (default) system-wide installation, the last command should be done as the root user. To install without root privileges, change the `./configure` command to

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

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

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

If you are compiling `Asymptote` with `gcc`, you will need a 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/share/asymptote` or `C:\Program Files\Asymptote` under MSDOS):

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

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

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

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

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

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

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

2.8 Git

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

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

```
cd asymptote
./autogen.sh
./configure
make all
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 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:

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

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

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

3.1 Drawing in batch mode

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

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

Then execute the command

```
asy -V test
```

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

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

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

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

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

3.2 Drawing in interactive mode

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

3.3 Figure size

In **Asymptote**, coordinates like $(0,0)$ and $(100,100)$, called *pairs*, are expressed in **PostScript** "big points" ($1\text{ bp} = 1/72\text{ inch}$) and the default line width is 0.5 bp . 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 $(0,0) \text{--} (1,1)$ to fit a rectangle of width 100.5 bp and height 100.5 bp (the extra 0.5 bp accounts for the line width):

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

One can also specify the size in **pt** ($1\text{ pt} = 1/72.27\text{ inch}$), **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 46):

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

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

```
size(3cm);
draw((0,0)--(1,0)--(1,1)--(0,1)--cycle);
```

For convenience, the path $(0,0) \text{--} (1,0) \text{--} (1,1) \text{--} (0,1) \text{--} \text{cycle}$ may be replaced with the predefined variable **unitsquare**, or equivalently, **box** $((0,0), (1,1))$.

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

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

3.4 Labels

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

</

defined as pairs in the **Asymptote** base module **plain** (a user who has a local variable named **E** may access the compass direction **E** by prefixing it with the name of the module where it is defined: **plain.E**).

3.5 Paths

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

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

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

In general, a path is specified as a list of points (or other paths) interconnected with **--**, which denotes a straight line segment, or **..**, which denotes a cubic spline (see Chapter 5 [Bezier curves], page 21). 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 40). 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 8.28 [three], page 119):

```
import three;

currentprojection=orthographic(5,4,
```

```
size3(3cm,5cm,8cm);  
  
draw(unitbox);  
  
dot(unitbox,red);  
  
label("$0$", (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 8.26 [graph], page 92 (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/asymptote>.

4 Drawing commands

All of **Asymptote**'s graphical capabilities are based on four primitive commands. The three **PostScript** drawing commands **draw**, **fill**, and **clip** add objects to a picture in the order in which they are executed, with the most recently drawn object appearing on top. The labeling command **label** can be used to add text labels and external EPS images, which will appear on top of the **PostScript** objects (since this is normally what one wants), but again in the relative order in which they were executed. After drawing objects on a picture, the picture can be output with the **shipout** function (see [shipout], page 46).

If you wish to draw **PostScript** objects on top of labels (or verbatim **tex** commands; see [tex], page 50), the **layer** command may be used to start a new **PostScript/LaTeX** layer:

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

The

of the arrow should be placed. The default arrowhead size when drawn with a pen `p` is `arrowsize(p)`. There are also arrow versions with slightly modified default values of `size` and `angle` suitable for curved arrows: `BeginArcArrow`, `EndArcArrow` (or equivalently `ArcArrow`), `MidArcArrow`, and `ArcArrows`.

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

The use of arrows, bars, and margins is illustrated by the examples `Pythagoras.asy` and `sqrtx01.asy`.

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

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

Here `xmargin` and `ymargin` specify the surrounding x and y margins, `perline` specifies the number of entries per line (default 1; 0 means choose this number automatically), `linelength` specifies the length of the path lines, `hskip` and `vskip` specify the line skip (as a multiple of the legend entry size), `maxwidth` and `maxheight` specify optional upper limits on the width and height of the resulting legend (0 means unlimited), <

```

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

```

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

```

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

```

See [pathmarkers], page 100 and Section 8.9 [markers], page 88 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);

```

```
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);
```

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

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

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

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

More general shading possibilities are available using \TeX engines that produce PDF output (see [texengines], page 149): the routine

```
void functionsshade(picture pic=currentpicture, path[] g, bool stroke=false,
                   pen fillrule=currentpen, string
```

```
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 44). For example, `rotate(45)*xscale(2)*L` first scales `L` in the x direction and then rotates it counter-clockwise 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

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 labels to match a given template, so that their baselines will be typeset on a horizontal line. See `Pythagoras.asy` for an example.

