Installation Guide

Release 10.6

The Sage Development Team

Apr 02, 2025

CONTENTS

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.

CHAPTER

ONE

MACOS

- **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*; 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*.

CHAPTER

TWO

WINDOWS

- **Do you want to do SageMath development?**
  - **Yes, development:**

    Enable Windows Subsystem for Linux (WSL) and install Ubuntu as follows.

    * Make sure that hardware-assisted virtualization is enabled in the EFI or BIOS of your system. If in doubt, refer to your system's documentation for instructions on how to do this.
    * Run the WSL install command as administrator. This will install Ubuntu Linux.

      Note that the basic instructions in the linked article apply to up-to-date installations of Windows 10 and 11, but there are also links to the procedures for older builds of Windows 10.
    * If you had installed WSL previously or installed it using different instructions, verify that you are running WSL 2.
    * Set up your Linux username and password. Do not include any spaces in your username.
    * If your computer has less than 10GB of RAM, change the WSL settings to make at least 5GB of RAM available to WSL.

    Start Ubuntu from the Start menu. Then follow the instructions for development on Linux below.
  - **No development:**

    Enable Windows Subsystem for Linux (WSL) and install Ubuntu as follows.

    * Make sure that hardware-assisted virtualization is enabled in the EFI or BIOS of your system. If in doubt, refer to your system's documentation for instructions on how to do this.
    * Run the WSL install command as administrator. This will install Ubuntu Linux.

      Note that the basic instructions in the linked article apply to up-to-date installations of Windows 10 and 11, but there are also links to the procedures for older builds of Windows 10.
    * If you had installed WSL previously or installed it using different instructions, verify that you are running WSL 2.
    * Set up your Linux username and password. Do not include any spaces in your username.
    * If your computer has less than 8GB of RAM, change the WSL settings to make at least 4GB of RAM available to WSL.

    Start Ubuntu from the Start menu, and type the following commands to install Sage from conda-forge. (The `$` represents the command line prompt, don't type it!) The second step will ask a few questions, and you may need to hit `Enter` to confirm or type `yes` and then hit `Enter`.

```
$ curl -L -O "https://github.com/conda-forge/miniforge/releases/latest/
↪download/Miniforge3-$(uname)-$(uname -m).sh"
$ bash Miniforge3-$(uname)-$(uname -m).sh
$ conda create -n sage sage python=3.11
```

(If there are any installation failures, please report them to the conda-forge maintainers by opening a GitHub Issue for conda-forge/sage-feedstock.)

You can now start SageMath as follows:

```
$ conda activate sage
$ sage
```

This way of starting Sage gives you the most basic way of using Sage in the terminal. See *Launching SageMath* for recommended next steps, in particular for setting up the Jupyter notebook, which is required if you want to use graphics.

CHAPTER

THREE

LINUX

- **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*; 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. Some Linux distributions have up-to-date versions of SageMath, see 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 9.5.
    * **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*.

CHAPTER

FOUR

IN THE CLOUD

- Sage Binder repo provides a Binder badge to launch JupyterLab environment with Sage.
- Sage Cell Server is a free online service for quick computations with Sage.
- CoCalc is an online commercial service that provides Sage and many other tools.
- Docker image sagemathinc/cocalc can be used on any system with Docker to run CoCalc locally.

More information:

4.1 Linux Package Managers

SageMath is available from various distributions and can be installed by package managers.

As of Sage 10.2, we can recommend the following distributions, which provide well-maintained and up-to-date SageMath packages: Arch Linux and Void Linux.

Gentoo users might want to give a try to sage-on-gentoo.

Do not install a version of Sage older than 9.5. 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.

See the _sagemath dummy package for the names of packages that provide a standard installation of SageMath, including documentation and Jupyter. See also repology.org: sagemath for information about versions of SageMath packages in various distributions.

The GitHub 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

SageMath used to provide pre-built binaries for Windows based on Cygwin. This has been discontinued, and the old binaries that can be found are no longer supported. Use Windows Subsystem for Linux instead.

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, M2, M3, M4) are supported.

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

```
$ curl -L -O "https://github.com/conda-forge/miniforge/releases/latest/download/
↪Miniforge3-$(uname)-$(uname -m).sh"
$ bash Miniforge3-$(uname)-$(uname -m).sh
```

- Miniforge (and Mambaforge) 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`

If you installed Miniforge (or Mambaforge), we recommend to use mamba in the following, which uses a faster dependency solver than `conda`.

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
```

```
$ conda create -n sage sage python=X
```

where `X` is version of Python, e.g. `3.12`.

To use Sage from there,

- Enter the new environment: `conda activate sage`
- Start SageMath: `sage`

If there are any installation failures, please report them to the conda-forge maintainers by opening a GitHub Issue for conda-forge/sage-feedstock.

4.3.2 Using conda to provide all dependencies for the Sage library

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.

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
```

- Create and activate a new conda environment with the dependencies of Sage and a few additional developer tools:

```
$ mamba env create --file environment-3.12-linux.yml --name sage-dev
$ conda activate sage-dev
```

```
$ conda env create --file environment-3.12-linux.yml --name sage-dev
$ conda activate sage-dev
```

  Alternatively, you can use `environment-3.12-linux.yml` or `environment-optional-3.12-linux.yml`, which will only install standard (and optional) packages without any additional developer tools.

  A different Python version can be selected by replacing `3.12` with the desired version.

- Bootstrap the source tree and install the build prerequisites and the Sage library:

```
$ ./bootstrap
$ pip install --no-build-isolation --config-settings editable_mode=compat -v -v --
↪editable ./src
```

  If you encounter any errors, try to install the `sage-conf` package first:

```
$ pip install --no-build-isolation -v -v --editable ./pkgs/sage-conf_conda
```

  and then run the last command again.

