
Installation Guide

Release 9.6

The Sage Development Team

Jun 30, 2024

CONTENTS

1	macOS	3
2	Windows	5
3	Linux	7
4	In the cloud	9
4.1	Linux package managers	9
4.2	Install from Pre-built Binaries	9
4.3	Install from conda-forge	10
4.4	Install from Source Code	13
4.5	Launching SageMath	30
4.6	Troubleshooting	33
	Index	35

If you are reading this manual at <https://doc.sagemath.org/>, note that it was built at the time the most recent stable release of SageMath was made.

More up-to-date information and details regarding supported platforms may have become available afterwards and can be found in the section “Availability and installation help” of the [release tour](#) for each SageMath release.

Where would you like to run SageMath? Pick one of the following sections.

- **Do you want to do SageMath development?**

- **Yes, development:**

- Obtain the SageMath sources via `git` as described in [The Sage Developer's Guide](#).

- * Then build SageMath from source as described in section [Install from Source Code](#).
 - * Alternatively, follow the instructions in section [Using conda to provide all dependencies for the Sage library \(experimental\)](#); these describe an experimental method that gets all required packages, including Python packages, from conda-forge.

- **No development:**

- * Install the [binary build of SageMath](#) from the 3-manifolds project. It is a signed and notarized app, which works for macOS 10.12 and newer. It is completely self-contained and provides the standard Sage distribution together with many optional packages. Additional optional Python packages can be installed with the `%pip` magic command and will go into your `~/ .sage` directory.
 - * Alternatively, install SageMath from the [conda-forge](#) project, as described in section [Install from conda-forge](#).
 - * Alternatively, build SageMath from source as described in section [Install from Source Code](#).

WINDOWS

- **Do you want to do SageMath development?**

- **Yes, development:**

- Enable Windows Subsystem for Linux (WSL) by following the [official WSL setup guide](#). Be sure to do the steps to install WSL2 and set it as default. Then go to the Microsoft Store and install Ubuntu (or another Linux distribution). Start Ubuntu from the start menu.

- Then follow the instructions for development on Linux below.

- **No development:**

- * Enable Windows Subsystem for Linux (WSL) by following the [official WSL setup guide](#). Be sure to do the steps to install WSL2 and set it as default. Then go to the Microsoft Store and install Ubuntu (or another Linux distribution). Start Ubuntu from the start menu.

- On the Linux running on WSL, you always have root access, so you can use any of the installation methods described below for Linux.

- * Alternatively, in particular if you cannot use WSL, install [Cygwin](#) and then build SageMath from source as described in section *Install from Source Code*.

- **Do you want to do SageMath development?**

- **Yes, development:**

- Obtain the SageMath sources via `git` as described in [The Sage Developer's Guide](#).

- * Then build SageMath from source as described in section [Install from Source Code](#).

- * Alternatively, follow the instructions in section [Using conda to provide all dependencies for the Sage library \(experimental\)](#); these describe an experimental method that gets all required packages, including Python packages, from conda-forge.

- No development: **Do you have root access (sudo)?**

- * **Yes, root access:** Then the easiest way to install SageMath is through a Linux distribution that provides it as a package. Most major Linux distributions have up-to-date versions of SageMath, see [repology.org: sagemath](http://repology.org:sagemath) for an overview. See [Linux package managers](#) for additional information.

- If you are on an older version of your distribution and a recent version of SageMath is only available on a newer version of the distribution, consider upgrading your distribution. In particular, do not install a version of Sage older than 9.2.

- * **No root access, or on an older distribution** Install SageMath from the [conda-forge](#) project, as described in section [Install from conda-forge](#).

- * Alternatively, build SageMath from source as described in section [Install from Source Code](#).

IN THE CLOUD

- **CoCalc**: an online service that provides SageMath and many other tools.
- On any system that allows you to bring your own Docker images to run in a container: Use the [Docker image sagemath/sagemath](#).
- **Sage Cell Server**: an online service for elementary SageMath computations.

More information:

4.1 Linux package managers

On Debian GNU/Linux version ≥ 11 , Ubuntu version ≥ 21.04 , and Arch Linux there are three packages to install

- `sagemath` (for the binaries)
- `sagemath-jupyter` (for the browser interface)
- and the documentation which is called `sagemath-doc-en` on Debian/Ubuntu and `sagemath-doc` on Arch Linux.

See repology.org: [sagemath](#) for information about SageMath packages in other distributions, and their available versions. Do not install a version of Sage older than 9.2. If you are on an older version of your distribution and a recent version of SageMath is only available on a newer version of the distribution, consider upgrading your distribution.

Gentoo users might want to give a try to [sage-on-gentoo](#).

The [Trac wiki page Distribution](#) collects information regarding packaging and distribution of SageMath.

4.2 Install from Pre-built Binaries

4.2.1 Linux

SageMath used to provide pre-built binaries for several Linux flavors. This has been discontinued, as most major Linux distributions have up-to-date distribution packages providing SageMath. See [Linux package managers](#) for information.

4.2.2 macOS

macOS binaries are available from the [3-manifolds project](#). These have been signed and notarized, eliminating various errors caused by Apple’s gatekeeper antimalware protections.

SageMath used to provide pre-built binaries for macOS on its mirrors. This has been discontinued, and the old binaries that are still available there are no longer supported.

4.2.3 Microsoft Windows (Cygwin)

SageMath on Windows requires a 64-bit Windows (which is likely to be the case on a modern computer). If you happen to have a 32-bit Windows, you can consider the alternatives mentioned at the end of *Welcome to the Sage Installation Guide!*.

To install SageMath on Windows, just download the installer (see the above “Download Guide” section) and run it.

4.3 Install from conda-forge

SageMath can be installed on Linux and macOS via Conda from the [conda-forge](#) conda channel.

Both the x86_64 (Intel) architecture and the arm64/aarch64 architectures (including Apple Silicon, M1) are supported.

