Sage Installation Guide

Release 6.3

The Sage Development Team

August 11, 2014

CONTENTS

This is a brief explanation of the installation of Sage. Once Sage is installed, you can easily upgrade to a more recent version using `sage -upgrade` (or type `sage -h` for more options regarding the installation of Sage packages.)

CHAPTER

ONE

INTRODUCTION

You can install Sage either from a pre-built binary tarball or from source. The binary method is fastest, and has the fewest prerequisites. The source method gives you access to possibly a slightly more up-to-date version of Sage, and ensures that you can modify any of the Sage source code and recompile. Also, installing from source should be simpler than you're used to with most software, since much testing is done to make sure Sage and all its components can be successfully compiled with no user interaction on a range of computers. Thus it's probably easier for you to build all of Sage from source than it would be for you to build some of the packages that come with Sage (e.g., Singular).

The Sage distribution includes most programs on which Sage depends – see a partial list below. These programs are all released under a license compatible with the GNU General Public License (GPL), version 3. See the COPYING.txt file in the Sage root directory for more details.

Here is a list of some of the software included with Sage:

- atlas: The ATLAS (Automatically Tuned Linear Algebra Software) project
- bzip2: bzip2 compression library
- ecl: common lisp interpreter
- cython: the Cython programming language: a language, based on Pyrex, for easily writing C extensions for Python
- eclib: John Cremona's programs for enumerating and computing with elliptic curves defined over the rational numbers
- ecm: elliptic curve method for integer factorization
- flint: fast library for number theory
- GAP: A System for Computational Discrete Algebra
- GCC: GNU compiler collection containing C, C++ and Fortran compilers
- genus2reduction: Reduction information about genus 2 curves
- gfan: Computation of Groebner fans and toric varieties
- givaro: a C++ library for arithmetic and algebraic computations
- mpir: MPIR is an open source multiprecision integer library derived from GMP (the GNU multiprecision library)
- gsl: GNU Scientific Library is a numerical library for C and C++ programmers
- ipython: An enhanced Python shell designed for efficient interactive work, a library to build customized interactive environments using Python as the basic language, and a system for interactive distributed and parallel computing
- jmol: a Java molecular viewer for three-dimensional chemical structures

- lapack: a library of Fortran 77 subroutines for solving the most commonly occurring problems in numerical linear algebra.
- lcalc: Rubinstein's L-functions calculator
- libfplll: contains different implementations of the floating-point LLL reduction algorithm, offering different speed/guarantees ratios
- libm4ri: Library for matrix multiplication, reduction and inversion over GF(2)
- linbox: C++ template library for exact, high-performance linear algebra computation
- mathjax: Javascript display engine for mathematics
- matplotlib: a Python 2-D plotting library
- maxima: symbolic algebra and calculus
- mercurial: a Source Control Management system designed for handling of very large distributed projects
- mpfi: a C library for arithmetic by multi-precision intervals, based on MPFR and GMP
- mpfr: a C library for multiple-precision floating-point computations with correct rounding
- networkx: a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks
- NTL: number theory C++ library
- numpy: numerical linear algebra and other numerical computing capabilities for Python
- palp: a package for analyzing lattice polytopes
- pari: PARI number theory library
- pexpect: Python expect (for remote control of other systems)
- polybori: provide high-level data types for Boolean polynomials and monomials, exponent vectors, as well as for the underlying polynomial rings and subsets of the power set of the Boolean variables
- PPL: The Parma Polyhedra Library
- pynac: a modified version of GiNaC (a C++ library for symbolic mathematical calculations) that replaces the dependency on CLN by Python
- Python: The Python programming language
- R: a language and environment for statistical computing and graphics
- readline: GNU Readline line editor library
- scipy: scientific tools for Python
- singular: Polynomial computations in algebraic geometry, etc.
- symmetrica: routines for computing in the representation theory of classical and symmetric groups, and related areas
- sympow: Symmetric power L-functions and modular degrees
- sympy: a Python library for symbolic mathematics
- tachyon: Tachyon(tm) parallel/multiprocessor ray tracing software
- termcap: Display terminal library
- Twisted: Networking framework
- zlib: zlib compression library

- zn_poly: C library for polynomial arithmetic in $\mathbf{Z}/n\mathbf{Z}[x]$

CHAPTER

TWO

QUICK DOWNLOAD AND INSTALL GUIDE

Not sure what to download? This short guide should get you started.

- Determine your operating system (Windows, Linux, Mac OS X, etc.).
- Determine your CPU type (32-bit, 64-bit or "atom" for Linux and Intel, or PowerPC for Mac OS X).
- Do you want a source or binary distribution? Even if you want to do development, a precompiled version of Sage (binary release) can be used for that purpose. The source distribution is mostly needed if you want to see the sources of the Sage packages, also known as SPKGs.
- If available, choose the appropriate binary version from one of the download mirrors. A list of mirrors is maintained at http://www.sagemath.org/mirrors.html
- Follow the binary installation guide (http://www.sagemath.org/doc/installation/binary.html) to actually install a pre-compiled version of Sage. The source installation guide (http://www.sagemath.org/doc/installation/source.html) contains more detailed information on compiling Sage from source.

2.1 Troubleshooting

- If no binary version is available for your system, download the source version. Note that Sage compiles on a wide variety of systems, but does not compile on every system.
- If you have downloaded a binary version of Sage, upon loading Sage might complain about an `illegal instruction` error. In that case, a solution is available at the FAQ wiki page http://wiki.sagemath.org/faq#Otherquestions
- Make sure there are no spaces in the path in which you have installed Sage.
- Ask for help on the sage-support mailing list. This mailing list is also referred to as the sage-support Google group (http://groups.google.com/group/sage-support).

CHAPTER

THREE

PRE-BUILT BINARY INSTALL

3.1 Linux and OS X

Installation from a pre-built binary tarball should in the long run be the easiest and fastest way to install Sage. This is not necessarily the case right now. Note that Sage is itself a programming environment, so building it from source guarantees you maximum flexibility in the long run. Nonetheless, we provide pre-built binaries.

Assumptions: You have a computer with at least 2 GB of free disk space and the operating system is Linux (32-bit or 64-bit) or OS X (10.4 or later).

Highly Recommended: It is highly recommended that you have LaTeX installed. If you want to view animations, you should install either ImageMagick or ffmpeg. ImageMagick or dvipng is also used for displaying some LaTeX output in the Sage notebook.

Download the latest binary tarball from http://www.sagemath.org/download.html. For example, it might be called `sage-x.y.z-x86_64-Linux.tgz`. Unpack it on your computer in a directory which you have permission to read and write:

```
tar zxvf sage-x.y.z-x86_64-Linux.tgz
```

You can move the resulting directory `sage-x.y.z-x86_64-Linux` anywhere and still run `./sage` from it, as long as the full path name has **no spaces** in it. You can also copy the file `sage` from that directory and put it anywhere, e.g., `/usr/local/bin/`, but then you have to edit the `#SAGE_ROOT=/path/to/sage-version` line at the top of the copied file `/usr/local/bin/sage` (you should not edit the original `sage` executable). The variable `SAGE_ROOT` should point to the directory `sage-x.y.z-x86_64-Linux` of the extracted Sage tarball. As long as `/usr/local/bin` is in your `$PATH`, you can then type `sage` from the command line to run Sage. Another approach is to create a symbolic link, say `/usr/local/bin/sage`, pointing to `sage-x.y.z-x86_64-Linux/sage`

```
ln -s /path/to/sage-x.y.z-x86_64-Linux/sage /usr/local/bin/sage
```

With this approach, there is no need to edit `/usr/local/bin/sage`, the `SAGE_ROOT` path will be discovered automatically thanks to the symbolic link. When you want to install a new version of Sage, just delete the old link and create a new one.