- Verify that Sage has been installed:

```
$ sage -c 'print(version())'
SageMath version 10.2.beta4, Release Date: 2023-09-24
```

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 the Sage library using:

```
$ pip install --no-build-isolation --config-settings editable_mode=compat -v -v --
↪editable src
```

In order to update the conda environment later, you can run:

```
$ mamba env update --file environment-3.12-linux.yml --name sage-dev
```

To build the documentation, use:

```
$ pip install --no-build-isolation -v -v --editable ./pkgs/sage-docbuild
$ sage --docbuild all html
```

Note

The switch `--config-settings editable_mode=compat` restores the legacy setuptools implementation of editable installations. Adventurous developers may omit this switch to try the modern, PEP-660 implementation of editable installations, see Issue #34209.

Note

You can update the conda lock files by running `.github/workflows/conda-lock-update.py` or by running `conda-lock --platform linux-64 --filename environment-3.12-linux.yml --lockfile environment-3.12-linux.lock` manually.

4.4 Install from 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 packages 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. Some familiarity with the use of the Unix command line may be required to build Sage from the source code.

4.4.1 Prerequisites

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).

Software prerequisites and recommended packages

Sage depends on a large number of software packages. Sage provides its own software distribution providing most of these packages, so you do not have to worry about having to download and install these packages yourself.

If you extracted Sage from a source tarball, the subdirectory `upstream` contains the source distributions for all standard packages 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.

However, there are minimal prerequisites for building Sage that already must be installed on your system:

- Fundamental system packages required for installing from source
- C/C++ compilers

If you have sufficient privileges (for example, on Linux you can use `sudo` to become the `root` user), then you can install these packages using the commands for your platform indicated in the pages linked above. If you do not have the privileges to do this, ask your system administrator to do this for you.

In addition to these minimal prerequisites, we strongly recommend to use system installations of the following:

- Fortran compiler
- Python

Sage developers will also need the system packages required for bootstrapping; they cannot be installed by Sage.

When the `./configure` script runs, it will check for the presence of many packages (including the above) and inform you of any that are missing or have unsuitable versions. **Please read the messages that ./configure prints:** It will inform you which additional system packages you can install to avoid having to build them from source. This can save a lot of time.

The following sections provide the commands to install a large recommended set of packages on various systems, which will minimize the time it takes to build Sage. This is intended as a convenient shortcut, but of course you can choose to take a more fine-grained approach.

Linux system package installation

We recommend that you install the following packages, depending on your distribution:

```
$ sudo apt-get install bc binutils bzip2 ca-certificates cliquer cmake curl \
      ecl eclib-tools fflas-ffpack flex g++ gap 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-dev libfplll-dev libfreetype-dev \
      libgap-dev libgc-dev libgd-dev libgf2x-dev libgivaro-dev libglpk-dev \
      libgmp-dev libgsl-dev libhomfly-dev libiml-dev liblfunction-dev \
      liblinbox-dev liblrcalc-dev liblzma-dev libm4ri-dev libm4rie-dev \
      libmpc-dev libmpfi-dev libmpfr-dev libncurses5-dev libntl-dev \
      libopenblas-dev libpari-dev libplanarity-dev libppl-dev \
      libprimecount-dev libprimesieve-dev libpython3-dev libqhull-dev \
      libreadline-dev librw-dev libsingular4-dev libsqlite3-dev libssl-dev \
      libsuitesparse-dev libsymmetrica2-dev libz-dev libzmq3-dev m4 make \
      maxima maxima-sage meson nauty ninja-build openssl palp pari-doc \
      pari-elldata pari-galdata pari-galpol pari-gp2c pari-seadata patch \
      patchelf perl pkg-config planarity ppl-dev python3 python3-setuptools \
      python3-venv qhull-bin singular singular-doc sqlite3 sympow tachyon \
      tar texinfo tox xz-utils
```

```
$ sudo yum install --setopt=tsflags= L-function L-function-devel Singular \
      Singular-devel binutils boost-devel brial brial-devel bzip2 \
      bzip2-devel cddlib cddlib-devel cliquer cliquer-devel cmake curl \
      diffutils ecl eclib eclib-devel fflas-ffpack-devel findutils flex \
      flint flint-devel gap gap-core gap-devel gap-libs gap-pkg-ace \
      gap-pkg-aclib gap-pkg-alnuth gap-pkg-anupq gap-pkg-atlasrep \
      gap-pkg-autodoc gap-pkg-automata gap-pkg-autpgrp gap-pkg-browse \
      gap-pkg-caratinterface gap-pkg-circle gap-pkg-congruence gap-pkg-crisp \
      gap-pkg-crypting gap-pkg-crystcat gap-pkg-curlinterface gap-pkg-cvec \
      gap-pkg-datastructures gap-pkg-digraphs gap-pkg-edim gap-pkg-ferret \
      gap-pkg-fga gap-pkg-fining gap-pkg-float gap-pkg-format gap-pkg-forms \
      gap-pkg-fplsa gap-pkg-fr gap-pkg-francy gap-pkg-genss \
      gap-pkg-groupoids gap-pkg-grpconst gap-pkg-images gap-pkg-io \
      gap-pkg-irredsol gap-pkg-json gap-pkg-jupyterviz gap-pkg-lpres \
      gap-pkg-nq gap-pkg-openmath gap-pkg-orb gap-pkg-permut gap-pkg-polenta \
      gap-pkg-polycyclic gap-pkg-primgrp gap-pkg-profiling gap-pkg-radiroot \
      gap-pkg-recog gap-pkg-resclasses gap-pkg-scscp gap-pkg-semigroups \
      gap-pkg-singular gap-pkg-smallgrp gap-pkg-smallsemi gap-pkg-sophus \
      gap-pkg-spinsym gap-pkg-standardff gap-pkg-tomlib gap-pkg-transgrp \
```