You will need a working Conda installation: either Mambaforge/Miniforge, Miniconda or Anaconda. If you don’t have one yet, we recommend installing [Mambaforge](#) as follows. In a terminal,

```
$ curl -L -O https://github.com/conda-forge/miniforge/releases/latest/download/
↪Mambaforge-$(uname)-$(uname -m).sh
$ sh Mambaforge-$(uname)-$(uname -m).sh
```

- Mambaforge and Miniforge use conda-forge as the default channel.
- If you are using Miniconda or Anaconda, set it up to use conda-forge:
 - Add the conda-forge channel: `conda config --add channels conda-forge`
 - Change channel priority to strict: `conda config --set channel_priority strict`

Optionally, use [mamba](#), which uses a faster dependency solver than conda. If you installed Mambaforge, it is already provided. Otherwise, use

```
$ conda install mamba
```

4.3.1 Installing all of SageMath from conda (not for development)

Create a new conda environment containing SageMath, either with mamba or conda:

```
$ mamba create -n sage sage python=X           # either
$ conda create -n sage sage python=X          # or
```

where X is version of Python, e.g. 3.9.

To use Sage from there,

- Enter the new environment: `conda activate sage`

- Start SageMath: `sage`

4.3.2 Using conda to provide system packages for the Sage distribution

If Conda is installed (check by typing `conda info`), there are two ways to prepare for installing SageMath from source:

- If you are using a git checkout:

```
$ ./bootstrap
```

- Create a new empty environment and activate:

```
$ conda create -n sage-build
$ conda activate sage-build
```

- Install standard packages recognized by sage's `spkg-configure` mechanism:

```
$ conda env update --file environment.yml -n sage-build
```

- Or install all standard and optional packages recognized by sage:

```
$ conda env update --file environment-optional.yml -n sage-build
```

- Then the SageMath distribution will be built using the compilers provided by Conda and using many packages installed by Conda:

```
$ ./bootstrap
$ ./configure --prefix=$CONDA_PREFIX
$ make
```

4.3.3 Using conda to provide all dependencies for the Sage library (experimental)

You can build and install the Sage library from source, using conda to provide all of its dependencies. This bypasses most of the build system of the Sage distribution and is the fastest way to set up an environment for Sage development.

Note that this is still an experimental feature and may not work as intended.

Here we assume that you are using a git checkout.

- Optionally, set the build parallelism for the Sage library. Use whatever the meaningful value for your machine is - no more than the number of cores:

```
$ export SAGE_NUM_THREADS=24
```

- As a recommended step, install the `mamba` package manager. If you skip this step, replace `mamba` by `conda` in the following steps:

```
$ conda install mamba
```

- Create and activate a new conda environment that provides the bootstrapping prerequisites. You can replace 3.9 by another Python version:

```
$ mamba create -n sage-build python=3.9 \
    gettext autoconf automake libtool pkg-config
$ conda activate sage-build
```

- Run bootstrap; this generates the files `src/environment*.yml` used in the next step:

```
$ ./bootstrap
```

- Populate the conda environment with the dependencies of Sage:

```
$ mamba env update -n sage-build -f src/environment.yml # alternatively, use
```

Alternatively, you can use `src/environment-optional.yml`, which will install some additional packages.

- Activate the conda environment again:

```
$ conda activate sage-build
```

- Run the configure script:

```
$ ./configure --with-python=$CONDA_PREFIX/bin/python \
              --prefix=$CONDA_PREFIX \
              $(for pkg in $(./sage -package list :standard: \
                           --has-file spkg-configure.m4 \
                           --has-file distros/conda.txt); do \
                  echo --with-system-$pkg=force; \
              done)
```

- Install the build prerequisites of the Sage library:

```
$ pip install --no-build-isolation -v -v --editable pkgs/sage-conf pkgs/sage-setup
```

- Install the Sage library:

```
$ pip install --no-build-isolation -v -v --editable src
```

- Verify that Sage has been installed:

```
$ sage -c 'print(version())'
SageMath version 9.6.beta5, Release Date: 2022-03-12
```

Note that `make` is not used at all. All dependencies (including all Python packages) are provided by conda.

Thus, you will get a working version of Sage much faster. However, note that this will invalidate the use of any Sage-the-distribution commands such as `sage -i`. Do not use them.

By using `pip install --editable` in the above steps, the Sage library is installed in editable mode. This means that when you only edit Python files, there is no need to rebuild the library; it suffices to restart Sage.

After editing any Cython files, rebuild by repeating the command:

```
$ pip install --no-build-isolation -v -v --editable src
```

4.4 Install from Source Code

Table of contents

- *Install from Source Code*
 - *Prerequisites*
 - *Additional software*
 - *Step-by-step installation procedure*
 - *Make targets*
 - *Environment variables*
 - *Installation in a Multiuser Environment*

Some familiarity with the use of the Unix command line may be required to build Sage from the [source code](#).

Building Sage from the source code has the major advantage that your install will be optimized 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.

See the file [README.md](#) in `SAGE_ROOT` for information on supported platforms and step-by-step instructions.

The following sections provide some additional details. Most users will not need to read them.

4.4.1 Prerequisites

General requirements

This section details the technical prerequisites needed on all platforms. See also the *System-specific requirements* below.

Disk space and memory

Your computer comes with at least 6 GB of free disk space. 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).

Command-line tools

In addition to standard [POSIX](#) utilities and the `bash` shell, the following standard command-line development tools must be installed on your computer:

- A **C/C++ compiler**: Since SageMath builds its own GCC if needed, a wide variety of C/C++ compilers is supported. Many GCC versions work, from as old as version 4.8 (but we recommend at least 5.1) to the most recent release. Clang also works. 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.
- **python:** Python 3.4 or later, or Python 2.7. (This range of versions is a minimal requirement for internal purposes of the SageMath build system, which is referred to as `sage-bootstrap-python`.)

Other versions of these may work, but they are untested.

Fortran and compiler suites

Sage installation also needs a Fortran compiler. It is determined automatically whether Sage's GCC package, or just its part containing Fortran compiler `gfortran` needs to be installed. This can be overwritten by running `./configure` with option `--without-system-gcc`.

Officially we support `gfortran` from [GNU Compiler Collection \(GCC\)](#). If C and C++ compilers also come from there (i.e., `gcc` and `g++`), their versions should match. Alternatively, one may use C and C++ compilers from [Clang: a C language family frontend for LLVM](#), and thus matching versions of `clang`, `clang++`, along with a recent `gfortran`. (Flang (or other LLVM-based Fortran compilers) are not officially supported, however it is possible to build Sage using `flang`, with some extra efforts needed to set various flags; this is work in progress at the moment (May 2019)).