Any time you move the Sage directory, you may see a message like

```
The Sage installation tree may have moved
(from /foo to /bar).
Changing various hardcoded paths...
(Please wait at most a few minutes.)
DO NOT INTERRUPT THIS.
```

We currently distribute `.dmg` files for OS X 10.4.x and 10.6.x. But we would like to make Sage more of a native application. Work for that is ongoing, but help is always welcome.

3.2 Microsoft Windows

The best way to install Sage on Windows is to install VirtualBox for Windows and then download and install the VirtualBox distribution of Sage. See this URL for further instructions on installing Sage on Windows. Be sure to read the file README.txt.

CHAPTER

FOUR

INSTALL FROM SOURCE CODE

More familiarity with computers may be required to build Sage from the source code. If you do have all the pre-requisite tools, the process should be completely painless, basically consisting in extracting the source tarball and typing `make`. It can take your computer a while to build Sage from the source code, although the procedure is fully automated and should need no human intervention. Building Sage from the source code has the major advantage that your install will be optimised for your particular computer and should therefore offer better performance and compatibility than a binary install. Moreover, it offers you full development capabilities: you can change absolutely any part of Sage or the programs on which it depends, and recompile the modified parts.

Download the Sage source code. If you changed your mind, you can also download a binary distribution for some operating systems.

4.1 Supported platforms

See http://wiki.sagemath.org/SupportedPlatforms for the full list of platforms on which Sage is supported and the level of support for these systems.

Sage is supported on a number of Linux, Mac OS X , Sun/Oracle Solaris releases, but not necessarily all versions of these operating systems. There is no native version of Sage which installs on Microsoft Windows, although Sage can be used on Windows with the aid of a virtual machine or the Cygwin Linux API layer.

On the list of supported platforms, you can find details about ports to other operating systems or processors which may be taking place.

4.2 Prerequisites

4.2.1 General requirements

Your computer comes with at least 5 GB of free disk space running one of the supported versions of an operating system listed at http://wiki.sagemath.org/SupportedPlatforms. It is recommended to have at least 2 GB of RAM, but you might get away with less (be sure to have some swap space in this case).

In addition to standard POSIX utilities and a bash-compatible shell, the following standard command-line development tools must be installed on your computer:

- A **C compiler**: Since Sage builds its own GCC if needed, a wide variety of C compilers is supported. Many GCC versions work, from as old as version 3.4.3 to the most recent release. Clang also works. On Solaris systems, the Sun compiler should also work. See also Using alternative compilers.
- **make**: GNU make, version 3.80 or later. Version 3.82 or later is recommended.

- **m4**: GNU m4 1.4.2 or later (non-GNU or older versions might also work).
- **perl**: version 5.8.0 or later.
- **ar** and **ranlib**: can be obtained as part of GNU binutils.
- **tar**: GNU tar version 1.17 or later, or BSD tar.

Sage also needs a C++ compiler and a Fortran compiler. The only configuration currently supported is matching versions of the C, C++ and Fortran compilers from the GNU Compiler Collection (GCC). Therefore, if you plan on using your own GCC compilers, then make sure that their versions match. Alternatively, Sage includes a GCC package, so that C, C++ and Fortran compilers will be built when the build system detects that it is needed, e.g., non-GCC compilers, or versions of the GCC compilers known to miscompile some components of Sage, or simply a missing C++ or Fortran compiler. Whatsoever, you always need at least a C compiler to build the GCC package and its prerequisites before the compilers it provides can be used. Note that you can always override this behavior through the environment variable `SAGE_INSTALL_GCC`, see *Using alternative compilers* and *Environment variables*.

Although some of Sage is written in Python, you do not need Python pre-installed on your computer, since the Sage installation includes virtually everything you need.

After extracting the Sage tarball, the subdirectory `spkg` contains the source distributions for everything on which Sage depends. We emphasize that all of this software is included with Sage, so you do not have to worry about trying to download and install any one of these packages (such as Python, for example) yourself.

When the Sage installation program is run, it will check that you have each of the above-listed prerequisites, and inform you of any that are missing, or have unsuitable versions.

4.2.2 System-specific requirements

On recent Debian or Ubuntu systems, the **dpkg-dev** package is needed for multiarch support.

On Cygwin, the **lapack** and **liblapack-devel** packages are required to provide ATLAS support as the ATLAS spkg is not built by default.

4.2.3 Installing prerequisites

To check if you have the above prerequisites installed, for example `perl`, type:

```
command -v perl
```

or:

```
which perl
```

on the command line. If it gives an error (or returns nothing), then either `perl` is not installed, or it is installed but not in your PATH.

On Linux systems (e.g., Ubuntu, Redhat, etc), `ar` and `ranlib` are in the binutils package. The other programs are usually located in packages with their respective names. Assuming you have sufficient privileges, you can install the `binutils` and other necessary components. If you do not have the privileges to do this, ask your system administrator to do this, or build the components from source code. The method of installing additional software varies from distribution to distribution, but on a Debian based system (e.g. Ubuntu or Mint), you would use apt-get:

```
sudo apt-get install binutils gcc make m4 perl tar
```

to install all general requirements (this was tested on Ubuntu 12.04.2). On other Linux systems, you might use rpm, yum, or other package managers.

On OS X systems, you need a recent version of Command Line Tools. It provides all the above requirements. You can download it for free at http://developer.apple.com/downloads/index.action?=command%20line%20tools provided you registered for a free Apple Developer account at http://developer.apple.com/register/. Alternatively, if you have already installed Xcode (which at the time of writing is freely available in the Mac App Store, or through http://developer.apple.com/downloads/ provided you registered for an Apple Developer account), you can install the command line tools from there: with OS X Mavericks, run the command `xcode-select --install` from a Terminal window and click "Install" in the pop-up dialog box. Using OS X Mountain Lion or earlier, run Xcode, open its "Downloads" preference pane and install the command line tools from there. On pre-Lion OS X systems, the command line tools are not available as a separate download and you have to install the full-blown Xcode supporting your system version.

On Solaris, you would use `pkgadd` and on OpenSolaris `ipf` to install the necessary software.

On Cygwin, you would use the `setup.exe` program. As on Linux systems, `ar` and `ranlib` are provided by the `binutils` package. As far as compilers are concerned, you should either install matching versions of the `gcc4-core`, `gcc4-g++`, and `gcc4-gfortran` packages, or the `gcc4-core` package alone if you plan on using Sage's own GCC.

On other systems, check the documentation for your particular operating system.

4.2.4 Specific notes for `make` and `tar`

On OS X, the system-wide BSD `tar` supplied will build Sage, so there is no need to install the GNU `tar`.

On Solaris or OpenSolaris, the Sun/Oracle versions of `make` and `tar` are unsuitable for building Sage. Therefore, you must have the GNU versions of `make` and `tar` installed and they must be the first `make` and `tar` in your `PATH`.

On Solaris 10, a version of GNU `make` may be found at `/usr/sfw/bin/gmake`, but you will need to copy it somewhere else and rename it to `make`. The same is true for GNU `tar`; a version of GNU `tar` may be found at `/usr/sfw/bin/gtar`, but it will need to be copied somewhere else and renamed to `tar`. It is recommended to create a directory `$HOME/bins-for-sage` and to put the GNU versions of `tar` and `make` in that directory. Then ensure that `$HOME/bins-for-sage` is first in your `PATH`. That's because Sage also needs `/usr/ccs/bin` in your `PATH` to execute programs like `ar` and `ranlib`, but `/usr/ccs/bin` has the Sun/Oracle versions of `make` and `tar` in it.