(continues on next page)

(continued from previous page)

```
      gap-pkg-transgrp-data gap-pkg-utils gap-pkg-uuid gap-pkg-xmod \
      gap-pkg-zeromqinterface gc gc-devel gcc gcc-c++ gcc-gfortran gd \
      gd-devel gengetopt gf2x gf2x-devel gfan givaro givaro-devel glpk \
      glpk-devel glpk-utils gmp gmp-devel gmp-ecm gmp-ecm-devel gsl \
      gsl-devel iml iml-devel info libatomic_ops libatomic_ops-devel \
      libbraiding-devel libcurl-devel libffi libffi-devel libfplll \
      libfplll-devel libgap libhomfly-devel libmpc libmpc-devel linbox \
      linbox-devel lrcalc-devel m4 m4ri-devel m4rie-devel make mathjax3 \
      maxima maxima-runtime-ecl meson mpfr-devel nauty ncurses-devel \
      ninja-build ntl-devel openblas-devel openssl openssl-devel palp \
      pari-devel pari-elldata pari-galdata pari-galpol pari-gp pari-seadata \
      patch patchelf perl perl-ExtUtils-MakeMaker perl-IPC-Cmd pkg-config \
      planarity planarity-devel ppl ppl-devel primecount primecount-devel \
      primesieve primesieve-devel python3 python3-devel python3-setuptools \
      python3-virtualenv qhull qhull-devel readline-devel rw-devel sqlite \
      sqlite-devel suitesparse suitesparse-devel symmetrica-devel sympow \
      tachyon tachyon-devel tar texinfo tox which xgap xz xz-devel zeromq \
      zeromq-devel zlib-devel
```

```
$ sudo pacman -S bc binutils boost brial cblas cddlib cliquer cmake ecl \
      eclib fflas-ffpack fplll gap gc gcc gcc-fortran gd gf2x gfan glpk gsl \
      iml lapack lcalc libatomic_ops libbraiding libhomfly linbox lrcalc m4 \
      m4ri m4rie make maxima-fas meson nauty ninja openblas openssl palp \
      pari pari-elldata pari-galdata pari-galpol pari-seadata patch perl \
      pkgconf planarity ppl primecount primesieve python python-tox qhull \
      rankwidth readline singular sqlite3 suitesparse symmetrica sympow \
      tachyon tar which zeromq
```

```
$ sudo zypper install bc binutils boost-devel brial-devel bzip2 \
      ca-certificates cddlib-tools cliquer cliquer-devel cmake curl \
      diffutils edge-addition-planarity-suite \
      edge-addition-planarity-suite-devel findutils flint-devel fplll \
      fplll-devel gawk gcc gcc-c++ gcc-fortran gd gfan glibc-locale-base \
      glpk glpk-devel gmp-devel gzip iml-devel libbraiding-devel \
      libhomfly-devel libopenssl-3-devel libprimecount-devel m4 make mathjax \
      maxima-exec-clisp meson mpc-devel mpfi-devel nauty nauty-devel ninja \
      ntl-devel openblas-devel pari-devel pari-galdata pari-gp patch \
      patchelf perl pkgconf pkgconfig\(atomic_ops\) pkgconfig\(bdw-gc\) \
      pkgconfig\(bzip2\) pkgconfig\(cddlib\) pkgconfig\(fflas-ffpack\) \
      pkgconfig\(fplll\) pkgconfig\(freetype2\) pkgconfig\(gdlib\) \
      pkgconfig\(gf2x\) pkgconfig\(givaro\) pkgconfig\(gsl\) \
      pkgconfig\(libcurl\) pkgconfig\(libffi\) pkgconfig\(liblzma\) \
      pkgconfig\(libpng16\) pkgconfig\(libzmq\) pkgconfig\(linbox\) \
      pkgconfig\(m4ri\) pkgconfig\(m4rie\) pkgconfig\(mpfr\) \
      pkgconfig\(ncurses\) pkgconfig\(ncursesw\) pkgconfig\(readline\) \
      pkgconfig\(sqlite3\) pkgconfig\(zlib\) ppl-devel primecount primesieve \
      python3 python3-devel python3-setuptools qhull-devel readline-devel \
      suitesparse-devel sympow tachyon tar texinfo which xz
```

```
$ sudo xbps-install SuiteSparse-devel bash bc binutils boost-devel \
      brial-devel bzip2-devel cddlib-devel cliquer-devel cmake curl \
```

(continues on next page)

(continued from previous page)

```
      diffutils ecl eclib-devel ecm-devel fflas-ffpack flintlib-devel \
      fplll-devel freetype-devel gc-devel gcc gcc-fortran gd-devel gengetopt \
      gf2x-devel gfan givaro-devel glpk-devel gmp-devel gmpxx-devel \
      gsl-devel gzip iml-devel lcalc-devel libatomic_ops-devel \
      libbraiding-devel libcurl-devel libffi-devel libgomp-devel \
      libhomfly-devel liblzma-devel libmpc-devel libpng-devel libqhull-devel \
      libxcrypt-devel linbox-devel lrcalc-devel m4 m4ri-devel m4rie-devel \
      make mathjax maxima-ecl mpfi-devel mpfr-devel nauty ncurses-devel \
      ninja ntl-devel openblas-devel openssl-devel palp pari pari-devel \
      pari-elldata-small pari-galdata pari-galpol-small pari-seadata patch \
      patchelf perl pkgconf planarity-devel ppl-devel primecount-devel \
      primesieve-devel python3 python3-appdirs python3-devel python3-distlib \
      python3-filelock python3-setuptools python3-virtualenv qhull \
      rankwidth-devel readline-devel singular sqlite-devel symmetrica-devel \
      sympow tachyon tar texinfo tox which xz zeromq-devel zlib-devel
```