Therefore, if you plan on using your own GCC compilers, then make sure that their versions match.

To force using specific compilers, set environment variables `CC`, `CXX`, and `FC` (for C, C++, and Fortran compilers, respectively) to the desired values, and run `./configure`. For example, `./configure CC=clang CXX=clang++ FC=gfortran` will configure Sage to be built with Clang C/C++ compilers and Fortran compiler `gfortran`.

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 Fortran compiler. In any case, you always need at least a C/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 configure options `--without-system-gcc` and `--with-system-gcc`, see [Using alternative compilers](#).

There are some known problems with old assemblers, in particular when building the `ecm` and `fflas_ffpack` packages. You should ensure that your assembler understands all instructions for your processor. On Linux, this means you need a recent version of `binutils`; on macOS you need a recent version of Xcode.

Python for venv

By default, Sage will try to use system's `python3` to set up a virtual environment, a.k.a. `venv` rather than building a Python 3 installation from scratch. Use the `configure` option `--without-system-python3` in case you want Python 3 built from scratch.

Sage will accept versions 3.7.x to 3.10.x.

You can also use `--with-python=/path/to/python3_binary` to tell Sage to use `/path/to/python3_binary` to set up the `venv`. Note that setting up `venv` requires a number of Python modules to be available within the Python in question. Currently, for Sage 9.6, these modules are as follows: `sqlite3`, `ctypes`, `math`, `hashlib`, `crypt`, `socket`, `zlib`, `distutils.core`, `ssl` - they will be checked for by the `configure` script.

Other notes

After extracting the Sage source tarball, the subdirectory `upstream` contains the source distributions for everything on which Sage depends.

If cloned from a git repository, the upstream tarballs will be downloaded, verified, and cached as part of the Sage installation process. 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.

System-specific requirements

On macOS, there are various developer tools needed which may require some registration on Apple's developer site; see [macOS prerequisite installation](#).

On Redhat-derived systems not all perl components are installed by default and you might have to install the `perl-ExtUtils-MakeMaker` package.

On Cygwin, the `lapack` and `liblapack-devel` packages are required.

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`.

Linux recommended installation

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/standard components. The lists provided below are longer than the minimal prerequisites, which are basically `binutils`, `gcc/clang`, `make`, `tar`, but there is no real need to build compilers and other standard tools and libraries on a modern Linux system, in order to be able to build Sage. 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`.

On Debian ("buster" or newer) or Ubuntu ("bionic" or newer):

```
$ sudo apt-get install bc binutils bzip2 ca-certificates cliquer cmake curl ecl eplib-
↳ tools fflas-ffpack flintqs g++ gcc gengetopt gfan gfortran glpk-utils gmp-ecm lcalc
↳ libatomic-ops-dev libboost-dev libbraiding-dev libbrial-dev libbrial-groebner-dev
↳ libbz2-dev libcdd-dev libcdd-tools libcliquer-dev libcurl4-openssl-dev libec-dev
↳ libecm-dev libffi-dev libflint-arb-dev libflint-dev libfpdll-dev libfreetype6-dev
↳ libgc-dev libgd-dev libgf2x-dev libgiac-dev libgivaro-dev libglpk-dev libgmp-dev
↳ libgsl-dev libhomfly-dev libiml-dev liblfunction-dev liblinbox-dev librcalib-dev
↳ liblzma-dev libm4ri-dev libm4rie-dev libmpc-dev libmpfi-dev libmpfr-dev libncurses5-
↳ dev libntl-dev libopenblas-dev libpari-dev libpcre3-dev libplanarity-dev libppl-dev
↳ libprimesieve-dev libpython3-dev libqhull-dev libreadline-dev librwn-dev libsingular4-
↳ dev libsqlite3-dev libssl-dev libsuitesparse-dev libsymmetrca2-dev libz-dev libzmq3-
↳ dev libzn-poly-dev m4 make nauty openssl palp pari-doc pari-elldata pari-galdata pari-
↳ galpol pari-gp2c pari-seadata patch perl pkg-config planarity ppl-dev python3 python3-
```

On Fedora / Redhat / CentOS:

```
$ sudo yum install --setopt=tsflags=L-function L-function-devel R R-devel Singular
↳ Singular-devel arb arb-devel binutils boost-devel brial brial-devel bzip2 bzip2-devel
↳ cddlib cliquer cliquer-devel cmake curl diffutils ecl eclib eclib-devel fflas-ffpack-
↳ devel findutils flint flint-devel gc gc-devel gcc gcc-c++ gcc-gfortran gd gd-devel
↳ gengetopt gf2x gf2x-devel gfan giac giac-devel givaro givaro-devel glpk glpk-devel
↳ glpk-utils gmp gmp-devel gmp-ecm gmp-ecm-devel gsl gsl-devel iml iml-devel libatomic_
↳ ops libatomic_ops-devel libbraiding libcurl-devel libffi libffi-devel libfpdll
↳ libfpdll-devel libhomfly-devel libmpc libmpc-devel linbox lrcalc-devel m4 m4ri-devel
↳ m4rie-devel make mpfr-devel nauty ncurses-devel ntl-devel openblas-devel openssl
↳ openssl-devel palp pari-devel pari-elldata pari-galdata pari-galpol pari-gp pari-
↳ seadata patch pcre pcre-devel perl perl-ExtUtils-MakeMaker perl-IPC-Cmd pkg-config
↳ planarity planarity-devel ppl ppl-devel primecount primecount-devel primesieve
↳ primesieve-devel python3 python3-devel qhull qhull-devel readline-devel rw-devel
↳ sqlite sqlite-devel suitesparse suitesparse-devel symmetrica-devel sympow tachyon
↳ tachyon-devel tar tox which xz xz-devel zeromq zeromq-devel zlib-devel zn_poly zn_poly-
↳ devel
```

On Arch Linux:

```
$ sudo pacman -S arb bc binutils boost brial cblas cddlib cmake ecl eclib fflas-
↳ ffpack flintqs gc gcc gcc-fortran gd gf2x gfan giac glpk gsl iml lapack lcalc
↳ libatomic_ops libbraiding libgiac libhomfly linbox lrcalc m4 m4ri m4rie make nauty
↳ openblas openssl palp pari pari-elldata pari-galdata pari-galpol pari-seadata patch
↳ perl planarity ppl primecount primesieve python qhull r rankwidth readline singular
↳ sqlite3 suitesparse symmetrica sympow tachyon tar which zn_poly
```