One can prevent labels from overwriting one another with the `overwrite` pen attribute (see [overwrite], page 43).

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` (see [envelope], page 45).

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`

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

5 Bezier curves

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

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

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

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

The `MetaPost` `...` path connector, which requests, when possible, an inflection-free curve confined to a triangle defined by the endpoints and directions, is implemented in `Asymptote` as the convenient abbreviation `::` for `..tension atleast 1 ..` (the ellipsis `...` is used in `Asymptote` to indicate a variable number of arguments; see Section 6.11.3 [Rest arguments], page 64). For example, compare

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

```
with
```

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

The `---` connector is an abbreviation for `..tension atleast infinity..` and the `&` connector concatenates two paths, after first stripping off the last node of the first path (which normally should coincide with the first node of the second path).

6 Programming

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

```
// This is a comment.

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

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

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

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

Asymptote supports **while**, **do**, **break**, and **continue** statements just as in C/C++. It also supports the Java-style shorthand for iterating over all elements of an array:

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

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

6.1 Data types

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

void	The void type is used only by functions that take or return no arguments.
bool	<p>a boolean type that can only take on the values true or false. For example:</p> <pre>bool b=true;</pre> <p>defines a boolean variable b and initializes it to the value true. If no initializer is given:</p> <pre>bool b;</pre> <p>the value false is assumed.</p>

bool3	an extended boolean type that can take on the values true , default , or false . A bool3 type can be cast to or from a bool . The default initializer for bool3 is default .
int	an integer type; if no initializer is given, the implicit value 0 is assumed. The minimum allowed value of an integer is intMin and the maximum value is intMax .
real	a real number; this should be set to the highest-precision native floating-point type on the architecture. The implicit initializer for reals is 0.0. Real numbers have precision realEpsilon , with realDigits significant digits. The smallest positive real number is realMin and the largest positive real number is realMax . The variables inf and nan , along with the function bool isnan(real x) are useful when floating-point exceptions are masked with the -mask command-line option (the default in interactive mode).
pair	complex number, that is, an ordered pair of real components (x,y). The real and imaginary parts of a pair z can read as z.x and z.y . We say that x and y are virtual members of the data element pair; they cannot be directly modified, however. The implicit initializer for pairs is (0.0,0.0).

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

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

Here **I** is the pair (0,1). A number of built-in functions are defined for pairs:

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

```
real length(pair z)
    returns the complex modulus  $|z|$  of its argument z. For example,
        pair z=(3,4);
        length(z);
    returns the result 5. A synonym for length(pair) is abs(pair).
    The function abs2(pair z) returns  $|z|^2$ ;
```

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

```
real degrees(pair z, bool warn=true)
    returns the angle of z in degrees in the interval
```

```

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

real xpart(pair z)
    returns z.x;

real ypart(pair z)
    returns z.y;

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

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

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

real orient(pair a, pair b, pair c);
    returns a positive (negative) value if a--b--c--cycle is oriented
    counterclockwise (clockwise) or zero if all three points are colinear.
    Equivalently, a positive (negative) value is returned if c lies to the
    left (right) of the line through a and b or zero if c lies on this line.
    The value returned can be expressed in terms of the 2D scalar cross
    product as cross(a-c,b-c), which is the determinant
    
$$\begin{vmatrix} a.x & a.y & 1 \\ b.x & b.y & 1 \\ c.x & c.y & 1 \end{vmatrix}$$


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
    
$$\begin{vmatrix} a.x & a.y & a.x^2+a.y^2 & 1 \\ b.x & b.y & b.x^2+b.y^2 & 1 \\ c.x & c.y & c.x^2+c.y^2 & 1 \\ d.x & d.y & d.x^2+d.y^2 & 1 \end{vmatrix}$$


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 `babel` package, see Section 8.21 [babel], page 91):