If you attempt to build Sage on AIX or HP-UX, you will need to install both GNU `make` and GNU `tar`.

4.2.5 Using alternative compilers

Sage developers tend to use fairly recent versions of GCC. Nonetheless, the Sage build process should succeed with any reasonable C compiler. This is because Sage will build GCC first (if needed) and then use that newly built GCC to compile Sage.

If you don't want this and want to try building Sage with a different set of compilers, you need to set the environment variable `SAGE_INSTALL_GCC` to `no`. Make sure you have C, C++ and Fortran compilers installed!

Building all of Sage with Clang is currently not supported, see trac ticket #12426.

If you are interested in working on support for commerical compilers from HP, IBM, Intel, Sun/Oracle, etc, please email the sage-devel mailing list at http://groups.google.com/group/sage-devel.

4.3 Additional software

4.3.1 Recommended programs

The following programs are recommended. They are not strictly required at build time or at run time, but provide additional capablities:

- **dvipng**.
- **ffmpeg**.
- **ImageMagick**.
- **latex**: highly recommended.

It is highly recommended that you have Latex installed, but it is not required.

On Linux systems, it is usually provided by packages derived from TeX Live and can be installed using:

```
sudo apt-get install texlive
```

or similar commands.

On other systems it might be necessary to install TeX Live from source code, which is quite easy, though a rather time-consuming process.

If you don't have either ImageMagick or ffmpeg, you won't be able to view animations. ffmpeg can produce animations in more different formats than ImageMagick, and seems to be faster than ImageMagick when creating animated GIFs. Either ImageMagick or dvipng is used for displaying some LaTeX output in the Sage notebook.

4.3.2 Notebook additional features

By default, the Sage notebook uses the HTTP protocol when you type the command `notebook()`. To run the notebook in secure mode by typing `notebook(secure=True)` which uses the HTTPS protocol, or to use OpenID authentication, you need to follow specific installation steps described in *Building the notebook with SSL support*.

Although all necessary components are provided through Sage optional packages, i.e. you can install a local version of OpenSSL by using Sage's **openssl** spkg and running `sage -i openssl` as suggested in *Building the notebook with SSL support* (this requires an Internet connection), you might prefer to install OpenSSL and the OpenSSL development headers globally on your system.

On Linux systems, those are usually provided by the **libssl** and **libssl-dev** packages and can be installed using:

```
sudo apt-get install libssl libssl-dev
```

or similar commands.

Finally, if you intend to distribute the notebook load onto several Sage servers, you will surely want to setup an SSH server and generate SSH keys. This can be achieved using OpenSSH.

On Linux systems, the OpenSSH server, client and utilities are usually provided by the **openssh-server** and **openssh-client** packages and can be installed using:

```
sudo apt-get install openssh-server openssh-client
```

or similar commands.

4.3.3 Tcl/Tk

If you want to use Tcl/Tk libraries in Sage, you need to install the Tcl/Tk and its development headers before building Sage. Sage's Python will then automatically recognize your system's install of Tcl/Tk.

On Linux systems, these are usually provided by the **tk** and **tk-dev** (or **tk-devel**) packages which can be installed using:

```
sudo apt-get install tk tk-dev
```

or similar commands.

If you installed Sage first, all is not lost. You just need to rebuild Sage's Python, , and ideally any part of Sage relying on it:

```
sage -f python  # rebuild Python
SAGE_UPGRADING=yes make # rebuild components of Sage depending on Python
```

after installing the Tcl/Tk development libraries as above.

If

```
sage: import _tkinter
sage: import Tkinter
```

does not raise an `ImportError`, then it worked.

4.4 Step-by-step installation procedure

4.4.1 General procedure

Installation from source is (potentially) very easy, because the distribution contains (essentially) everything on which Sage depends.

Make sure there are **no spaces** in the path name for the directory in which you build: several of Sage's components will not build if there are spaces in the path. Running Sage from a directory with spaces in its name will also fail.

1. Go to http://www.sagemath.org/download-source.html, select a mirror, and download the file `sage-x.y.z.tar`.

   This tarfile contains the source code for Sage and the source for all programs on which Sage depends. Note that this file is not compressed; it's just a plain tarball (which happens to be full of compressed files).

   Download it into any directory you have write access to, preferably on a fast filesystem, avoiding NFS and the like. On personal computers, any subdirectory of your `HOME` directory should do. The directory where you built Sage is **NOT** hardcoded. You should be able to safely move or rename that directory. (It's a bug if this is not the case.)

2. Extract the tarfile:

   ```
   tar xvf sage-x.y.z.tar
   ```

   This creates a directory `sage-x.y.z`.

3. Change into that directory:

   ```
   cd sage-x.y.z
   ```

   This is Sage's home directory. It is also referred to as `SAGE_ROOT` or the top level Sage directory.

4. Optional, but highly recommended: Read the `README.txt` file there.
5. On OSX 10.4, OS 10.5, Solaris 10 and OpenSolaris, if you wish to build a 64-bit version of Sage, assuming your computer and operating system are 64-bit, type:

   ```
   export SAGE64=yes
   ```

   It should be noted that as of April 2011, 64-bit builds of Sage on both Solaris 10 and OpenSolaris are not very stable, so you are advised not to set `SAGE64` to `yes`. This will then create stable 32-bit versions of Sage. See http://wiki.sagemath.org/solaris for the latest information.
6. Start the build process:

   ```
   make
   ```

   or if your system is multithreaded and you want to use several threads to build Sage:

   ```
   MAKE='make -jNUM' make
   ```

   to tell the `make` program to run `NUM` jobs in parallel when building Sage. This compiles Sage and all its dependencies.

   Note that you do not need to be logged in as root, since no files are changed outside of the `sage-x.y.z` directory. In fact, **it is inadvisable to build Sage as root**, as the root account should only be used when absolutely necessary and mistyped commands can have serious consequences if you are logged in as root. There has been a bug reported (see trac ticket #9551) in Sage which would have overwritten a system file had the user been logged in as root.

   Typing `make` performs the usual steps for each Sage's dependency, but installs all the resulting files into the local build tree. Depending on the age and the architecture of your system, it can take from a few tens of minutes to several hours to build Sage from source. On really slow hardware, it can even take a few days to build Sage.

   Each component of Sage has its own build log, saved in `SAGE_ROOT/logs/pkgs`. If the build of Sage fails, you will see a message mentioning which package(s) failed to build and the location of the log file for each failed package. If this happens, then paste the contents of these log file(s) to the Sage support newsgroup at http://groups.google.com/group/sage-support. If the log files are very large (and many are), then don't paste the whole file, but make sure to include any error messages. It would also be helpful to include the type of operating system (Linux, OS X, Solaris, OpenSolaris, Cygwin, or any other system), the version and release date of that operating system and the version of the copy of Sage you are using. (There are no formal requirements for bug reports – just send them; we appreciate everything.)