In addition to these, if you don't want Sage to build optional packages that might be available from your OS, cf. the growing list of such packages on [trac ticket #27330](#), install on Debian ("buster" or newer) or Ubuntu ("bionic" or newer):

```
$ sudo apt-get install 4ti2 clang coinor-cbc coinor-libcbc-dev graphviz libfile-slurp-
↳ perl libgraphviz-dev libigraph-dev libisl-dev libjson-perl libmongodb-perl libnauty-
↳ dev libperl-dev libpolymake-dev libsvg-perl libterm-readkey-perl libterm-readline-gnu-
↳ perl libxml-libxslt-perl libxml-writer-perl libxml2-dev lrslib ninja-build pari-gp2c
↳ pdf2svg polymake texinfo
```

On Fedora / Redhat / CentOS:

```
$ sudo yum install 4ti2 clang coin-or-Cbc coin-or-Cbc-devel coxeter coxeter-devel
↳ coxeter-tools graphviz igraph igraph-devel isl-devel libnauty-devel libxml2-devel
↳ lrslib ninja-build pari-galpol pari-seadata pdf2svg perl-ExtUtils-Embed perl-File-
↳ Slurp perl-JSON perl-MongoDB perl-Term-ReadLine-Gnu perl-TermReadKey perl-XML-LibXML
↳ perl-XML-LibXSLT perl-XML-Writer polymake texinfo
```

On Arch Linux:

```
$ sudo pacman -S 4ti2 clang coin-or-cbc coxeter graphviz igraph libxml2 lrs ninja
↳ pari-elldata pari-galpol pari-seadata pdf2svg perl-term-readline-gnu polymake
```

On other Linux systems, you might use [rpm](#), [yum](#), or other package managers.

macOS prerequisite installation

On macOS systems, you need a recent version of [Command Line Tools](#). It provides all the above requirements.

Run the command `xcode-select --install` from a Terminal window and click “Install” in the pop-up dialog box.

If you have already installed [Xcode](#) (which at the time of writing is freely available in the Mac App Store, or through <https://developer.apple.com/downloads/> provided you registered for an Apple Developer account), you can install the command line tools from there as well.

If you have not installed [Xcode](#) you can get these tools as a relatively small download, but it does require a registration.

- First, you will need to register as an Apple Developer at <https://developer.apple.com/register/>.
- Having done so, you should be able to download it for free at <https://developer.apple.com/downloads/index.action?command%20line%20tools>
- Alternately, <https://developer.apple.com/opensource/> should have a link to Command Line Tools.

macOS recommended installation

Although Sage can in theory build its own version of `gfortran`, this can take a while, and the process fails on some recent versions of OS X. So instead you can install your own copy. One advantage of this is that you can install it once, and it will get used every time you build Sage, rather than building `gfortran` every time.

One way to do that is with the [Homebrew package manager](#). Install Homebrew as their web page describes, and then the command

```
$ brew install gcc
```

will install Homebrew’s `gcc` package, which includes `gfortran`. Sage will also use other Homebrew packages, if they are present. You can install the following:

```
$ brew install arb bdw-gc boost bzip2 cmake curl ecl flint fplll freetype gcc gd_
↪ gengetopt gfortran glpk gmp gpatch gsl libatomic_ops libffi libiconv libmpc libpng_
↪ mpfi mpfr nauty ncurses ntl openblas openssl pari pari-elldata pari-galdata pari-
↪ galpol pari-seadata pcre pkg-config ppl primesieve python3 qhull r readline singular_
↪ sqlite suite-sparse tox xz zeromq zlib
```

Some Homebrew packages are installed “keg-only,” meaning that they are not available in standard paths. To make them accessible when building Sage, run

```
$ source SAGE_ROOT/.homebrew-build-env
```

(replacing `SAGE_ROOT` by Sage’s home directory). You can add a command like this to your shell profile if you want the settings to persist between shell sessions.

Some additional optional packages are taken care of by:

```
$ brew install apaffenholz/polymake/polymake cbc graphviz igraph isl libxml2 llvm_
↪ nauty ninja pdf2svg texinfo
```

Cygwin prerequisite installation

Sage can be built only on the 64-bit version of Cygwin. See the file [README.md](#) in `SAGE_ROOT` for the most up-to-date instructions for building Sage on Cygwin.

Although it is possible to install Sage's dependencies using the Cygwin graphical installer, it is recommended to install the `apt-cyg` command-line package installer, which is used for the remainder of these instructions. To run `apt-cyg`, you must have already installed (using the graphical installer) the following packages at a minimum:

```
bzip2 coreutils gawk gzip tar wget
```

With the exception of `wget` most of these are included in the default package selection when you install Cygwin. Then, to install `apt-cyg` run:

```
$ curl -OL https://rawgit.com/transcode-open/apt-cyg/master/apt-cyg
$ install apt-cyg /usr/local/bin
$ rm -f apt-cyg
```

To install the current set of system packages known to work for building Sage, run:

```
$ apt-cyg install R binutils bzip2 cddlib-devel cddlib-tools cmake curl findutils gcc-
↳core gcc-fortran gcc-g++ gengetopt glpk libatomic_ops-devel libboost-devel libbz2-
↳devel libcrypt-devel libcurl-devel libffi-devel libflint-devel libfreetype-devel libgc-
↳devel libgd-devel libglpk-devel libgmp-devel libgsl-devel libiconv-devel liblapack-
↳devel liblzma-devel libmpc-devel libmpfr-devel libncurses-devel libntl-devel
↳libopenblas libpcre-devel libreadline-devel libsqlite3-devel libssl-devel
↳libsuitesparseconfig-devel libtirpc-devel libzmq-devel m4 make patch perl perl-
↳ExtUtils-MakeMaker python39 python39-devel python39-urllib3 qhull singular singular-
↳devel sqlite3 tar which xz zlib-devel
```

Optional packages that are also known to be installable via system packages include:

```
$ apt-cyg install clang graphviz info lib4ti2-devel lib4ti2_0 libisl-devel libxml2-
↳devel ninja perl-Term-ReadLine-Gnu
```

Ubuntu on Windows Subsystem for Linux (WSL) prerequisite installation

Sage can be installed onto Linux running on Windows Subsystem for Linux (WSL). These instructions describe a fresh install of Ubuntu 20.10, but other distributions or installation methods should work too, though have not been tested.