If you wish to do Sage development, we recommend that you additionally install the following:

```
$ sudo apt-get install autoconf automake gh git gpgconf libtool \
      openssh-client pkg-config
```

```
$ sudo yum install autoconf automake gh git gnupg2 libtool openssh \
      pkg-config
```

```
$ sudo pacman -S autoconf automake git github-cli gnupg libtool openssh \
      pkgconf
```

```
$ sudo zypper install autoconf automake gh git gpg2 libtool openssh \
      pkgconfig
```

```
$ sudo xbps-install autoconf automake git github-cli gnupg2 libtool \
      mk-configure openssh pkg-config xtools
```

For all users, we recommend that you install the following system packages, which provide additional functionality and cannot be installed by Sage. In particular, this includes LaTeX and related tools. In addition to a base install of TeX Live, our lists of system packages below include everything that is needed for generating the Sage documentation in PDF format. For converting Jupyter notebooks to PDF, also the document converter pandoc is needed. For making animations, Sage needs to use one of the packages FFmpeg and ImageMagick.

```
$ sudo apt-get install default-jdk dvipng ffmpeg fonts-freefont-otf \
      imagemagick latexmk libavdevice-dev libjpeg-dev pandoc tex-gyre \
      texlive-fonts-recommended texlive-lang-cyrillic texlive-lang-english \
      texlive-lang-european texlive-lang-french texlive-lang-german \
      texlive-lang-italian texlive-lang-japanese texlive-lang-polish \
      texlive-lang-portuguese texlive-lang-spanish texlive-latex-extra \
      texlive-luatex texlive-xetex xindy
```

```
$ sudo yum install ImageMagick ffmpeg-free ffmpeg-free-devel \
      gnu-free-mono-fonts gnu-free-sans-fonts gnu-free-serif-fonts latexmk \
      libjpeg-turbo-devel pandoc texlive texlive-collection-langcyrillic \
      texlive-collection-langeuropean texlive-collection-langfrench \
```

(continues on next page)

(continued from previous page)

```
      texlive-collection-langgerman texlive-collection-langitalian \
      texlive-collection-langjapanese texlive-collection-langpolish \
      texlive-collection-langportuguese texlive-collection-langspanish \
      texlive-collection-latexextra texlive-gnu-freefont texlive-luatex \
      texlive-xindy
```

```
$ sudo pacman -S ffmpeg gnu-free-fonts imagemagick libjpeg-turbo pandoc \
      texlive-core texlive-langcyrillic texlive-langjapanese \
      texlive-latexextra texlive-luatex
```

```
$ sudo zypper install ImageMagick ffmpeg gnu-free-fonts libjpeg-devel pandoc \
      texlive texlive-luatex xindy
```

```
$ sudo xbps-install ImageMagick ffmpeg freefont-ttf libjpeg-turbo-devel \
      pandoc texlive
```

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 Issue #27330, install:

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

```
$ sudo yum install 4ti2 4ti2-devel bliss bliss-devel clang coin-or-Cbc \
      coin-or-Cbc-devel coxeter coxeter-devel coxeter-tools giac giac-devel \
      gp2c graphviz graphviz-devel igraph igraph-devel isl isl-devel \
      libnauty libnauty-devel libsemigroups libsemigroups-devel \
      libxml2-devel lrslib lrslib-devel pari-galpol pari-seadata pdf2svg \
      perl-ExtUtils-Embed perl-File-Slurp perl-JSON perl-MongoDB perl-SVG \
      perl-Term-ReadLine-Gnu perl-TermReadKey perl-XML-LibXML \
      perl-XML-LibXSLT perl-XML-Writer polymake sbcl tbb tbb-devel
```

```
$ sudo pacman -S 4ti2 bliss clang coin-or-cbc coxeter giac graphviz igraph \
      intel-oneapi-tbb libxml2 lrs pari-elldata pari-galpol pari-seadata \
      pdf2svg perl-term-readline-gnu polymake sbcl symengine
```

```
$ sudo zypper install 4ti2 4ti2-devel bliss bliss-devel coxeter fricas \
      giac-devel gp2c graphviz libxml2 llvm lrslib lrslib-devel pari-elldata \
      pari-galpol pari-nftables pari-seadata pdf2svg \
      perl\(Term::ReadLine::Gnu\) pkgconfig\(isl\) \
      pkgconfig\(libsemigroups\) polymake sbcl symengine tbb
```

```
$ sudo xbps-install CoinMP-devel clang giac-devel gp2c graphviz \
      graphviz-devel igraph-devel isl-devel libxml2-devel nauty-devel \
      pari-elldata-small pari-galpol-small pari-nftables pari-seadata \
      perl-File-Slurp perl-JSON perl-SVG perl-Term-ReadKey \
      perl-Term-ReadLine-Gnu perl-XML-LibXML perl-XML-LibXSLT \
      perl-XML-Writer sbcl tbb-devel
```

macOS prerequisites

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 package installation

If you use the Homebrew package manager, you can install the following:

```
$ brew install bdw-gc boost bzip2 cddlib cmake curl ecl flint fplll freetype \
      gcc gd gengetopt gfortran glpk gmp gpatch gsl libatomic_ops libffi \
      libiconv libmpc libpng maxima meson mpfi mpfr nauty ncurses ninja ntl \
      openblas openssl pari pari-elldata pari-galdata pari-galpol \
      pari-seadata patchelf pkg-config ppl primecount primesieve \
      python-setuptools python3 qhull readline singular sqlite suite-sparse \
      texinfo 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.

If you wish to do Sage development, we recommend that you additionally install the following:

```
$ brew install autoconf automake gh git gnupg libtool pkg-config
```

For all users, we recommend that you install the following system packages, which provide additional functionality and cannot be installed by Sage:

```
$ brew install ffmpeg imagemagick jpeg-turbo pandoc texinfo
```

Some additional optional packages are taken care of by:

```
$ brew install apaffenholz/polymake/polymake cbc graphviz igraph isl libxml2 \
      llvm nauty pdf2svg sbcl symengine tbb
```

WSL prerequisites

Ubuntu on Windows Subsystem for Linux (WSL) prerequisite installation

Refer to *Windows* for installing Ubuntu on Windows Subsystem for Linux (WSL). These instructions describe a fresh install of Ubuntu, the default distribution in WSL, but other distributions or installation methods should work too.

From this point on, follow the instructions in the *Linux system package 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/...`).

WSL permission denied error when building `packaging` package

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.

WSL post-installation notes

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.

Tcl/Tk (and system's Python)

If you want to use Tcl/Tk libraries in Sage, and you are going to use your OS's Python3 as Sage's Python, you merely need to install its **Tkinter** module. On Linux systems, it is usually provided by the **python3-tk** or a similarly named (e.g. **python3-tkinter**) package, which can be installed using:

```
$ sudo apt-get install python3-tk
```

or similar commands.

Tcl/Tk (and Sage's own Python)

If you want to use Tcl/Tk libraries in Sage, and you are going to build Sage's Python from source, you need to install these, and the corresponding headers. 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.

Sage's Python will then automatically recognize your system's install of Tcl/Tk. 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.

4.4.2 Installation steps

Hint

The following steps use the classical `./configure && make` build process. The modern Meson build system is also supported, see *Building from source using Meson*.

1. Follow the procedure in the file README.md in `SAGE_ROOT`.
2. If you wish to prepare for having to build Sage in an environment without sufficient Internet connectivity:
   - After running `configure`, you can use `make download` to force downloading packages before building. After this, the packages are in the subdirectory `upstream`.
   - Alternatively, instead of cloning the git repository, you can download a self-contained release tarball for any stable release from the Sage project's GitHub Releases. Use the file named `sage-x.y.tar.gz` (1.25 GB as of Sage 10.2) in the Release Assets, which contains a prepopulated subdirectory `upstream`.

     After downloading the source tarball `sage-x.y.tar.gz` into a directory `~/sage/`:

     ```
     $ cd ~/sage/
     $ tar xf sage-x.y.tar.gz  # adapt x.y; takes a while
     ```

     This creates the subdirectory `sage-x.y`. Now change into it:

     ```
     $ cd sage-x.y/  # adapt x.y
     ```

     Note

     On Windows, it is crucial that you unpack the source tree from the WSL *bash* using the WSL *tar* utility and not using other Windows tools (including mingw).

     This is because the Sage source tree contains symbolic links, and the build will not work if Windows line endings rather than UNIX line endings are used.

- The Sage mirrors also provide such self-contained tarballs for all stable releases and additionally for all development releases.

3. Additional remarks: You do not need to be logged in as root, since no files are changed outside of the `SAGE_ROOT` 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 installation prefix. 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, 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.

4. 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 3.10.4. 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
```

5. 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!