   See *Make targets* for some targets for the `make` command, *Environment variables* for additional informatio on useful environment variables used by Sage, and *Building the notebook with SSL support* for additional instruction on how to build the notebook with SSL support.
7. To start Sage, you can now simply type from Sage's home directory:

   ```
   ./sage
   ```

   You should see the Sage prompt, which will look something like this:

   ```
   $ sage
   ----------------------------------------------------------------------
   | Sage Version 5.8, Release Date: 2013-03-15                         |
   | Type "notebook()" for the browser-based notebook interface.        |
   | Type "help()" for help.                                            |
   ----------------------------------------------------------------------
   sage:
   ```

   Note that Sage should take well under a minute when it starts for the first time, but can take several minutes if the file system is slow or busy. Since Sage opens a lot of files, it is preferable to install Sage on a fast filesystem if possible.

Just starting successfully tests that many of the components built correctly. Note that this should have been already automatically tested during the build process. If the above is not displayed (e.g., if you get a massive traceback), please report the problem, e.g., at http://groups.google.com/group/sage-support.

After Sage has started, try a simple command:

```
sage: 2 + 2
4
```

Or something slightly more complicated:

```
sage: factor(2005)
5 * 401
```

8. Optional, but highly recommended: Test the install by typing `./sage --testall`. This runs most examples in the source code and makes sure that they run exactly as claimed. To test all examples, use `./sage --testall --optional=all --long`; this will run examples that take a long time, and those that depend on optional packages and software, e.g., Mathematica or Magma. Some (optional) examples will therefore likely fail.

   Alternatively, from within `$SAGE_ROOT`, you can type `make test` (respectively `make ptest`) to run all the standard test code serially (respectively in parallel).

   Testing the Sage library can take from half an hour to several hours, depending on your hardware. On slow hardware building and testing Sage can even take several days!

9. Optional: Check the interfaces to any other software that you have available. Note that each interface calls its corresponding program by a particular name: Mathematica is invoked by calling `math`, Maple by calling `maple`, etc. The easiest way to change this name or perform other customizations is to create a redirection script in `$SAGE_ROOT/local/bin`. Sage inserts this directory at the front of your `PATH`, so your script may need to use an absolute path to avoid calling itself; also, your script should pass along all of its arguments. For example, a `maple` script might look like:

   ```
   #!/bin/sh

   exec /etc/maple10.2/maple.tty "$@"
   ```

10. Optional: There are different possibilities to make using Sage a little easier:

    - Make a symbolic link from `/usr/local/bin/sage` (or another directory in your `PATH`) to `$SAGE_ROOT/sage`:

      ```
      ln -s /path/to/sage-x.y.z/sage /usr/local/bin/sage
      ```

      Now simply typing `sage` from any directory should be sufficient to run Sage.

    - Copy `$SAGE_ROOT/sage` to a location in your `PATH`. If you do this, make sure you edit the line:

      ```
      #SAGE_ROOT=/path/to/sage-version
      ```

      at the beginning of the copied `sage` script according to the direction given there to something like:

      ```
      SAGE_ROOT=<SAGE_ROOT>
      ```

      (note that you have to change `<SAGE_ROOT>` above!). It is best to edit only the copy, not the original.

    - For KDE users, create a bash script called `sage` containing the lines (note that you have to change `<SAGE_ROOT>` below!):

```
#!/bin/bash

konsole -T "sage" -e <SAGE_ROOT>/sage
```

make it executable:

```
chmod a+x sage
```

and put it somewhere in your `PATH`.

You can also make a KDE desktop icon with this line as the command (under the Application tab of the Properties of the icon, which you get my right clicking the mouse on the icon).

- On Linux and OS X systems, you can make an alias to `$SAGE_ROOT/sage`. For example, put something similar to the following line in your `.bashrc` file:

  ```
  alias sage=<SAGE_ROOT>/sage
  ```

  (Note that you have to change `<SAGE_ROOT>` above!) Having done so, quit your terminal emulator and restart it. Now typing `sage` within your terminal emulator should start Sage.

11. Optional: Install optional Sage packages and databases. Type `sage --optional` to see a list of them (this requires an Internet connection), or visit http://www.sagemath.org/packages/optional/. Then type `sage -i <package-name>` to automatically download and install a given package.
12. Optional: Run the `install_scripts` command from within Sage to create GAP, GP, Maxima, Singular, etc., scripts in your `PATH`. Type `install_scripts?` in Sage for details.
13. Have fun! Discover some amazing conjectures!

4.4.2 Building the notebook with SSL support

Read this section if you are intending to run a Sage notebook server for multiple users.

For security, you may wish users to access the server using the HTTPS protocol (i.e., to run `notebook(secure=True)`). You also may want to use OpenID for user authentication. The first of these requires you to install pyOpenSSL, and they both require OpenSSL.

If you have OpenSSL and the OpenSSL development headers installed on your system, you can install pyOpenSSL by building Sage and then typing:

```
./sage -i pyopenssl
```

Alternatively, `make ssl` builds Sage and installs pyOpenSSL at once. Note that these commands require Internet access.

If you are missing either OpenSSL or OpenSSL's development headers, you can install a local copy of both into your Sage installation first. Ideally, this should be done before installing Sage; otherwise, you should at least rebuild Sage's Python, and ideally any part of Sage relying on it. The procedure is as follows (again, with a computer connected to the Internet). Starting from a fresh Sage tarball:

```
./sage -i openssl
make ssl
```

And if you've already built Sage:

```
./sage -i openssl
./sage -f python
SAGE_UPGRADING=yes make ssl
```

The third line will rebuild all parts of Sage that depend on Python; this can take a while.

4.4.3 Rebasing issues on Cygwin

Building on Cygwin will occasionally require "rebasing" `dll` files. Sage provides some scripts, located in `$SAGE_LOCAL/bin`, to do so:

- `sage-rebaseall.sh`, a shell script which calls Cygwin's `rebaseall` program. It must be run within a `dash` shell from the `SAGE_ROOT` directory after all other Cygwin processes have been shut down and needs write-access to the system-wide rebase database located at `/etc/rebase.db.i386`, which usually means administrator privileges. It updates the system-wide database and adds Sage dlls to it, so that subsequent calls to `rebaseall` will take them into account.
- `sage-rebase.sh`, a shell script which calls Cygwin's `rebase` program together with the `-O/--oblivious` option. It must be run within a shell from `SAGE_ROOT` directory. Contrary to the `sage-rebaseall.sh` script, it neither updates the system-wide database, nor adds Sage dlls to it. Therefore, subsequent calls to `rebaseall` will not take them into account.
- `sage-rebaseall.bat` (respectively `sage-rebase.bat`), an MS-DOS batch file which calls the `sage-rebaseall.sh` (respectively `sage-rebase.sh`) script. It must be run from a Windows command prompt, after adjusting `SAGE_ROOT` to the Windows location of Sage's home directory, and, if Cygwin is installed in a non-standard location, adjusting `CYGWIN_ROOT` as well.

Some systems may encounter this problem frequently enough to make building or testing difficult. If executing the above scripts or directly calling `rebaseall` does not solve rebasing issues, deleting the system-wide database and then regenerating it from scratch, e.g., by executing `sage-rebaseall.sh`, might help.

Finally, on Cygwin, one should also avoid the following:

- building in home directories of Windows domain users;
- building in paths with capital letters (see trac ticket #13343, although there has been some success doing so).