- Enable hardware-assisted virtualization in the EFI or BIOS of your system. Refer to your system (or motherboard) maker's documentation for instructions on how to do this.
- Set up WSL by following the [official WSL setup guide](#). Be sure to do the steps to install WSL2 and set it as default.
- Go to the Microsoft Store and install Ubuntu.
- Start Ubuntu from the start menu. Update all packages to the latest version.
- Reboot the all running WSL instances one of the following ways:
 - Open Windows Services and restart the `LxssManager` service.
 - Open the Command Prompt or Powershell and enter this command:

```
wsl --shutdown
```

- [Upgrade to the Ubuntu 20.10](#). This step will not be necessary once Ubuntu 20.10 is available in the Microsoft Store.

From this point on, follow the instructions in the [Linux recommended installation](#) section. It is strongly recommended to put the Sage source files in the Linux file system, for example, in the `/home/username/sage` directory, and not in the Windows file system (e.g. `/mnt/c/...`).

You may encounter permission errors of the kind `"[Errno 13] Permission denied: 'build/bdist.linux-x86_64/wheel/<package>.dist-info'"` during `make`. This usually comes from a permission conflict between the Windows and Linux file system. To fix it create a temporary build folder in the Linux file system using `mkdir -p ~/tmp/sage` and use it for building by `eval SAGE_BUILD_DIR="~/tmp/sage" make`. Also see the [related Github issue](#) for other workarounds.

When the installation is complete, you may be interested in [WSL Post-installation steps](#).

Other platforms

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

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

Notes on using conda

If you don't want conda to be used by sage, deactivate conda (for the current shell session).

- Type:

```
$ conda deactivate
```

- Repeat the command until `conda info` shows:

```
$ conda info

active environment : None
...
```

Then SageMath will be built either using the compilers provided by the operating system, or its own compilers.

Specific notes for make and tar

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

Using alternative compilers

Sage developers tend to use fairly recent versions of GCC. Nonetheless, the Sage build process on Linux should succeed with any reasonable C/C++ compiler; (we do not recommend GCC older than version 5.1). 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 pass Sage's `./configure` compiler names, via environment variables `CC`, `CXX`, and `FC`, for C, C++, and Fortran compilers, respectively, e.g. if you C compiler is `clang`, your C++ compiler is `clang++`, and your Fortran compiler is `flang` then you would need to run:

```
$ CC=clang CXX=clang++ FC=flang ./configure
```

before running `make`. It is recommended that you inspect the output of `./configure` in order to check that Sage will not try to build GCC. Namely, there should be lines like:

```
gcc-7.2.0 will not be installed (configure check)
...
gfortran-7.2.0 will not be installed (configure check)
```

indicating that Sage will not attempt to build `gcc/g++/gfortran`.

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

4.4.2 Additional software

Recommended programs

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

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

It is highly recommended that you have **LaTeX** installed, but it is not required. The most popular packaging is **TeX Live**, which can be installed following the directions on their web site. On Linux systems you can alternatively install your distribution's `texlive` packages:

```
$ sudo apt-get install texlive      # debian
$ sudo yum install texlive         # redhat
```

or similar commands. In addition to the base TeX Live install, you may need some optional TeX Live packages, for example country-specific `babel` packages for the localized Sage documentation.

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.

On Debian/Ubuntu, the following system packages are recommended.

- `texlive-generic-extra` (to generate pdf documentation)
- `texlive-xetex` (to convert Jupyter notebooks to pdf)

- `latexmk` (to generate pdf documentation)
- `pandoc` (to convert Jupyter notebooks to pdf)
- `dvipng` (to render text with LaTeX in Matplotlib)
- `default-jdk` (to run the Jmol 3D viewer from the console and generate images for 3D plots in the documentation)
- `ffmpeg` (to produce animations)
- `libavdevice-dev` (to produce animations)

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 any part of Sage relying on it:

```
$ sage -f python3 # rebuild Python3
$ 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.

4.4.3 Step-by-step installation procedure

General procedure

1. Follow the procedure in the file `README.md` in `SAGE_ROOT`.
2. Additional remarks: You do not need to be logged in as root, since no files are changed outside of the `sage-x.y` 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.

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 <https://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, macOS, 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 and *Environment variables* for additional information on useful environment variables used by Sage.

3. 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
+-----+
| SageMath version 8.8, Release Date: 2019-06-26           |
| Using Python 3.7.3. 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 <https://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
```

4. 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!

5. 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 "$@"
```

6. 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/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!):

```
#!/usr/bin/env 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 by right clicking the mouse on the icon).

- On Linux and macOS 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.

7. Optional: Install optional Sage packages and databases. Type `sage --optional` to see a list of them (this requires an Internet connection), or visit <https://www.sagemath.org/packages/optional/>. Then type `sage -i <package-name>` to automatically download and install a given package.
8. 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.
9. Have fun! Discover some amazing conjectures!

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).

4.4.4 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 doc-html-no-plot` builds Sage’s documentation in html format but skips the inclusion of graphics auto-generated using the `.. PLOT` markup and the `sphinx_plot` function. This is primarily intended for use when producing certain binary distributions of Sage, to lower the size of the distribution. As of this writing (December 2014, Sage 6.5), there are only a few such plots, adding about 4M to the `local/share/doc/sage/` directory. In the future, this may grow, of course. Note: after using this, if you want to build the documentation and include the pictures, you should run `make doc-uninstall`, because the presence, or lack, of pictures is cached in the documentation output. You can benefit from this no-plot feature with other make targets by doing `export SAGE_DOCBUILD_OPTS+= ' --no-plot '`
- `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 doc-uninstall` and `make doc-clean` each remove several directories which are produced when building the documentation.
- `make distclean` restores the Sage directory to its state before doing any building: it is almost equivalent to deleting Sage's entire home directory and unpacking the source tarfile again, the only difference being that the `.git` directory is preserved, so `git` branches are not deleted.

4.4.5 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.