- `\"` maps to `"`
- `\\` maps to `\`

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

- `\'` maps to `'`
- `\"` maps to `"`
- `\?` maps to `?`
- `\\` maps to backslash
- `\a` maps to alert
- `\b` maps to backspace
- `\f` maps to form feed
- `\n` maps to newline
- `\r` maps to carriage return
- `\t` maps to tab
- `\v` maps to vertical tab
- `\0-\377` map to corresponding octal byte
- `\x0-\xFF` map to corresponding hexadecimal byte

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

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

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

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

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

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

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

```

string reverse(string s)
    returns the string formed by reversing string s;

string replace(string s, string before, string after)
    returns a string with all occurrences of the string before in the
    string s changed to the string after;

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

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

string[] array(string s)
    returns an array of strings obtained by splitting s into individ-
    ual 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 behaviour 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 behaviour of the C function
    fprintf), except that only one data field is allowed, trailing zeros
    are removed by default (unless # is specified), and if s specifies
    math mode or forcemath=true, TeX is used to typeset scientific
    notation using the defaultseparator="\!\times\!";

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

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

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

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

string time(string format="%a %b %d %T %Z %Y")
    returns the current time formatted by the ANSI C routine strftime
    according to the string format using the current locale. Thus

```

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

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

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

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

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

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

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

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

```
void asy(string format, bool overwrite=false ... string[] s)
    conditionally process each file name in array s in a new envi-
    ronment, 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 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 6.11 [Functions], page 61).

6.2 Paths and guides

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

For example, the routine **circle(pair c, real r)**, which returns a Bezier curve approximating a circle of radius **r** centered on **c**, is based on **unitcircle** (see [unitcircle], page 11):

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

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

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

A circular arc consistent with **circle** centered on **c** with radius **r** from **angle1** to **angle2** degrees, drawing counterclockwise if **angle2** \geq **angle1**, can be constructed with

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

One may also specify the direction explicitly:

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

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

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

If high accuracy is needed, true arcs may be produced with routines in the module **graph** that produce Bezier curves with

```

path ellipse(pair c, real a, real b)
{
    return shift(c)*scale(a,b)*unitcircle;
}
A brace can be constructed between pairs a and b with
path brace(pair a, pair b, real amplitude=bracedefaultratio*length(b-a));
This example illustrates the use of all five guide connectors discussed in
Chapter 3 [Tutorial], page 9 and Chapter 5 [Bezier curves], page 21:
size(300,0);
pair[] z=new pair[10];

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

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

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

draw(p,greyscale(0.5),linewidth(4mm));

dot(z);

```

Here are some useful functions for paths:

```

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

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

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

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

bool piecewisestraight(path p)
    returns true iff the path p is piecewise straight.

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

```

```

pair point(path p, real t);
    This returns the coordinates of the point between node floor(t)
    and floor(t)+1 corresponding to the cubic spline parameter
    t-floor(t) (see Chapter 5 [Bezier curves], page 21). If t lies
    outside the range [0,length(p)], it is first reduced modulo
    length(p) in the case where p is cyclic or else converted to the
    corresponding endpoint of p.

pair dir(path p, int t, int sign=0, bool normalize=true);
    If sign < 0, return the direction (as a pair) of the incoming tangent
    to path p at node t; if sign > 0, return the direction of the outgoing
    tangent. If sign=0, the mean of these two directions is returned.

pair dir(path p, real t, bool normalize=true);
    returns the direction of the tangent to path p at the point between
    node floor(t) and floor(t)+1 corresponding to the cubic spline
    parameter t-floor(t) (see Chapter 5 [Bezier curves], page 21).

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
```

cumulative arclength (measured from the beginning of the path) equals `L`.

```
pair arcpoint(path p, real L);
    returns point(p,arctime(p,L)).

real dirstime(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
```

within the absolute error specified by `fuzz`, or if `fuzz < 0`, to machine precision.

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

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

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

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

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

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

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

```

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 40).

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 40), -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 cyclic path p according
    to
```

```

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`

```
pen gray(real g);
```

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

```
pen rgb(real r, real g, real b);
```

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

```
pen RGB(int r, int g, int b);
```

This produces an RGB color, where each of the red, green, and blue intensities `r`, `g`, `b`, lies in the interval `[0,255]`.

```
pen cmyk(real c, real
```


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 `-f pdf` output format option; other formats can be produced from the resulting PDF file with the ImageMagick `convert` program. Labels are always drawn with an opacity of 1. A simple example of transparent filling is provided in the example file `transparency.asy`.

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

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

To fill or draw using pattern name, use the pen pattern("name"). For example,
rectangular tilings can be constructed using
```

You may need to turn off aliasing in your **PostScript** viewer for patterns to appear correctly. Custom patterns can easily be constructed, following the examples in module **patterns**. The tiled pattern can even incorporate shading (see [gradient shading], page 15), as illustrated in this example (not included in the manual because not all printers support **PostScript 3**):

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

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

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

- One can specify a custom pen nib as an arbitrary polygonal path with **pen makepen(path)**; this path represents the mark to be drawn for paths containing a single point. This pen nib path can be recovered from a pen with **path nib(pen)**. Unlike in **MetaPost**, the path need not be convex:

```
size(200);
pen convex=makepen(scale
```

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

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

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

6.4 Transforms

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

`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

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

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

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

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

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

If

may be modified by changing the variable `orientation`. To output in landscape mode, simply set the variable `orientation=Landscape` or issue the command `shipout(Landscape)`;

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

A picture `pic` can be explicitly fit to a frame by calling

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

The default size and aspect ratio settings are those given to the `size` command (which default to 0, 0, and `true`, respectively). The transformation that would currently be used to fit a picture `pic` to a frame is returned by the member function `pic.calculateTransform()`.

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

```
filltype filltype=NoFill, bool above=true);
```

The first example adds `src` to `currentpicture`; the second one adds `src` to `dest`. The `group` option specifies whether or not the graphical user interface should treat all of the elements of `src` as a single entity (see Chapter 11 [GUI], page 153), `filltype` requests optional background filling or clipping, and `above` specifies whether to add `src` above or below existing objects.

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

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

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 <

- 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

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:

equivalent to `alias` and `!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:

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

S s;                                // Initializes s with new S;

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

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

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

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

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

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

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

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

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

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

```
}
```

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

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

```
from Person unravel Person;
```

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

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

If, in the body of a structure, Asymptote encounters the

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

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

Much like in C++, casting (see Section 6.13 [Casts], page 76) provides for an elegant implementation of structure inheritance, including virtual functions:

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

6.9 Operators

6.9.1 Arithmetic & logical operators

`Asymptote` uses the standard binary arithmetic operators. However, when one integer is divided by another, both arguments are converted to real values before dividing and a real

quotient is returned (since this is 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 <code>quotient(x,y)</code> . Noting that the Python3 community adopted our comment symbol (<code>//</code>) 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 <code>q*(p # q)+p % q == p</code> for all integers <code>p</code> and nonzero integers <code>q</code> .
^	power; if the exponent (second argument) is an int, recursive multiplication is used; otherwise, logarithms and exponentials are used (<code>**</code> is a synonym for <code>^</code>).

6.9.2 Self & prefix operators

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

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

is equivalent to the code

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

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

6.10 Implicit scaling

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

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

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

This produces the output

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

6.11 Functions

```

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

```

3. Casting can be used to resolve ambiguities:

```

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

```

```

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

```

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

```

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

```


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

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

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

is equivalent to the statements

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

and outputs 43.

Parameters can be specified as “keyword-only” by putting **keyword** immediately before the parameter name, as in `int f(int keyword x)` or `int f(int keyword x=`


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

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

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

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

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

In this example,

```
int pow
```

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

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

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

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

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

```
struct
```



```

size(4cm,0);
pen colour1=red;
pen colour2=green;

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

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

add(intersection);

draw(c1);
draw(c2);

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

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

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

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



```
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"));
```

8.11 tree

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

8.12 binarytree

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

```
import binarytree;

picture pic,pic2;

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

binarytree st=searchtree(10,
```