4.5 Make targets

To build Sage from scratch, you would typically execute `make` in Sage's home directory to build Sage and its HTML documentation. The `make` command is pretty smart, so if your build of Sage is interrupted, then running `make` again should cause it to pick up where it left off. The `make` command can also be given options, which control what is built and how it is built:

- `make build` builds Sage: it compiles all of the Sage packages. It does not build the documentation.
- `make doc` builds Sage's documentation in HTML format. Note that this requires that Sage be built first, so it will automatically run `make build` first. Thus, running `make doc` is equivalent to running `make`.
- `make doc-pdf` builds Sage's documentation in PDF format. This also requires that Sage be built first, so it will automatically run `make build`.
- `make build-serial` builds the components of Sage serially, rather than in parallel (parallel building is the default). Running `make build-serial` is equivalent to setting the environment variable `SAGE_PARALLEL_SPKG_BUILD` to "no" – see below for information about this variable.
- `make ptest` and `make ptestlong`: these run Sage's test suite. The first version skips tests that need more than a few seconds to complete and those which depend on optional packages or additional software. The second version includes the former, and so it takes longer. The "p" in `ptest` stands for "parallel": tests are run in parallel. If you want to run tests serially, you can use `make test` or `make testlong` instead. If you

want to run tests depending on optional packages and additional software, you can use `make testall`, `make ptestall`, `make testalllong`, or `make ptestalllong`.

- `make distclean` restores the Sage directory to its state before doing any building: it is equivalent to deleting the entire Sage's home directory and unpacking the source tarfile again.

4.6 Environment variables

Sage uses several environment variables to control its build process. Most users won't need to set any of these: the build process just works on many platforms. (Note though that setting `MAKE`, as described below, can significantly speed up the process.) Building Sage involves building about 100 packages, each of which has its own compilation instructions.

Here are some of the more commonly used variables affecting the build process:

- `MAKE` - one useful setting for this variable when building Sage is `MAKE='make -jNUM'` to tell the `make` program to run `NUM` jobs in parallel when building. Note that not all Sage packages (e.g. ATLAS) support this variable.

  Some people advise using more jobs than there are CPU cores, at least if the system is not heavily loaded and has plenty of RAM; for example, a good setting for `NUM` might be between 1 and 1.5 times the number of cores. In addition, the `-l` option sets a load limit: `MAKE='make -j4 -l5.5`, for example, tells `make` to try to use four jobs, but to not start more than one job if the system load average is above 5.5. See the manual page for GNU `make`: Command-line options and Parallel building.

  Warning: Some users on single-core OS X machines have reported problems when building Sage with `MAKE='make -jNUM'` with `NUM` greater than one.

- `SAGE_NUM_THREADS` - if set to a number, then when building the documentation, parallel doctesting, or running `sage -b`, use this many threads. If this is not set, then determine the number of threads using the value of the `MAKE` (see above) or `MAKEFLAGS` environment variables. If none of these specifies a number of jobs, use one thread (except for parallel testing: there we use a default of the number of CPU cores, with a maximum of 8 and a minimum of 2).

- `SAGE_PARALLEL_SPKG_BUILD` - if set to `no`, then build spkgs serially rather than in parallel. If this is set to `no`, then each spkg may still take advantage of the setting of `MAKE` to build using multiple jobs, but the spkgs will be built one at a time. Alternatively, run `make build-serial` which sets this environment variable for you.

- `SAGE_CHECK` - if set to `yes`, then during the build process, and when running `sage -i <package-name>` or `sage -f <package-name>`, run the test suite for each package which has one. See also `SAGE_CHECK_PACKAGES`.

- `SAGE_CHECK_PACKAGES` - if `SAGE_CHECK` is set to `yes`, then the default behavior is to run test suites for all spkgs which contain them. If `SAGE_CHECK_PACKAGES` is set, it should be a comma-separated list of strings of the form `package-name` or `!package-name`. An entry `package-name` means to run the test suite for the named package regardless of the setting of `SAGE_CHECK`. An entry `!package-name` means to skip its test suite. So if this is set to `mpir,!python`, then always run the test suite for MPIR, but always skip the test suite for Python.

  Note: As of this writing (April 2013, Sage 5.8), the test suite for the Python spkg fails on most platforms. So when this variable is empty or unset, Sage uses a default of `!python`.

- `SAGE64` - if set to `yes`, then build a 64-bit binary on platforms which default to 32-bit, even though they can build 64-bit binaries. It adds the compiler flag `-m64` when compiling programs. The `SAGE64` variable is mainly of use on OS X (pre 10.6), Solaris and OpenSolaris, though it will add the `-m64` flag on any operating system. If you are running Linux or version 10.6 or later of OS X on a 64-bit machine, then Sage will automatically build a 64-bit binary, so this variable does not need to be set.

- `CFLAG64` - default value `-m64`. If Sage detects that it should build a 64-bit binary, then it uses this flag when compiling C code. Modify it if necessary for your system and C compiler. This should not be necessary on most systems – this flag will typically be set automatically, based on the setting of `SAGE64`, for example.
- `SAGE_INSTALL_GCC` - by default, Sage will automatically detect whether to install the GNU Compiler Collection (GCC) package or not (depending on whether C, C++ and Fortran compilers are present and the versions of those compilers). Setting `SAGE_INSTALL_GCC=yes` will force Sage to install GCC. Setting `SAGE_INSTALL_GCC=no` will prevent Sage from installing GCC.
- `SAGE_INSTALL_CCACHE` - by default Sage doesn't install ccache, however by setting `SAGE_INSTALL_CCACHE=yes` Sage will install ccache. Because the Sage distribution is quite large, the maximum cache is set to 4G. This can be changed by running `sage -sh -c "ccache --max-size=SIZE"`, where `SIZE` is specified in gigabytes, megabytes, or kilobytes by appending a "G", "M", or "K".

  Sage does not include the sources for ccache since it is an optional package. Because of this, it is necessary to have an Internet connection while building ccache for Sage, so that Sage can pull down the necessary sources.
- `SAGE_DEBUG` - controls debugging support. There are three different possible values:
  - Not set (or set to anything else than "yes" or "no"): build binaries with debugging symbols, but no special debug builds. This is the default. There is no performance impact, only additional disk space is used.
  - `SAGE_DEBUG=no`: `no` means no debugging symbols (that is, no `gcc -g`), which saves some disk space.
  - `SAGE_DEBUG=yes`: build debug versions if possible (in particular, Python is built with additional debugging turned on and Singular is built with a different memory manager). These will be notably slower but, for example, make it much easier to pinpoint memory allocation problems.
- `SAGE_SPKG_LIST_FILES` - if set to `yes`, then enable verbose extraction of tar files, i.e. Sage's spkg files. Since some spkgs contain such a huge number of files that the log files get very large and harder to search (and listing the contained files is usually less valuable), we decided to turn this off by default. This variable affects builds of Sage with `make` (and `sage --upgrade`) as well as the manual installation of individual spkgs with e.g. `sage -i` or `sage -f`.
- `SAGE_SPKG_INSTALL_DOCS` - if set to `yes`, then install package-specific documentation to `$SAGE_ROOT/local/share/doc/PACKAGE_NAME/` when an spkg is installed. This option may not be supported by all spkgs. Some spkgs might also assume that certain programs are available on the system (for example, `latex` or `pdflatex`).
- `SAGE_DOC_MATHJAX` - by default, any LaTeX code in Sage's documentation is processed by MathJax. If this variable is set to `no`, then MathJax is not used – instead, math is processed using LaTeX and converted by dvipng to image files, and then those files are included into the documentation. Typically, building the documentation using LaTeX and dvipng takes longer and uses more memory and disk space than using MathJax.
- `SAGE_BUILD_DIR` - the default behavior is to build each spkg in a subdirectory of `$SAGE_ROOT/local/var/tmp/sage/build/`; for example, build version 3.8.3.p12 of `atlas` in the directory `$SAGE_ROOT/local/var/tmp/sage/build/atlas-3.8.3.p12/`. If this variable is set, then build in `$SAGE_BUILD_DIR/atlas-3.8.3.p12/` instead. If the directory `$SAGE_BUILD_DIR` does not exist, it is created. As of this writing (Sage 4.8), when building the standard Sage packages, 1.5 gigabytes of free space are required in this directory (or more if `SAGE_KEEP_BUILT_SPKGS=yes` – see below); the exact amount of required space varies from platform to platform. For example, the block size of the file system will affect the amount of space used, since some spkgs contain many small files.