6. 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 "$@"
```

7. 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_root/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 my 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.

8. Optional: Install optional Sage packages and databases. See the list of optional packages in the reference manual for detailed information, or type `sage --optional` (this requires an Internet connection).

   Then type `sage -i <package-name>` to automatically download and install a given package.

9. Have fun! Discover some amazing conjectures!

4.4.3 Make targets

To build Sage from scratch, you would typically execute `make` in Sage's home directory to build Sage and its documentation in HTML format, suitable for viewing in a web browser.

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.

4.4.4 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 *MAKEFLAGS*, as described below, can significantly speed up the process.) Building Sage involves building many packages, each of which has its own compilation instructions.

Standard environment controlling the build process

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

MAKEFLAGS

This variable can be set to tell the `make` program to build things in parallel. Set it to `-jNUM` to run `NUM` jobs in parallel when building. Add `-lNUM` to tell make not to spawn more processes when the load exceeds `NUM`.

A good value for this variable is `MAKEFLAGS="-j$(nproc) -l$(nproc).5"` on Linux and `MAKEFLAGS="-j$(sysctl -n hw.ncpu) -l$(sysctl -n hw.ncpu).5"` on macOS. This instructs make to use all the execution threads of your CPU while bounding the load if there are other processes generating load. If your system does not have a lot of RAM, you might want to choose lower limits, if you have lots of RAM, it can sometimes be beneficial to set these limits slightly higher.

Note that some parts of the SageMath build system do not respect this variable, e.g., when ninja gets invoked, it figures out the number of processes to use on its own so the number of processes and the system load you see might exceed the number configured here.

See the manual page for GNU `make`: Command-line options and Parallel building.

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.)

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

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.

CPPFLAGS

LDFLAGS

CXXFLAG64

LDFLAG64

LD

Similar comments apply to these compiler and linker flags.

Sage-specific environment variables controlling the build process

SAGE_SERVER

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).

If *SAGE_SERVER* is set, the specified Sage mirror is contacted first. Note that Sage will search the directory `SAGE_SERVER/spkg/upstream` for upstream tarballs.

If downloading a file from there fails or *SAGE_SERVER* is not set, files will be attempted to download from release assets of the Sage GitHub repository.

If that fails too, the Sage mirror network is contacted to determine the nearest mirrors.

This sequence of operations is defined by the files in the directory SAGE_ROOT/.upstream.d.

SAGE_NUM_THREADS

If set to a number, then when rebuilding with `sage -b` or parallel doctesting with `sage -t -p 0`, use at most 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,

- `sage -b` only uses one thread
- `sage -t -p 0` uses a default of the number of CPU cores, with a maximum of 8 and a minimum of 2.

When `sage -t -p` runs under the control of the GNU `make` jobserver, then Sage will request as most this number of job slots.

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_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 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 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 controlling the documentation build

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:

- add `--no-plot` to this variable to avoid building the graphics coming from the `.. PLOT` directive within the documentation,
- add `--no-preparsed-examples` to only show the original Sage code of "EXAMPLES" blocks, suppressing the tab with the preparsed, plain Python version, or
- 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_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_USE_CDNS

If set to `yes`, then build the documentation using CDNs (Content Distribution Networks) for scripts necessary for HTML documentation, such as MathJax.

SAGE_LIVE_DOC

If set to `yes`, then build live Sage documentation. If the `Make live` button on any webpage of the live doc is clicked, every example code gets a CodeMirror code cell runnable via Thebe. Thebe is responsible in sending the code to the Sage computing environment built by Binder and showing the output result. The Sage computing environment can be specified to either a Binder repo or a local Jupyter server. The environment variable *SAGE_JUPYTER_SERVER* is used for this purpose.

SAGE_JUPYTER_SERVER

Set this to either `binder`, `binder:repo` with `repo` specifying a Binder repo or the URL to a local Jupyter server.

- `binder` refers to Sage's official Binder repo. This is assumed if the environment variable *SAGE_JUPYTER_SERVER* is not set.
- `binder:repo` specifies a Binder repo with `repo`, which is a GitHub repository name, optionally added with a branch name with `/` separator.
- To use a local Jupyter server instead of Binder, then set the URL to *SAGE_JUPYTER_SERVER* and the secret token to environment variable `SAGE_JUPYTER_SERVER_TOKEN`, which can be left unset if the default token `secret` is used. If the live doc was built with `SAGE_JUPYTER_SERVER=http://localhost:8889`, run a local Jupyter server by

```
./sage --notebook=jupyterlab \
       --ServerApp.token='secret' \
       --ServerApp.allow_origin='null' \
       --ServerApp.disable_check_xsrf=true \
       --ServerApp.port=8889 \
       --ServerApp.open_browser=false
```

  before opening the Sage documentation webpage.

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.

OPENBLAS_CONFIGURE

Adds additional configuration flags for the OpenBLAS package that gets added to the `make` command. (see Issue #23272)

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.in` 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.

Environment 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).

Environment variables set within Sage environments

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.

4.4.5 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.

1. Using `sudo`, create the installation directory, for example, `/opt/sage/sage-x.y`. We refer to it as `SAGE_LOCAL` in the instructions below. Do not try to install into a directory that already contains other software, such as `/usr/local`:

```
$ sudo mkdir -p SAGE_LOCAL
```

2. Make the directory writable for you and readable by everyone:

```
$ sudo chown $(id -un) SAGE_LOCAL
$ sudo chmod 755 SAGE_LOCAL
```

3. Build and install Sage, following the instructions in README.md, using the `configure` option `--prefix=SAGE_LOCAL`.

   Do not use `sudo` for this step; building Sage must be done using your normal user account.

4. Optionally, create a symbolic link to the installed `sage` script in a directory that is in the users' `PATH`, for example `/usr/local/bin`:

```
$ sudo ln -s SAGE_LOCAL/bin/sage /usr/local/bin/sage
```

5. Optionally, change permissions to prevent accidental changes to the installation by yourself:

```
$ sudo chown -R root SAGE_LOCAL
```

4.4.6 Upgrading the system and upgrading Sage

Caveats when upgrading system packages

When Sage has been installed from source, it will make use of various system packages; in particular, it will link to shared libraries provided by the system.

The system's package manager does not keep track of the applications that make use of the shared libraries. Therefore indiscriminate upgrades of system packages can break a Sage installation.

This can always be fixed by a full rebuild:

```
$ make distclean && make build
```

But this time-consuming step can often be avoided by just reinstalling a few packages. The command `make -j list-broken-packages` assists with this:

```
$ make -j list-broken-packages
make --no-print-directory auditwheel_or_delocate-no-deps
...
# Checking .../local/var/lib/sage/installed/bliss-0.73+debian-1+sage-2016-08-02.p0
...
Checking shared library file '.../local/lib/libumfpack.dylib'
Checking shared library file '.../local/var/tmp/sage/build/suitesparse-5.10.1/src/lib/
↪libsliplu.1.0.2.dylib'
Error during installcheck of 'suitesparse': .../local/var/tmp/sage/build/suitesparse-
↪5.10.1/src/lib/libsliplu.1.0.2.dylib
...
Uninstall broken packages by typing:

    make lcalc-SAGE_LOCAL-uninstall;
    make ratpoints-SAGE_LOCAL-uninstall;
    make r-SAGE_LOCAL-uninstall;
    make suitesparse-SAGE_LOCAL-uninstall;
```

After running the suggested commands, run:

```
$ make build
```

Upgrading Sage using a separate git worktree

When you have a working installation of Sage built from source and wish to try out a new version, we strongly recommend to use a separate git worktree, so that you can keep using your existing installation when something goes wrong.

Start from the directory created when you used `git clone`, perhaps `~/sage/sage/`. Let's verify that this is indeed a git repository by looking at the hidden `.git` subdirectory. It will looks like this, but the exact contents can vary:

```
[alice@localhost sage]$ ls .git
COMMIT_EDITMSG HEAD           branches       description    gitk.cache
index          logs           packed-refs    FETCH_HEAD     ORIG_HEAD
config         hooks          info           objects        refs
```

Good. Now let's see what worktrees already exist:

```
[alice@localhost sage]$ git worktree list
/home/alice/sage/sage                    c0ffeefe10 [master]
```

We see just one line, the directory created when you used `git clone`. We will call this the "main worktree" from now on. Next to the directory, you can see the abbreviated commit sha and the name of the branch that we're on (`master`).

To try out a new version of Sage, let's fetch it first from the main repository:

```
[alice@localhost sage]$ git fetch upstream 10.3.beta8
From https://github.com/sagemath/sage
 * tag                     10.3.beta8 -> FETCH_HEAD
```

Now let's create a new worktree. We need a name for it; it should start with `worktree-` but can be anything after that. Experience shows that worktrees are often repurposed later, and because a directory containing a Sage installation cannot be moved without breaking the installation in it, it may be a good idea to choose a memorable name without much meaning:

```
[alice@localhost sage]$ git worktree add worktree-purple FETCH_HEAD
Preparing worktree (detached HEAD 30b3d78fac)
Updating files: 100% (11191/11191), done.
HEAD is now at 30b3d78fac Updated SageMath version to 10.3.beta8
```

We now have a subdirectory `worktree-purple`. This is a "linked worktree":

```
[alice@localhost sage]$ git worktree list
/home/alice/sage/sage                    c0ffeefe10 [master]
/home/alice/sage/sage/worktree-purple    30b3d78fac (detached HEAD)
[alice@localhost sage]$ cd worktree-purple
[alice@localhost worktree-purple]$ cat VERSION.txt
SageMath version 10.3.beta8, Release Date: 2024-02-13
```

All worktrees created in this way share the same repository, so they have access to all branches:

```
[alice@localhost worktree-purple]$ git --no-pager branch -v
* (no branch) 30b3d78fac Updated SageMath version to 10.3.beta8
+ master      2a9a4267f9 Updated SageMath version to 10.2
```

In fact, `.git` here is not a directory, just a hidden file:

```
[alice@localhost worktree-purple]$ ls -l .git
-rw-r--r--  1 alice  staff  59 Feb 20 18:16 .git
```

In the new worktree, we now build Sage from scratch. This is completely independent of and will not disrupt your existing working installation in the main worktree.

We will refer again to the step-by-step instructions from the file README.md. Our worktree `worktree-purple` is the `SAGE_ROOT` for this purpose.

One thing that we can share between worktrees without worry is the directory `upstream`, where Sage caches downloaded archives of packages. To have the new worktree share it with the main worktree, let's create a symbolic link. This is an optional step that will avoid re-downloading files that you already have:

```
[alice@localhost worktree-purple]$ ln -s ../upstream/ .
```

Now let's build Sage, starting with the step:

```
[alice@localhost worktree-purple]$ make configure
```

Refer to the file README.md for the following steps.

4.5 Building from source using Meson

This is a short guide on how to build the Sage from source using Meson.

4.5.1 Walkthrough

Assume we're starting from a clean repo and a fully set up conda environment (modify `-linux` according to your operating system):

```
$ mamba env create --file environment-3.11-linux.yml --name sage-dev
$ conda activate sage-dev
```

Alternatively, install all build requirements as described in section *Prerequisites*. In the likely case that you have to install some dependencies manually, set the correct environment variables to point to the installed libraries:

```
$ export C_INCLUDE_PATH=$C_INCLUDE_PATH:/your/path/to/include
$ export CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH:/your/path/to/include
$ export LIBRARY_PATH=$LIBRARY_PATH:/your/path/to/lib
```

Note

If you have previously build Sage in-place, you first have to delete the already compiled files, e.g. with `shopt -s globstar` followed by `rm src/**/*.so` or `for f in src/**/*.so ; do mv "$f" "$f.old"; done`. Moreover, remove the old generated files with `find src/sage/ext/interpreters -type f ! -name 'meson.build' -delete`. Also uninstall the 'old' sage packages with `pip uninstall sage-conf sage-setup sagemath-standard`.

To compile and install the project in editable install, just use:

```
$ pip install --no-build-isolation --editable .
```

This will install Sage in the current Python environment. In a Conda environment, the `--no-build-isolation` flag is necessary to allow the build system to reuse the already installed build dependencies. If you don't use Conda, you can omit this flag.

You can then start Sage from the command line with `./sage` or run the tests with `./sage -t`.

Note

By using `pip install --editable` in the above steps, the Sage library is installed in editable mode. This means that when you only edit source files, there is no need to rebuild the library; it suffices to restart Sage. Note that this even works when you edit Cython files (they will be recompiled automatically), so you no longer need to manually compile after editing Cython files.

Note

Note that `make` is not used at all, nor is `configure`. This means that any Sage-the-distribution commands such as `sage -i` will not work.

Note

By default, Meson will automatically determine the number of jobs to run in parallel based on the number of CPU available. This can be adjusted by passing `--config-settings=compile-args=-jN` to `pip install`.

`--verbose` can be passed to `pip install`, then the meson commands internally used by pip will be printed out.

4.5.2 Background information

Under the hood, pip invokes meson to configure and build the project. We can also use meson directly as follows.

To configure the project, we need to run the following command:

```
$ meson setup builddir --prefix=$PWD/build-install
```

This will create a build directory `builddir` that will hold the build artifacts. The `--prefix` option specifies the directory where the Sage will be installed, and can be omitted when `pip` is used to install as explained below.

If pip is used as above with `--editable`, `builddir` is set to be `build/cp[Python major version][Python minor version]`, such as `build/cp311`.

To compile the project, run the following command:

```
$ meson compile -C builddir
```

Installing is done with the following command:

```
$ meson install -C builddir
```

This will then install in the directory specified by `--prefix`, e.g. `build-install/lib/python3.11/site-packages/sage`. Usually, this directory is not on your Python path, so you have to use:

```
$ PYTHONPATH=build-install/lib/python3.11/site-packages ./sage
```

When editable install is used, it is not necessary to reinstall after each compilation.

Alternatively, we can still use pip to install (which does not require specifying `--prefix` in advance and automatically works with conda environment):

```
$ pip install --no-build-isolation --config-settings=builddir=builddir --editable .
```

Tip

Package maintainers may want to specify further build options or need to install to a different directory than the install prefix. Both are supported naturally by Meson:

```
$ meson setup builddir --prefix=/usr --libdir=... -Dcpp_args=...
$ meson compile -C builddir
$ DESTDIR=/path/to/staging/root meson install -C builddir
```

With the default prefix being `/usr/local`, it may then install to `$DESTDIR/usr/local/lib/python3.12/site-packages/sage`.

See Meson's quick guide and Meson's install guide for more information.

4.5.3 Miscellaneous tips

The environment variable `MESONPY_EDITABLE_VERBOSE=1` can be set while running `./sage`, so that when Cython files are recompiled a message is printed out. See https://mesonbuild.com/meson-python/how-to-guides/editable-installs.html#verbose-mode.

If a new `.pyx` file is added, it need to be added to `meson.build` file in the containing directory.

Unlike the `make`-based build system which relies on header comments `# distutils: language = c++` to determine whether C++ should be used, Meson-based build system requires specifying `override_options:`

`['cython_language=cpp']` in the `meson.build` file. Similarly, dependencies need to be specified by `dependencies: [...]`.

4.6 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 in the part 6 of the "Installation steps" Section in *Installation steps*. 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.

You can pass extra parameters to this command. For example,

```
sage -n jupyter --port 8899
```

will run the Jupyter server on a port different from the default (8888). In particular on WSL, this is very useful because Jupyter may not be able to detect whether the default port is already taken by another instance of Jupyter running in Windows.

4.6.1 Environment variables

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.
- `TMPDIR` - this variable is used by Python, and hence by Sage; it gives the directory in which temporary files should be stored. This includes files used by the notebook. Some browsers have security settings which restrict the locations of files that they will access, and users may need to set this variable to handle this situation.
- See https://docs.python.org/3/using/cmdline.html#environment-variables for more variables used by Python (not an exhaustive list). A brief summary can also be obtained by running $python3 - -help - env$.

4.6.2 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.

4.6.3 WSL Post-installation steps

If you've installed SageMath 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
# 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\`
- 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/.

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

By default, SageMath installs itself as a Jupyter kernel in the same environment as the SageMath installation. This is the most convenient way to use SageMath in a Jupyter notebook. To check if the Sage kernel is available, start a Jupyter notebook and look for the kernel named `sagemath` in the list of available kernels. Alternatively, you can use the following command to check which kernels are available:

```
$ jupyter kernelspec list
Available kernels:
  python3     <path to env>/share/jupyter/kernels/python3
  sagemath    <path to env>/share/jupyter/kernels/sagemath
```

Note

The kernel is not automatically available if you have installed SageMath in editable mode (`pip install -e`). In that case, it is recommended to reinstall SageMath in a non-editable way.

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. Alternatively, use `jupyter kernelspec list` from the same environment where SageMath is installed to find the location of the SageMath kernel.

Now pick a name for the kernel that identifies it clearly and uniquely.

For example, if you install Sage from source tarballs, you could decide to include the version number in the name, such as `sagemath-9.6`. If you build SageMath from a clone of the git repository, it is better to choose a name that identifies the directory, perhaps `sagemath-dev` or `sagemath-teaching` because the version will change.

Now assuming that the Jupyter notebook can be started by typing `jupyter notebook`, the following command will install SageMath as a new kernel named `sagemath-dev`.

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

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/kernels/sagemath-dev
```

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/kernels/sagemath-dev
```

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="jmol")
```

Using Jupyter notebook through Visual Studio Code (VS Code) in WSL

If you have installed Sage on Windows using Windows Subsystem for Linux (WSL), it is convenient to use Visual Studio Code (VS Code) to interact with Sage.

Here are steps to use SageMath in a Jupyter notebook in VS Code:

- Install and run VS Code in Windows.
- Click the "Extension" icon on the left (or press `Ctrl` + `Shift` + `X`) to open a list of extensions. Install the "Remote - WSL" and "Jupyter" extensions.
- In the command palette (`Ctrl` + `Shift` + `P`), enter "Remote-WSL: New Window", and hit `Enter`.
- In the command palette, enter "Create: New Jupyter Notebook", and hit `Enter`.
- Click "Select Kernel" on the right (or press `Ctrl` + `Alt` + `Enter`), select SageMath, and hit `Enter`.

4.7 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 Sage Installation Guide*.

If you have any problems building or running Sage, please take a look at the 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.

INDEX

T