The Sage source tarball already includes the sources for all standard packages, that is, it allows you to build Sage without internet connection. The `git` repository, however, does not contain the source code for third-party packages. Instead, it will be downloaded as needed (Note: you can run `make download` to force downloading packages before building). Package downloads use the Sage mirror network, the nearest mirror will be determined automatically for you. This is influenced by the following environment variable:

- `SAGE_SERVER` - Try the specified mirror first, before falling back to the official Sage mirror list. Note that Sage will search the directory
 - `SAGE_SERVER/spkg/upstream`
 for upstream tarballs.

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 some Sage packages may not 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 macOS 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).
- `V` - if set to `0`, silence the build. Instead of showing a detailed compilation log, only one line of output is shown at the beginning and at the end of the installation of each Sage package. To see even less output, use:

```
$ make -s V=0
```

(Note that the above uses the syntax of setting a Makefile variable.)

- `SAGE_CHECK` - if set to `yes`, then during the build process, or when installing packages manually, run the test suite for each package which has one, and stop with an error if tests are failing. If set to `warn`, then only a warning is printed in this case. 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 `ppl, !python3`, then always run the test suite for PPL, but always skip the test suite for Python 3.

Note: As of Sage 9.1, the test suites for the Python 2 and 3 spkgs fail on most platforms. So when this variable is empty or unset, Sage uses a default of `!python2, !python3`.

- `SAGE_INSTALL_GCC` - **Obsolete, do not use, to be removed**
- `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.

Instead of using `SAGE_DEBUG` one can configure with `--enable-debug={no|symbols|yes}`.

- `SAGE_PROFILE` - controls profiling support. If this is set to `yes`, profiling support is enabled where possible. Note that Python-level profiling is always available; This option enables profiling in Cython modules.
- `SAGE_SPKG_INSTALL_DOCS` - if set to `yes`, then install package-specific documentation to `$$SAGE_ROOT/local/share/doc/PACKAGE_NAME/` when an spkg is installed. This option may not be supported by all spkgs. Some spkgs might also assume that certain programs are available on the system (for example, `latex` or `pdflatex`).
- `SAGE_DOCBUILD_OPTS` - the value of this variable is passed as an argument to `sage --docbuild all html` or `sage --docbuild all pdf` when you run `make`, `make doc`, or `make doc-pdf`. For example, you can add `--no-plot` to this variable to avoid building the graphics coming from the `.. PLOT` directive within the documentation, or you can add `--include-tests-blocks` to include all "TESTS" blocks in the reference manual. Run `sage --docbuild help` to see the full list of options.
- `SAGE_BUILD_DIR` - the default behavior is to build each spkg in a subdirectory of `$$SAGE_ROOT/local/var/tmp/sage/build/`; for example, build version 7.27.0 of `ipython` in the directory `$$SAGE_ROOT/local/var/tmp/sage/build/ipython-7.27.0/`. If this variable is set, then build in `$$SAGE_BUILD_DIR/ipython-7.27.0/` 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 a `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 build binaries that will run on the widest range of target CPUs set this variable to `yes` before building Sage or configure with `--enable-fat-binary`. This does not make the binaries relocatable, it only avoids newer CPU instruction set extensions. For relocatable (=can be moved to a different directory) binaries, you must use <https://github.com/sagemath/binary-pkg>
- `SAGE_SUDO` - set this to `sudo -E` or to any other command prefix that is necessary to write into a installation hierarchy (`SAGE_LOCAL`) owned by root or another user. Note that this command needs to preserve environment variable settings (plain `sudo` does not).

Not all Sage packages currently support `SAGE_SUDO`.

Therefore this environment variable is most useful when a system administrator wishes to install an additional Sage package that supports `SAGE_SUDO`, into a root-owned installation hierarchy (`SAGE_LOCAL`).

Environment variables dealing with specific Sage packages:

- `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.
- `PARI_CONFIGURE` - use this to pass extra parameters to PARI's `Configure` script, for example to specify graphics support (which is disabled by default). See the file `build/pkgs/pari/spkg-install` for more information.
- `SAGE_TUNE_PARI`: If `yes`, enable PARI self-tuning. Note that this can be time-consuming. If you set this variable to "yes", you will also see this: **WARNING: Tuning PARI/GP is unreliable. You may find your build of PARI fails, or PARI/GP does not work properly once built. We recommend to build this package with `SAGE_CHECK="yes"`.**
- `PARI_MAKEFLAGS`: The value of this variable is passed as an argument to the `$MAKE` command when compiling PARI.

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. Note also that `export CFLAGS=""` does not have the same effect as `unset CFLAGS`. The latter is preferable.
- Similar comments apply to other compiler and linker flags like `CPPFLAGS`, `LDLFLAGS`, `CXXFLAG64`, `LDLFLAG64`, and `LD`.
- `OPENBLAS_CONFIGURE` - adds additional configuration flags for the OpenBLAS package that gets added to the make command. (see [trac ticket #23272](#))

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`.
- `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.

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 300 seconds (5 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_TEST_GLOBAL_ITER`, `SAGE_TEST_ITER`: these can be used instead of passing the flags `--global-iterations` and `--file-iterations`, respectively, to `sage -t`. Indeed, these variables are only used if the flags are unset. Run `sage -t -h` for more information on the effects of these flags (and therefore these variables).

Sage sets some other environment variables. The most accurate way to see what Sage does is to first run `env` from a shell prompt to see what environment variables you have set. Then run `sage --sh -c env` to see the list after Sage sets its variables. (This runs a separate shell, executes the shell command `env`, and then exits that shell, so after running this, your settings will be restored.) Alternatively, you can peruse the shell script `src/bin/sage-env`.

Sage also has some environment-like settings. Some of these correspond to actual environment variables while others have names like environment variables but are only available while Sage is running. To see a list, execute `sage .env`. [TAB] while running Sage.

4.4.6 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.

System-wide install

In the instructions below, we assume that `/path/to/sage-x.y` is the directory where you want to install Sage.