Warning: The variable `SAGE_BUILD_DIR` must be set to the full path name of either an existing directory for which the user has write permissions, or to the full path name of a nonexistent directory which the user has permission to create. The path name must contain **no spaces**.

- `SAGE_KEEP_BUILT_SPKGS` - the default behavior is to delete each build directory – the appropriate subdirectory of `$SAGE_ROOT/local/var/tmp/sage/build` or `$SAGE_BUILD_DIR` – after each spkg is successfully built, and to keep it if there were errors installing the spkg. Set this variable to `yes` to keep the subdirectory regardless. Furthermore, if you install an spkg for which there is already a corresponding subdirectory, for example left over from a previous build, then the default behavior is to delete that old subdirectory. If this variable is set to `yes`, then the old subdirectory is moved to `$SAGE_ROOT/local/var/tmp/sage/build/old/` (or `$SAGE_BUILD_DIR/old`), overwriting any already existing file or directory with the same name.

  Note: After a full build of Sage (as of version 4.8), these subdirectories can take up to 6 gigabytes of storage, in total, depending on the platform and the block size of the file system. If you always set this variable to `yes`, it can take even more space: rebuilding every spkg would use double the amount of space, and any upgrades to spkgs would create still more directories, using still more space.

  Note: In an existing Sage installation, running `sage -i -s <package-name>` or `sage -f -s <package-name>` installs the spkg `<package-name>` and keeps the corresponding build directory; thus setting `SAGE_KEEP_BUILT_SPKGS` to `yes` mimics this behavior when building Sage from scratch or when installing individual spkgs. So you can set this variable to `yes` instead of using the `-s` flag for `sage -i` and `sage -f`.

- `SAGE_FAT_BINARY` - to prepare a binary distribution that will run on the widest range of target machines, set this variable to `yes` before building Sage:

  ```
  export SAGE_FAT_BINARY="yes"
  make
  ./sage --bdist x.y.z-fat
  ```

Variables to set if you're trying to build Sage with an unusual setup, e.g., an unsupported machine or an unusual compiler:

- `SAGE_PORT` - if you try to build Sage on a platform which is recognized as being unsupported (e.g. AIX, or HP-UX), or with a compiler which is unsupported (anything except GCC), you will see a message saying something like:

  ```
  You are attempting to build Sage on IBM's AIX operating system,
  which is not a supported platform for Sage yet. Things may or
  may not work. If you would like to help port Sage to AIX,
  please join the sage-devel discussion list -- see
  http://groups.google.com/group/sage-devel
  The Sage community would also appreciate any patches you submit.

  To get past this message and try building Sage anyway,
  export the variable SAGE_PORT to something non-empty.
  ```

  If this is case and you want to try to build Sage anyway, follow the directions: set `SAGE_PORT` to something non-empty (and expect to run into problems).

- `SAGE_USE_OLD_GCC` - the Sage build process requires GCC with a version number of at least 4.0.1. If the most recent version of GCC on your system is the older 3.4.x series and you want to build with `SAGE_INSTALL_GCC=no`, then set `SAGE_USE_OLD_GCC` to something non-empty. Expect the build to fail in this case.

Environment variables dealing with specific Sage packages:

- `SAGE_ATLAS_ARCH` - if you are compiling ATLAS (in particular, if `SAGE_ATLAS_LIB` is not set), you can use this environment variable to set a particular architecture and instruction set extension, to control the maximum number of threads ATLAS can use, and to trigger the installation of a static library (which is disabled by default unless building our custom shared libraries fails). The syntax is

  `SAGE_ATLAS_ARCH=[threads:n,][static,]arch[,isaext1][,isaext2]...[,isaextN]`.

  While ATLAS comes with precomputed timings for a variety of CPUs, it only uses them if it finds an exact match. Otherwise, ATLAS runs through a lengthy automated tuning process in order to optimize performance for your particular system, which can take several days on slow and unusual systems. You drastically reduce the total Sage compile time if you manually select a suitable architecture. It is recommended to specify a suitable architecture on laptops or other systems with CPU throttling or if you want to distribute the binaries. Available architectures are

  `POWER3`, `POWER4`, `POWER5`, `PPCG4`, `PPCG5`, `POWER6`, `POWER7`, `IBMz9`, `IBMz10`, `IBMz196`, `x86x87`, `x86SSE1`, `x86SSE2`, `x86SSE3`, `P5`, `P5MMX`, `PPRO`, `PII`, `PIII`, `PM`, `CoreSolo`, `CoreDuo`, `Core2Solo`, `Core2`, `Corei1`, `Corei2`, `Atom`, `P4`, `P4E`, `Efficeon`, `K7`, `HAMMER`, `AMD64K10h`, `AMDDOZER`, `UNKNOWNx86`, `IA64Itan`, `IA64Itan2`, `USI`, `USII`, `USIII`, `USIV`, `UST2`, `UnknownUS`, `MIPSR1xK`, `MIPSICE9`, `ARMv7`.

  and instruction set extensions are

  `VSX`, `AltiVec`, `AVXMAC`, `AVXFMA4`, `AVX`, `SSE3`, `SSE2`, `SSE1`, `3DNow`, `NEON`.

  In addition, you can also set

  - `SAGE_ATLAS_ARCH=fast` which picks defaults for a modern (2-3 year old) CPU of your processor line, and
  - `SAGE_ATLAS_ARCH=base` which picks defaults that should work for a ~10 year old CPU.

  For example,

  `SAGE_ATLAS_ARCH=Corei2,AVX,SSE3,SSE2,SSE1`

  would be appropriate for a Core i7 CPU.

- `SAGE_ATLAS_LIB` - if you have an installation of ATLAS on your system and you want Sage to use it instead of building and installing its own version of ATLAS, set this variable to be the directory containing your ATLAS installation. It should contain the files `libatlas`, `liblapack`, `libcblas`, `libf77blas` (and optionally `libptcblas` and `libptf77blas` for multi-threaded computations), with extensions `.a`, `.so`, or `.dylib`. For backward compatibility, the libraries may also be in the subdirectory `SAGE_ATLAS_LIB/lib/`.

- `SAGE_MATPLOTLIB_GUI` - if set to anything non-empty except `no`, then Sage will attempt to build the graphical backend when it builds the matplotlib package.

- `INCLUDE_MPFR_PATCH` - this is used to add a patch to MPFR to bypass a bug in the memset function affecting sun4v machines with versions of Solaris earlier than Solaris 10 update 8 (10/09). Earlier versions of Solaris 10 can be patched by applying Sun patch 142542-01. Recognized values are:

  - `INCLUDE_MPFR_PATCH=0` - never include the patch - useful if you know all sun4v machines Sage will be used are running Solaris 10 update 8 or later, or have been patched with Sun patch 142542-01.
  - `INCLUDE_MPFR_PATCH=1` - always include the patch, so the binary will work on a sun4v machine, even if created on an older sun4u machine.
  - If this variable is unset, include the patch on sun4v machines only.