1. First of all, extract the Sage source tarball in `/path/to` (this will create the directory `/path/to/sage-x.y`). After extracting, you can change the directory name if you do not like `sage-x.y`.
2. Change the ownership of the `/path/to/sage-x.y` directory tree to your normal user account (as opposed to root). This is because Sage will refuse to compile as root.

```
$ chown -R user:group /path/to/sage-x.y
```

3. Using your normal user account, build Sage. See the *Step-by-step installation procedure* above.
4. Make a symbolic link to the `sage` script in `/usr/local/bin`:

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

Alternatively, copy the Sage script:

```
$ cp /path/to/sage-x.y/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`.

5. 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.

This page was last updated in April 2022 (Sage 9.6).

4.5 Launching SageMath

Now we assume that you installed SageMath properly on your system. This section quickly explains how to start the Sage console and the Jupyter Notebook from the command line.

If you did install the Windows version or the macOS application you should have icons available on your desktops or launching menus. Otherwise you are strongly advised to create shortcuts for Sage as indicated at the end of the “Linux” Section in *Install from Pre-built Binaries*. Assuming that you have this shortcut, running

```
sage
```

in a console starts a Sage session. To quit the session enter `quit` and then press `<Enter>`.

To start a Jupyter Notebook instead of a Sage console, run the command

```
sage -n jupyter
```

instead of just `sage`. To quit the Jupyter Notebook press `<Ctrl> + <C>` twice in the console where you launched the command.

4.5.1 Using a Jupyter Notebook remotely

If Sage is installed on a remote machine to which you have `ssh` access, you can launch a Jupyter Notebook using a command such as

```
ssh -L localhost:8888:localhost:8888 -t USER@REMOTE sage -n jupyter --no-browser --  
->port=8888
```

where `USER@REMOTE` needs to be replaced by the login details to the remote machine. This uses local port forwarding to connect your local machine to the remote one. The command will print a URL to the console which you can copy and paste in a web browser.

Note that this assumes that a firewall which might be present between server and client allows connections on port 8888. See details on port forwarding on the internet, e.g. <https://www.ssh.com/ssh/tunneling/example>.

4.5.2 WSL Post-installation steps

If you’ve installed Sage Math from source on WSL, there are a couple of extra steps you can do to make your life easier:

Create a notebook launch script

If you plan to use JupyterLab, install that first.

Now create a script called `~/sage_nb.sh` containing the following lines, and fill in the correct paths for your desired starting directory and `SAGE_ROOT`

```
#!/bin/bash  
# Switch to desired windows directory  
cd /mnt/c/path/to/desired/starting/directory  
# Start the Jupyter notebook  
SAGE_ROOT/sage --notebook
```

(continues on next page)

(continued from previous page)

```
# Alternatively you can run JupyterLab - delete the line above, and uncomment the line
↪ below
#SAGE_ROOT/sage --notebook jupyterlab
```

Make it executable:

```
chmod ug+x ~/sage_nb.sh
```

Run it to test:

```
cd ~
./sage_nb.sh
```

The Jupyter(Lab) server should start in the terminal window, and you windows browser should open a page showing the Jupyter or JupyterLab starting page, at the directory you specified.

Create a shortcut

This is a final nicety that lets you start the Jupyter or JupyterLab server in one click:

- Open Windows explorer, and type %APPDATA%\Microsoft\Windows\Start Menu\Programs in the address bar and press enter. This is the folder that contains you start menu shortcuts. If you want the sage shortcut somewhere else (like your desktop), open that folder instead.
- Open a separate window and go to %LOCALAPPDATA%\Microsoft\WindowsApps\
here from the context menu when you drop it.
- Right-click-drag the ubuntu.exe icon from the second window into the first, then choose Create shortcuts here from the context menu when you drop it.
- To customize this shortcut, right-click on it and choose properties.
 - On the General tab:
 - * Change the name to whatever you want, e.g. “Sage 9.2 JupyterLab”
 - On the Shortcut tab:
 - * Change Target to: ubuntu.exe run ~/sage_nb.sh
 - * Change Start in to: %USERPROFILE%
 - * Change Run to: Minimised
 - * Change the icon if you want

Now hit the start button or key and type the name you gave it. it should appear in the list, and should load the server and fire up your browser when you click on it.

For further reading you can have a look at the other documents in the SageMath documentation at <http://doc.sagemath.org/>.

4.5.3 Setting up SageMath as a Jupyter kernel in an existing Jupyter notebook or JupyterLab installation

You may already have a global installation of Jupyter. For added convenience, it is possible to link your installation of SageMath into your Jupyter installation, adding it to the list of available kernels that can be selected in the notebook or JupyterLab interface.

Assuming that SageMath can be invoked by typing `sage`, you can use

```
sage -sh -c 'ls -d $SAGE_VENV/share/jupyter/kernels/sagemath'
```

to find the location of the SageMath kernel description.

Assuming that the Jupyter notebook can be started by typing `jupyter notebook`, the following command will install SageMath as a new kernel.

```
jupyter kernelspec install --user $(sage -sh -c 'ls -d $SAGE_VENV/share/jupyter/kernels/
↪sagemath')
```

This installs the kernel under the name `sagemath`. If you wish to rename it to something more specific in order to distinguish between different installations of SageMath, you can use the additional option `--name`, for example

```
jupyter kernelspec install --user $(sage -sh -c 'ls -d $SAGE_VENV/share/jupyter/kernels/
↪sagemath') --name sagemath-dev-worktree
```

The `jupyter kernelspec` approach by default does lead to about 2Gb of SageMath documentation being copied into your personal jupyter configuration directory. You can avoid that by instead putting a symlink in the relevant spot. and

```
jupyter --paths
```

to find valid data directories for your Jupyter installation. A command along the lines of

```
ln -s $(sage -sh -c 'ls -d $SAGE_VENV/share/jupyter/kernels/sagemath') $HOME/.local/
↪share/jupyter
```

can then be used to create a symlink to the SageMath kernel description in a location where your own `jupyter` can find it.

If you have installed SageMath from source, the alternative command

```
ln -s $(sage -sh -c 'ls -d $SAGE_ROOT/venv/share/jupyter/kernels/sagemath') $HOME/.local/
↪share/jupyter
```

creates a symlink that will stay current even if you switch to a different Python version later.

To get the full functionality of the SageMath kernel in your global Jupyter installation, the following Notebook Extension packages also need to be installed (or linked) in the environment from which the Jupyter installation runs.

You can check the presence of some of these packages using the command `jupyter nbextension list`.

- For the Sage interacts, you will need the package `widgetsnbextension` installed in the Python environment of the Jupyter installation. If your Jupyter installation is coming from the system package manager, it is best to install `widgetsnbextension` in the same way. Otherwise, install it using `pip`.

To verify that interacts work correctly, you can evaluate the following code in the notebook:

```
@interact
def _(k=slider(vmin=-1.0, vmax= 3.0, step_size=0.1, default=0), auto_update=True):
    plot([lambda u:u^2-1, lambda u:u+k], (-2,2),
         ymin=-1, ymax=3, fill={1:[0]}, fillalpha=0.5).show()
```

- For 3D graphics using Three.js, by default, internet connectivity is needed, as SageMath's custom build of the Javascript package Three.js is retrieved from a content delivery network.

To verify that online 3D graphics with Three.js works correctly, you can evaluate the following code in the notebook:

```
plot3d(lambda u,v:(u^2+v^2)/4-2, (-2,2), (-2,2)).show()
```

However, it is possible to configure graphics with Three.js for offline use. In this case, the Three.js installation from the Sage distribution needs to be made available in the environment of the Jupyter installation. This can be done by copying or symlinking. The Three.js installation in the environment of the Jupyter installation must exactly match the version that comes from the Sage distribution. It is not supported to use several Jupyter kernels corresponding to different versions of the Sage distribution.

To verify that offline 3D graphics with Three.js works correctly, you can evaluate the following code in the notebook:

```
plot3d(lambda u,v:(u^2+v^2)/4-2, (-2,2), (-2,2), online=False).show()
```

- For 3D graphics using jsmol, you will need the package `jupyter-jsmol` installed in the Python environment of the Jupyter installation. You can install it using `pip`. (Alternatively, you can copy or symlink it.)

To verify that jsmol graphics work correctly, you can evaluate the following code in the notebook:

```
plot3d(lambda u,v:(u^2+v^2)/4-2, (-2,2), (-2,2)).show(viewer="jsmol")
```

4.6 Troubleshooting

If no binary version is available for your system, you can fallback to the *Install from Source Code* or use one of the alternatives proposed at the end of *Welcome to the Sage Installation Guide!*.

If you have any problems building or running Sage, please take a look at the Installation FAQ in the *Sage Release Tour* corresponding to the version that you are installing. It may offer version-specific installation help that has become available after the release was made and is therefore not covered by this manual.

Also please do not hesitate to ask for help in the *SageMath forum* or the sage-support mailing list at <https://groups.google.com/forum/#!forum/sage-support>.

Also note the following. Each separate component of Sage is contained in an SPKG; these are stored in `build/pkgs/`. As each one is built, a build log is stored in `logs/pkgs/`, so you can browse these to find error messages. If an SPKG fails to build, the whole build process will stop soon after, so check the most recent log files first, or run:

```
grep -li "^Error" logs/pkgs/*
```

from the top-level Sage directory to find log files with error messages in them. Send the file `config.log` as well as the log file(s) of the packages that have failed to build in their entirety to the sage-support mailing list at <https://groups.google.com/group/sage-support>; probably someone there will have some helpful suggestions.

This work is licensed under a [Creative Commons Attribution-Share Alike 3.0 License](https://creativecommons.org/licenses/by-sa/3.0/).

INDEX

B

BROWSER, 28

C

CC, 28

CFLAGS, 28

CPP, 28

CPPFLAGS, 28

CXX, 28

CXXFLAG64, 28

CXXFLAGS, 28

CYGWIN_ROOT, 24

D

DOT_SAGE, 28

E

environment variable

BROWSER, 28

CC, 28

CFLAGS, 28

CPP, 28

CPPFLAGS, 28

CXX, 28

CXXFLAG64, 28

CXXFLAGS, 28

CYGWIN_ROOT, 24

DOT_SAGE, 28

FC, 28

FCFLAGS, 28

LD, 28

LDFLAG64, 28

LDFLAGS, 28

MAKE, 25

MAKEFLAGS, 25

OPENBLAS_CONFIGURE, 28

PARI_CONFIGURE, 27

PARI_MAKEFLAGS, 27

PATH, 22, 23

SAGE_BUILD_DIR, 26, 27

SAGE_CHECK, 26

SAGE_CHECK_PACKAGES, 26

SAGE_DEBUG, 26

SAGE_DOCBUILD_OPTS, 26

SAGE_FAT_BINARY, 27

SAGE_INSTALL_CCACHE, 26

SAGE_INSTALL_GCC, 26

SAGE_KEEP_BUILT_SPKGS, 27

SAGE_LOCAL, 27

SAGE_MATPLOTLIB_GUI, 27

SAGE_NUM_THREADS, 25

SAGE_PROFILE, 26

SAGE_ROOT, 24

SAGE_SERVER, 25

SAGE_SPKG_INSTALL_DOCS, 26

SAGE_STARTUP_FILE, 28

SAGE_SUDO, 27

SAGE_TEST_GLOBAL_ITER, 28

SAGE_TEST_ITER, 28

SAGE_TIMEOUT, 28

SAGE_TIMEOUT_LONG, 28

SAGE_TUNE_PARI, 27

V, 25

F

FC, 28

FCFLAGS, 28

L

LD, 28

LDFLAG64, 28

LDFLAGS, 28

M

MAKE, 25

MAKEFLAGS, 25

O

OPENBLAS_CONFIGURE, 28

P

PARI_CONFIGURE, 27

PARI_MAKEFLAGS, 27

PATH, 22, 23

S

SAGE_BUILD_DIR, 26, 27
SAGE_CHECK, 26
SAGE_CHECK_PACKAGES, 26
SAGE_DEBUG, 26
SAGE_DOCBUILD_OPTS, 26
SAGE_FAT_BINARY, 27
SAGE_INSTALL_CCACHE, 26
SAGE_INSTALL_GCC, 26
SAGE_KEEP_BUILT_SPKGS, 27
SAGE_LOCAL, 27
SAGE_MATPLOTLIB_GUI, 27
SAGE_NUM_THREADS, 25
SAGE_PROFILE, 26
SAGE_ROOT, 24
SAGE_SERVER, 25
SAGE_SPKG_INSTALL_DOCS, 26
SAGE_STARTUP_FILE, 28
SAGE_SUDO, 27
SAGE_TEST_GLOBAL_ITER, 28
SAGE_TEST_ITER, 28
SAGE_TIMEOUT, 28
SAGE_TIMEOUT_LONG, 28
SAGE_TUNE_PARI, 27

V

V, 25