Some standard environment variables which are used by Sage:

- `CC` - while some programs allow you to use this to specify your C compiler, **not every Sage package recognizes this**. If GCC is installed within Sage, `CC` is ignored and Sage's `gcc` is used instead.

- `CPP` - similarly, this will set the C preprocessor for some Sage packages, and similarly, using it is likely quite risky. If GCC is installed within Sage, `CPP` is ignored and Sage's `cpp` is used instead.
- `CXX` - similarly, this will set the C++ compiler for some Sage packages, and similarly, using it is likely quite risky. If GCC is installed within Sage, `CXX` is ignored and Sage's `g++` is used instead.
- `FC` - similarly, this will set the Fortran compiler. This is supported by all Sage packages which have Fortran code. However, for historical reasons, the value is hardcoded during the initial `make` and subsequent changes to `$FC` might be ignored (in which case, the original value will be used instead). If GCC is installed within Sage, `FC` is ignored and Sage's `gfortran` is used instead.
- `CFLAGS`, `CXXFLAGS` and `FCFLAGS` - the flags for the C compiler, the C++ compiler and the Fortran compiler, respectively. The same comments apply to these: setting them may cause problems, because they are not universally respected among the Sage packages.

Sage uses the following environment variables when it runs:

- `DOT_SAGE` - this is the directory, to which the user has read and write access, where Sage stores a number of files. The default location is `$HOME/.sage/`.
- `SAGE_STARTUP_FILE` - a file including commands to be executed every time Sage starts. The default value is `$DOT_SAGE/init.sage`.
- `SAGE_SERVER` - if you want to install a Sage package using `sage -i <package-name>`, Sage downloads the file from the web, using the address `http://www.sagemath.org/` by default, or the address given by `SAGE_SERVER` if it is set. If you wish to set up your own server, then note that Sage will search the directory
  - `SAGE_SERVER/packages/upstream`

  for clean upstream tarballs, and it searches the directories
  - `SAGE_SERVER/packages/standard/`,
  - `SAGE_SERVER/packages/optional/`,
  - `SAGE_SERVER/packages/experimental/`,
  - and `SAGE_SERVER/packages/archive/`

  for old-style Sage packages. See the script `$SAGE_ROOT/src/bin/sage-spkg` for the implementation.
- `SAGE_PATH` - a colon-separated list of directories which Sage searches when trying to locate Python libraries.
- `SAGE_BROWSER` - on most platforms, Sage will detect the command to run a web browser, but if this doesn't seem to work on your machine, set this variable to the appropriate command.
- `SAGE_ORIG_LD_LIBRARY_PATH_SET` - set this to something non-empty to force Sage to set the `LD_LIBRARY_PATH` variable before executing system commands.
- `SAGE_ORIG_DYLD_LIBRARY_PATH_SET` - similar, but only used on OS X to set the `DYLD_LIBRARY_PATH` variable.
- `SAGE_CBLAS` - used in the file `SAGE_ROOT/src/sage/misc/cython.py`. Set this to the base name of the BLAS library file on your system if you want to override the default setting. That is, if the relevant file is called `libcblas_new.so` or `libcblas_new.dylib`, then set this to `cblas_new`.

Sage overrides the user's settings of the following variables:

- `MPLCONFIGDIR` - ordinarily, this variable lets the user set their matplotlib config directory. Due to incompatibilies in the contents of this directory among different versions of matplotlib, Sage overrides the user's setting, defining it instead to be `$DOT_SAGE/matplotlib-VER`, with `VER` replaced by the current matplotlib version number.

Variables dealing with doctesting:

- `SAGE_TIMEOUT` - used for Sage's doctesting: the number of seconds to allow a doctest before timing it out. If this isn't set, the default is 360 seconds (6 minutes).
- `SAGE_TIMEOUT_LONG` - used for Sage's doctesting: the number of seconds to allow a doctest before timing it out, if tests are run using `sage -t --long`. If this isn't set, the default is 1800 seconds (30 minutes).
- `SAGE_PICKLE_JAR` - if you want to update the the standard pickle jar, set this to something non-empty and run the doctest suite. See the documentation for the functions `picklejar()` and `unpickle_all()` in `$SAGE_ROOT/src/sage/structure/sage_object.pyx`, online here (picklejar) and here (unpickle_all).

4.7 Installation in a Multiuser Environment

This section addresses the question of how a system administrator can install a single copy of Sage in a multi-user computer network.

4.7.1 System-wide install

1. After building Sage, you may optionally copy or move the entire build tree to `/usr/local` or another location. If you do this, then you must run `./sage` once so that various hardcoded locations get updated. For this reason, it might be easier to simply build Sage in its final location.
2. Make a symbolic link to the `sage` script in `/usr/local/bin`:

   ```
   ln -s /path/to/sage-x.y.z/sage /usr/local/bin/sage
   ```

   Alternatively, copy the Sage script:

   ```
   cp /path/to/sage-x.y.z/sage /usr/local/bin/sage
   ```

   If you do this, make sure you edit the line:

   ```
   #SAGE_ROOT=/path/to/sage-version
   ```

   at the beginning of the copied `sage` script according to the direction given there to something like:

   ```
   SAGE_ROOT=<SAGE_ROOT>
   ```

   (note that you have to change `<SAGE_ROOT>` above!). It is recommended not to edit the original `sage` script, only the copy at `/usr/local/bin/sage`.
3. Make sure that all files in the Sage tree are readable by all (note that you have to change `<SAGE_ROOT>` below!):

   ```
   chmod a+rX -R <SAGE_ROOT>
   ```

4. Optionally, you can test Sage by running:

   ```
   make testlong
   ```

   or `make ptestlong` which tests files in parallel using multiple processes. You can also omit `long` to skip tests which take a long time.

4.7.2 Sagetex

To make SageTeX available to your users, see the instructions for *installation in a multiuser environment* .

4.8 Some common problems

4.8.1 ATLAS

Usually Sage will build ATLAS with architectural defaults that are not tuned to your particular CPU. In particular, if your CPU has powersaving enabled then no accurate timings can be made to tune the ATLAS build for your hardware. If BLAS performance is critical for you, you must recompile ATLAS after installing Sage either with architecture settings that match your hardware, or run through ATLAS' automatic tuning process where timings of different implementations are compared and the best choice used to build a custom ATLAS library. To do so, you have to

- Leave the computer idle while you are reinstalling ATLAS. Most of ATLAS will intentionally only compile/run on a single core. Accurate timings of cache edges require that the CPU is otherwise idle.
- Make sure that CPU powersaving mode (that is, anything but the `performance` CPU scaling governor in Linux) is turned off when building ATLAS. This requires administrator privileges.
- If your architecture is listed in `SAGE_ATLAS_ARCH`, you should set it as it can help ATLAS in narrowing down the timing search.

To help you disable CPU power saving, Sage includes an `atlas-config` script that will turn off CPU powersave and rebuild ATLAS. The script will call `sudo` to gain the necessary rights, which may prompt you for your password. For example:

```
atlas-config
```

will run through the full automated tuning, and:

```
SAGE_ATLAS_ARCH=Corei2,AVX,SSE3,SSE2,SSE1 atlas-config
```

would be appropriate if you have a Core i3/5/7 processor with AVX support.

This page was last updated in May 2014 (Sage 6.2).

CHAPTER

FIVE

MAKE SAGETEX KNOWN TO TEX

Sage is largely self-contained, but some parts do need some intervention to work properly. SageTeX is one such part.

The SageTeX package allows one to embed computations and plots from Sage into a LaTeX document. SageTeX is installed in Sage by default, but to use SageTeX with your LaTeX documents, you need to make your TeX installation aware of it before it will work.

The key to this is that TeX needs to be able to find `sagetex.sty`, which can be found in `SAGE_ROOT/local/share/texmf/tex/generic/sagetex/`, where `SAGE_ROOT` is the directory where you built or installed Sage. If TeX can find `sagetex.sty`, then SageTeX will work. There are several ways to accomplish this.

- The first and simplest way is simply to copy `sagetex.sty` into the same directory as your LaTeX document. Since the current directory is always searched when typesetting a document, this will always work.

  There are a couple small problems with this, however: the first is that you will end up with many unnecessary copies of `sagetex.sty` scattered around your computer. The second and more serious problem is that if you upgrade Sage and get a new version of SageTeX, the Python code and LaTeX code for SageTeX may no longer match, causing errors.

- The second way is to use the `TEXINPUTS` environment variable. If you are using the bash shell, you can do

  ```
  export TEXINPUTS="SAGE_ROOT/local/share/texmf//:"
  ```

  where `SAGE_ROOT` is the location of your Sage installation. Note that the double slash and colon at the end of that line are important. Thereafter, TeX and friends will find the SageTeX style file. If you want to make this change permanent, you can add the above line to your `.bashrc` file. If you are using a different shell, you may have to modify the above command to make the environment variable known; see your shell's documentation for how to do that.

  One flaw with this method is that if you use applications like TeXShop, Kile, or Emacs/AucTeX, they will not necessarily pick up the environment variable, since when they run LaTeX, they may do so outside your usual shell environment.

  If you ever move your Sage installation, or install a new version into a new directory, you'll need to update the above command to reflect the new value of `SAGE_ROOT`.

- The third (and best) way to make TeX aware of `sagetex.sty` is to copy that file into a convenient place in your home directory. In most TeX distributions, the `texmf` directory in your home directory is automatically searched for packages. To find out exactly what this directory is, do the following on the command line:

  ```
  kpsewhich -var-value=TEXMFHOME
  ```

  which will print out a directory, such as `/home/drake/texmf` or `/Users/drake/Library/texmf`. Copy the `tex/` directory from `SAGE_ROOT/local/share/texmf/` into your home `texmf` directory with a command like

```
cp -R SAGE_ROOT/local/share/texmf/tex TEXMFHOME
```

where `SAGE_ROOT` is, as usual, replaced with the location of your Sage installation and `TEXMFHOME` is the result of the `kpsewhich` command above.

If you upgrade Sage and discover that SageTeX no longer works, you can simply repeat these steps and the Sage and TeX parts of SageTeX will again be synchronized.

- For installation on a multiuser system, you just modify the above instructions appropriately to copy `sagetex.sty` into a systemwide TeX directory. Instead of the directory `TEXMFHOME`, probably the best choice is to use the result of

  ```
  kpsewhich -var-value=TEXMFLOCAL
  ```

  which will likely produce something like `/usr/local/share/texmf`. Copy the `tex` directory as above into the `TEXMFLOCAL` directory. Now you need to update TeX's database of packages, which you can do simply by running

  ```
  texhash TEXMFLOCAL
  ```

  as root, replacing `TEXMFLOCAL` appropriately. Now all users of your system will have access to the LaTeX package, and if they can also run Sage, they will be able to use SageTeX.

Warning: it's very important that the file `sagetex.sty` that LaTeX uses when typesetting your document match the version of SageTeX that Sage is using. If you upgrade your Sage installation, you really should delete all the old versions of `sagetex.sty` floating around.
Because of this problem, we recommend copying the SageTeX files into your home directory's texmf directory (the third method above). Then there is only one thing you need to do (copy a directory) when you upgrade Sage to insure that SageTeX will work properly.

5.1 SageTeX documentation

While not strictly part of installation, it bears mentioning here that the documentation for SageTeX is maintained in `SAGE_ROOT/local/share/doc/sagetex/sagetex.pdf`. There is also an example file in the same directory – see `example.tex` and `example.pdf`, the pre-built result of typesetting that file with LaTeX and Sage. You can also get those files from the SageTeX bitbucket page.

5.2 SageTeX and TeXLive

One potentially confusing issue is that the popular TeX distribution TeXLive 2009 includes SageTeX. This may seem nice, but with SageTeX, it's important that the Sage bits and LaTeX bits be synchronized – which is a problem in this case, since both Sage and SageTeX are updated frequently, and TeXLive is not. While at the time of this writing (March 2013), many Linux distributions have moved on to more recent releases of TeXLive, the 2009 release lingers and is, in fact, the source of most bug reports about SageTeX!

Because of this, it is *strongly recommended* that you always install the LaTeX part of SageTeX from Sage, as described above. The instructions above will insure that both halves of SageTeX are compatible and will work properly. Using TeXLive to provide the LaTeX side of SageTeX is not supported.

CHAPTER

SIX

DESKTOP ICON

These instructions will help you make a KDE desktop icon which starts the Sage notebook. Instructions for a Gnome desktop icon are probably similar.

1. Create a `notebook.sage` file containing only the line

   ```
   notebook(openviewer=True)
   ```

2. In your Desktop subdirectory, create a file `Sage-notebook.desktop` containing the lines

   ```
   [Desktop Entry]
   Comment=
   Comment[de]=
   Encoding=UTF-8
   Exec=/usr/local/bin/sage /home/martin/notebook.sage
   GenericName=
   GenericName[de]=
   Icon=
   MimeType=
   Name=Sage
   Name[de]=Sage
   Path=$HOME
   StartupNotify=true
   Terminal=false
   TerminalOptions=
   Type=Application
   X-DCOP-ServiceType=
   X-KDE-SubstituteUID=false
   X-KDE-Username=
   ```

   You will have to edit the `Exec=` line to point to your `sage` script and your `notebook.sage` file.

3. Right click on the Sage notebook desktop icon and click on `Properties`, then `Application`, then `Advanced Options`, then `Run in Terminal`. If you want to title the xwindow terminal, add in the terminal option box `-T "sage notebook"`.

To quit the Sage notebook, first enter `Ctrl-c` in the xwindow terminal running Sage, then enter `Ctrl-d` to quit Sage in the terminal, and finally close the browser (or browser tab) which was displaying the Sage notebook server.

For a picture for your icon, check out the Sage art at http://wiki.sagemath.org/art.

CHAPTER

SEVEN

THE DOCUMENTATION

You do not need to install the documentation separately: it is included with Sage. The Sage standard documentation includes a guided tour of Sage, a tutorial, a reference manual, a developer's guide, an installation guide, and other documents. The tutorial is a good starting point to learn how to use Sage. The reference manual describes what is available in Sage along with examples on how to use specific commands.

When you build Sage from source, by default the HTML version of the standard documentation is built as well. But note that in this way the HTML version does not have links to the PDF version. To build the HTML or PDF version of the documentation yourself, use the general command

```
sage -docbuild {document} {format}
```

For example, the command

```
sage -docbuild reference html
```

builds the HTML version of the reference manual.

You can choose to have the built HTML version of the documentation link to the PDF version. To do so, you need to build both the HTML and PDF versions. To have the HTML version link to the PDF version, do

```
sage -docbuild all html
sage -docbuild all pdf
```

Type `sage -docbuild -H` to see a list of available options for building the documentation or any part of it. See the file `SAGE_ROOT/Makefile` for further information on how the documentation is built by default.

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX