The SageTeX package[*]

Dan Drake and others[†]

August 12, 2020

1   Introduction

Why should the Haskell and R folks have all the fun? Literate Haskell is a popular way to mix Haskell source code and LaTeX documents. (Actually any kind of text or document, but here we're concerned only with LaTeX.) You can even embed Haskell code in your document that writes part of your document for you. Similarly, the R statistical computing environment includes Sweave, which lets you do the same thing with R code and LaTeX.

The SageTeX package allows you to do (roughly) the same thing with the Sage mathematics software suite (see http://sagemath.org) and LaTeX. (If you know how to write literate Haskell: the `\eval` command corresponds to `\sage`, and the `code` environment to the `sageblock` environment.) As a simple example, imagine in your document you are writing about how to count license plates with three letters and three digits. With this package, you can write something like this:

```
There are $26$ choices for each letter, and $10$ choices for
each digit, for a total of $26^3 \cdot 10^3 =
\sage{26^3*10^3}$ license plates.
```

and it will produce

There are 26 choices for each letter, and 10 choices for each digit, for a total of $26^3 \cdot 10^3 = 17576000$ license plates.

The great thing is, you don't have to do the multiplication. Sage does it for you. This process mirrors one of the great aspects of LaTeX: when writing a LaTeX document, you can concentrate on the logical structure of the document and trust LaTeX and its army of packages to deal with the presentation and typesetting. Similarly, with SageTeX, you can concentrate on the mathematical structure ("I need the product of 26^3 and 10^3") and let Sage deal with the base-10 presentation of the number.

A less trivial, and perhaps more useful example is plotting. You can include a plot of the sine curve without manually producing a plot, saving an EPS or PDF

---

[*]This document corresponds to SageTeX v3.5, dated 2020/08/12.

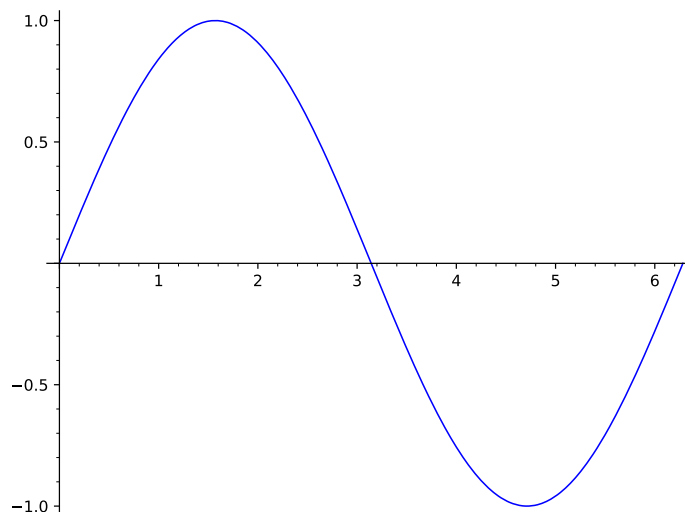[†]Package's website: github.com/sagemath/sagetex.

file, and doing the \includegraphics business with the correct filename yourself. If you write this:

> Here is a lovely graph of the sine curve:
>
> \sageplot[width=.75\textwidth]{plot(sin(x), x, 0, 2*pi)}

in your LATEX file, it produces

> Here is a lovely graph of the sine curve:

Again, you need only worry about the logical/mathematical structure of your document ("I need a plot of the sine curve over the interval $[0, 2\pi]$ here"), while SageTEX takes care of the gritty details of producing the file and sourcing it into your document.

But \sageplot isn't magic   I just tried to convince you that SageTEX makes putting nice graphics into your document very easy; let me turn around and warn you that using graphics *well* is not easy, and no LATEX package or Python script will ever make it easy. What SageTEX does is make it easy to *use Sage* to create graphics; it doesn't magically make your graphics good, appropriate, or useful. (For instance, look at the sine plot above—I would say that a truly lovely plot of the sine curve would not mark integer points on the x-axis, but rather $\pi/2$, π, $3\pi/2$, and 2π. Incidentally, you can do this in Sage: do `sage.plot.plot?` and look for `ticks` and `tick_formatter`.)

   Till Tantau has some good commentary on the use of graphics in the "Guidelines on Graphics" section of the PGF manual (chapter 7 of the manual for version 2.10). You should always give careful thought and attention to creating graphics for your document; I have in mind that a good workflow for using SageTEX for plotting is something like this:

1. Figure out what sort of graphic you need to communicate your ideas or information.

2. Fiddle around in Sage until you get a graphics object and set of options that produce the graphic you need.

3. Copy those commands and options into SageTeX commands in your LaTeX document.

The SageTeX package's plotting capabilities don't help you find those Sage commands to make your lovely plot, but they do eliminate the need to muck around with saving the result to a file, remembering the filename, including it into your document, and so on. In section 3, we will see what what we can do with SageTeX.

2 Installation

SageTeX needs two parts to work: a Python module known to Sage, and a LaTeX package known to TeX. These two parts need to come from the same version of SageTeX to guarantee that everything works properly. As of Sage version 4.3.1, SageTeX comes included with Sage, so you only need to make `sagetex.sty`, the LaTeX package, known to TeX. Full details of this are in the Sage tutorial at `doc.sagemath.org/html/en/tutorial/sagetex.html` in the obviously-named section "Make SageTeX known to TeX". Here's a brief summary of how to do that:

- Copy `sagetex.sty` to the same directory as your document. This always works, but requires lots of copies of `sagetex.sty` and is prone to version skew.

- Copy the directory containing `sagetex.sty` to your home directory with a command like

      cp -R $SAGE_ROOT/local/share/texmf ~/

  where `$SAGE_ROOT` is replaced with the location of your Sage installation.

- Use the environment variable `TEXINPUTS` to tell TeX to search the directory containing `sagetex.sty`; in the bash shell, you can do

      export TEXINPUTS=$SAGE_ROOT/local/share/texmf//:

  You should again replace `$SAGE_ROOT` with the location of your Sage installation.

The best method is likely the second; while that does require you to recopy the files every time you update your copy of Sage, it does not depend on your shell, so if you use, say, Emacs with AucTeX or some other editor environment, everything will still work since TeX's internal path-searching mechanisms can find `sagetex.sty`.

Note that along with `sagetex.sty`, this documentation, an example file, and other useful scripts are all located in the directory `$SAGE_ROOT/local/share/texmf`.

2.1 SageTₑX and TₑXLive

SageTₑX was included in TₑXLive, which seemed nice, except that the Python module and LATₑX package for SageTₑX need to be synchronized and the Python module in Sage was much easier to update than the LATₑX style file in TₑXLive. If you are so unlucky as to be using a version of TₑXLive that includes SageTₑX, I strongly recommend using SageTₑX only from what is included with Sage and ignoring what's included with TₑXLive.

2.2 The noversioncheck option

As of version 2.2.4, SageTₑX automatically checks to see if the versions of the style file and Python module match. This is intended to prevent strange version mismatch problems, but if you would like to use mismatched sources, you can—at your peril—give the `noversioncheck` option when you load the SageTₑX package. Don't be surprised if things don't work when you do this.

If you are considering using this option because the Sage script complained and exited, you really should just get the LATₑX and Python modules synchronized. Every copy of Sage since version 4.3.2 comes with a copy of `sagetex.sty` that is matched up to Sage's baked-in SageTₑX support, so you can always use that. See the SageTₑX section of the Sage tutorial.

2.3 Using TₑXShop

Starting with version 2.25, TₑXShop includes support for SageTₑX. If you move the file `sage.engine` from `~/Library/TeXShop/Engines/Inactive/Sage` to `~/Library/TeXShop/Engines` and put the line

```
%!TEX TS-program = sage
```

at the top of your document, then TₑXShop will automatically run Sage for you when compiling your document.

Note that you will need to make sure that LATₑX can find `sagetex.sty` using any of the methods above. You also might need to edit the `sage.engine` script to reflect the location of your Sage installation. TₑXShop includes further documentation in the `~/Library/TeXShop/Engines/Inactive/Sage` folder; be sure to check out the "About Sage" PDF there, which has good advice on setting up things so that typesetting SageTₑX-ified documents works automatically, and so that it continues to work when you upgrade your Sage installation.

2.4 Other scripts included with SageTₑX

SageTₑX includes several Python files which may be useful for working with "SageTₑX-ified" documents. At this point they have likely bitrotted but are, for now, included for archaeological purposes.

The `remote-sagetex.py` script allows you to use SageTₑX on a computer that doesn't have Sage installed; see section 5 for more information.

Also included are `makestatic.py` and `extractsagecode.py`, which are convenience scripts that you can use after you've written your document. See section 4.7 and section 4.8 for information on using those scripts. The file `sagetexparse.py` is a module used by both those scripts. These three files are independent of SageTEX. If you install from a spkg, these scripts can be found in `$SAGE_ROOT/local/share/texmf/`.

3 Usage

Let's begin with a rough description of how SageTEX works. Naturally the very first step is to put `\usepackage{sagetex}` in the preamble of your document. When you use macros from this package and run LaTeX on your file, along with the usual zoo of auxiliary files, a `.sage` file is written with the same basename as your document. This is a Sage source file that uses the Python module from this package and when you run Sage on that file, it will produce a `.sout` and a `.scmd` file. The `.sout` file contains LaTeX code that, when you run LaTeX on your source file again, will pull in all the results of Sage's computation.

The `sagecommandline` environment additionally logs the plain sage commands and output furthermore in a `.scmd` file.

All you really need to know is that to typeset your document, you need to run LaTeX, then run Sage, then run LaTeX again.

Also keep in mind that everything you send to Sage is done within one Sage session. This means you can define variables and reuse them throughout your LaTeX document; if you tell Sage that `foo` is 12, then anytime afterwards you can use `foo` in your Sage code and Sage will remember that it's 12—just like in a regular Sage session.

Now that you know that, let's describe what macros SageTEX provides and how to use them. If you are the sort of person who can't be bothered to read documentation until something goes wrong, you can also just look through the `example.tex` file included with this package.[1]

> **WARNING!** When you run LaTeX on a file named $\langle filename\rangle$`.tex`, the file $\langle filename\rangle$`.sagetex.sage` is created—and will be *automatically overwritten* if it already exists. If you keep Sage scripts in the same directory as your SageTEX-ified LaTeX documents, use a different file name!

> **WARNING!** Speaking of filenames, SageTEX really works best on files whose names don't have spaces or other "funny" characters in them. SageTEX *should* work on such files—and you should let us know if it doesn't—but it's safest to stick to files with alphanumeric characters and "safe" punctuation (i.e., nothing like `<`, `"`, `!`, `\`, or other characters that would confuse a shell).

---

[1] Then again, if you're such a person, you're probably not reading this, and are already fiddling with `example.tex`...

The final option On a similar note, SageTeX, like many LaTeX packages, accepts the `final` option. When passed this option, either directly in the `\usepackage` line, or from the `\documentclass` line, SageTeX will not write a `.sage` file. It will try to read in the `.sout` file so that the SageTeX macros can pull in their results. However, this will not allow you to have an independent Sage script with the same basename as your document, since to get the `.sout` file, you need the `.sage` file.

3.1 Inline Sage

sage | `\sage{⟨`*Sage code*`⟩}` | takes whatever Sage code you give it, runs Sage's `latex` function on it, and puts the result into your document.

For example, if you do `\sage{matrix([[1, 2], [3,4]])^2}`, then that macro will get replaced by

```
\left(\begin{array}{rr}
7 & 10 \\
15 & 22
\end{array}\right)
```

in your document—that LaTeX code is exactly exactly what you get from doing

```
latex(matrix([[1, 2], [3,4]])^2)
```

in Sage.

Note that since LaTeX will do macro expansion on whatever you give to `\sage`, you can mix LaTeX variables and Sage variables! If you have defined the Sage variable `foo` to be 12 (using, say, the `sageblock` environment), then you can do something like this:

```
The prime factorization of the current page number plus foo
is $\sage{factor(foo + \thepage)}$.
```

Here, I'll do just that right now: the prime factorization of the current page number plus 12 is $2 \cdot 3^2$. (Wrong answer? See footnote.[2]) The `\sage` command doesn't automatically use math mode for its output, so be sure to use dollar signs or a displayed math environment as appropriate.

\sagestr | `\sagestr{⟨`*Sage code*`⟩}` | is identical to `\sage`, but is does *not* run Sage's `latex` function on the code you give it; it simply runs the Sage code and pulls the result into your LaTeX file. This is useful for calling functions that return LaTeX code;

---

[2]Is the above factorization wrong? If the current page number plus 12 is one larger than the claimed factorization, another Sage/LaTeX cycle on this source file should fix it. Why? The first time you run LaTeX on this file, the sine graph isn't available, so the text where I've talked about the prime factorization is back one page. Then you run Sage, and it creates the sine graph and does the factorization. When you run LaTeX again, the sine graph pushes the text onto the next page, but it uses the Sage-computed value from the previous page. Meanwhile, the `.sage` file has been rewritten with the correct page number, so if you do another Sage/LaTeX cycle, you should get the correct value above. However, in some cases, even *that* doesn't work because of some kind of TeX weirdness in ending the one page a bit short and starting another.

see the example file distributed along with SageTeX for a demonstration of using this command to easily produce a table.

\percent   If you are doing modular arithmetic or string formatting and need a percent sign in a call to \sage (or \sageplot), you can use \percent. Using a bare percent sign won't work because LaTeX will think you're starting a comment and get confused; prefixing the percent sign with a backslash won't work because then "\%" will be written to the .sage file and Sage will get confused. The \percent macro makes everyone happy.

Note that using \percent inside the verbatim-like environments described in section 3.3 isn't necessary; a literal "%" inside such an environment will get written, uh, verbatim to the .sage file.

Arguments with side effects   Be careful when feeding \sage and \sagestr arguments that have side effects, since in some situations they can get evaluated more than once; see section 4.1.

3.2   Graphics and plotting

\sageplot $\boxed{\texttt{\textbackslash sageplot[}\langle\textit{ltx opts}\rangle\texttt{][}\langle\textit{fmt}\rangle\texttt{]\{}\langle\textit{graphics obj}\rangle\texttt{, }\langle\textit{keyword args}\rangle\texttt{\}}}$ plots the given Sage graphics object and runs an \includegraphics command to put it into your document. It does not have to actually be a plot of a function; it can be any Sage graphics object. The options are described in Table 1.

Option	Description
⟨ltx options⟩	Any text here is passed directly into the optional arguments (between the square brackets) of an \includegraphics command.
⟨fmt⟩	You can optionally specify a file extension here; Sage will then try to save the graphics object to a file with extension *fmt*. If not specified, SageTeX will save to EPS and PDF files; if saving to those formats does not work, SageTeX will save to a PNG file.
⟨graphics obj⟩	A Sage object on which you can call .save() with a graphics filename.
⟨keyword args⟩	Any keyword arguments you put here will all be put into the call to .save().

Table 1: Explanation of options for the \sageplot command.

This setup allows you to control both the Sage side of things, and the LaTeX side. For instance, the command

```
\sageplot[angle=30, width=5cm]{plot(sin(x), 0, pi), axes=False,
chocolate=True}
```

will run the following command in Sage:

```
sage: plot(sin(x), 0, pi).save(filename=autogen, axes=False,
chocolate=True)
```

Then, in your LaTeX file, the following command will be issued automatically:

```
\includegraphics[angle=30, width=5cm]{autogen}
```

You can specify a file format if you like. This must be the *second* optional argument, so you must use empty brackets if you're not passing anything to `\includegraphics`:

```
\sageplot[][png]{plot(sin(x), x, 0, pi)}
```

The filename is automatically generated, and unless you specify a format, both EPS and PDF files will be generated. This allows you to freely switch between using, say, a DVI viewer (many of which have support for automatic reloading, source specials and make the writing process easier) and creating PDFs for posting on the web or emailing to colleagues. SageTeX will fall back to creating a PNG file for any graphics object that cannot be saved as an EPS or PDF file; this is useful for three dimensional plot objects, which currently cannot be saved as EPS or PDF files.

If you ask for, say, a PNG file (or if one is automatically generated for you as described above), keep in mind that ordinary `latex` and DVI files have no support for PNG files; SageTeX detects this and will warn you that it cannot find a suitable file if using `latex`.[3] If you use `pdflatex`, there will be no problems because PDF files can include PNG graphics.

When SageTeX cannot find a graphics file, it inserts this into your document:

<div style="border:1px solid black; width:150px; height:150px; text-align:center; line-height:150px;">

??

</div>
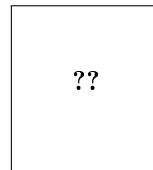
That's supposed to resemble the image-not-found graphics used by web browsers and use the traditional "**??**" that LaTeX uses to indicate missing references.

You needn't worry about the filenames; they are automatically generated and will be put into the directory `sage-plots-for-filename.tex`. You can safely delete that directory anytime; if SageTeX can't find the files, it will warn you to run Sage to regenerate them.

> **WARNING!** When you run Sage on your `.sage` file, all files in the `sage-plots-for-⟨filename⟩.tex` directory *will be deleted!* Do not put any files into that directory that you do not want to get automatically deleted.

---

[3] We use a typewriter font here to indicate the executables which produce DVI and PDF files, respectively, as opposed to "LaTeX" which refers to the entire typesetting system.

The epstopdf option One of the graphics-related options supported by SageTeX is `epstopdf`. This option causes SageTeX to use the `epstopdf` command to convert EPS files into PDF files. Like with the `imagemagick` option, it doesn't check to see if the `epstopdf` command exists or add options: it just runs the command. This option was motivated by a bug in the matplotlib PDF backend which caused it to create invalid PDFs. Ideally, this option should never be necessary; if you do need to use it, file a bug!

This option will eventually be removed, so do not use it.

3.2.1 3D plotting

Right now there is, to put it nicely, a bit of tension between the sort of graphics formats supported by `latex` and `pdflatex`, and the graphics formats supported by Sage's 3D plotting systems. LaTeX is happiest, and produces the best output, with EPS and PDF files, which are vector formats. Tachyon, Sage's 3D plotting system, produces bitmap formats like BMP and PNG.

SageTeX will automatically fall back to saving plot objects in PNG format if saving to EPS and PDF fails, so it should automatically work with 3D plot objects. However, since `latex` does not support PNGs, when using 3D plotting (and therefore a bitmap format like PNG), SageTeX will always issue a warning about incompatible graphics if you use `latex`, provided you've processed the `.sage` file and the PNG file exists. The only exception is if you're using the `imagemagick` option below.

The imagemagick option As a response to the above issue, the SageTeX package has an `imagemagick` option. If you specify this option in the preamble of your document with the usual "`\usepackage[imagemagick]{sagetex}`", then when you are compiling your document using `latex`, any `\sageplot` command which requests a non-default format will cause the SageTeX Python script to convert the resulting file to EPS using the Imagemagick `convert` utility. It does this by executing "`convert filename.EXT filename.eps`" in a subshell. It doesn't add any options, check to see if the `convert` command exists or belongs to Imagemagick—it just runs the command.

The resulting EPS files are not very high quality, but they will work. This option is not intended to produce good graphics, but to allow you to see your graphics when you use `latex` and DVI files while writing your document.

3.2.2 But that's not good enough!

The `\sageplot` command tries to be both flexible and easy to use, but if you are just not happy with it, you can always do things manually: inside a `sagesilent` environment (see the next section) you could do

```
your special commands
x = your graphics object
x.save(filename=myspecialfile.ext, options, etc)
```

and then, in your source file, do your own `\includegraphics` command. The SageTEX package gives you full access to Sage and Python and doesn't turn off anything in LATEX, so you can always do things manually.

3.3 Verbatim-like environments

The SageTEX package provides several environments for typesetting and executing blocks of Sage code.

sageblock   Any text between `\begin{sageblock}` and `\end{sageblock}` will be typeset into your file, and also written into the `.sage` file for execution. This means you can do something like this:

```
\begin{sageblock}
    var('x')
    f(x) = sin(x) - 1
    g(x) = log(x)
    h(x) = diff(f(x) * g(x), x)
\end{sageblock}
```

and then anytime later write in your source file

```
We have $h(2) = \sage{h(2)}$, where $h$ is the derivative of
the product of $f$ and $g$.
```

and the `\sage` call will get correctly replaced by $\sin(1)-1$. You can use any Sage or Python commands inside a `sageblock`; all the commands get sent directly to Sage.

sagesilent   This environment is like `sageblock`, but it does not typeset any of the code; it just writes it to the `.sage` file. This is useful if you have to do some setup in Sage that is not interesting or relevant to the document you are writing.

sageverbatim   This environment is the opposite of the one above: whatever you type will be typeset, but not written into the `.sage` file. This allows you to typeset psuedocode, code that will fail, or take too much time to execute, or whatever.

comment   Logically, we now need an environment that neither typesets nor executes your Sage code... but the `verbatim` package, which is always loaded when using SageTEX, provides such an environment: `comment`. Another way to do this is to put stuff between `\iffalse` and `\fi`.

\sagetexindent   There is one final bit to our verbatim-like environments: the indentation. The SageTEX package defines a length `\sagetexindent`, which controls how much the Sage code is indented when typeset. You can change this length however you like with `\setlength`: do `\setlength{\sagetexindent}{6ex}` or whatever.

sageexample   This environment allow you to include doctest-like snippets in your document that will be nicely typeset. For example,

```
\begin{sageexample}
  sage: 1+1
  2
  sage: factor(x^2 + 2*x + 1)
  (x + 1)^2
\end{sageexample}
```

in your document will be typeset with the Sage inputs in the usual fixed-width font, and the outputs will be typeset as if given to a `\sage` macro, as you see below:

```
sage: 1+1
```

$$2$$

```
sage: factor(x^2 + 2*x + 1)
```

$$(x + 1)^2$$

When typesetting the document, there is no test of the validity of the outputs (that is, typesetting with a typical LaTeX-Sage-LaTeX cycle does not do doctesting), but when using the `sageexample` environment, an extra file named "`myfile_doctest.sage`" is created with the contents of all those environments; it is formatted so that Sage can doctest that file. You should be able to doctest your document with "`sage -t myfile_doctest.sage`". (This does not always work; if this fails for you, please contact the sage-support group.)

If you would like to see both the original text input and the typeset output, you can issue `\renewcommand{\sageexampleincludetextoutput}{True}` in your document. You can do the same thing with "False" to later turn it off. In the above example, this would cause SageTeX to output both `(x + 1)^2` and $(x+1)^2$ in your typeset document.

Just as in doctests, multiline statements are acceptable. The only limitation is that triple-quoted strings delimited by `"""` cannot be used in a `sageexample` environment; instead, you can use triple-quoted strings delimited by `'''`.

See the `example.tex` file for many examples and more explanation.

The initial implementation of this environment is due to Nicolas M. Thiéry.

`sagecommandline`   This environment is similar to the sageexample environment in that it allow you to use SageTeX as a pretty-printing command line, or to include doctest-like snippets in your document. The difference is that the output is typeset as text, much like running Sage on the command line, using the `lstlisting` environment. In particular, this environment provides Python syntax highlighting and line numbers. For example,

```
\begin{sagecommandline}
  sage: t = 1+1
  sage: for i in range(101): # multiline input example
  ....:     t+=1
```

```
      sage: t
      103
      sage: factor(x^2 + 2*x + 1)
    \end{sagecommandline}
```

becomes

```
sage: t = 1+1                                                    1
sage: for i in range(101): # multiline input example            2
....:      t+=1                                                  3
sage: t                                                          4
103                                                             5
sage: factor( x^2 + 2*x + 1 )                                   6
(x + 1)^2                                                       7
```

You have a choice of either explicitly providing the Sage output or leaving it up to the computer to fill in the blanks. Above, the output for t was provided, but the output for the `factor()` command wasn't. Moreover, any Sage comment that starts with a "at" sign is escaped to LaTeX. In particular, you can use `\label` to mark line numbers in order to `\reference` and `\pageref`erence them as usual. See the example file to see this mechanism in action.

If you prefer to typeset the output in LaTeX, you can set

```
    \renewcommand{\sagecommandlinetextoutput}{False}
```

which produces

```
sage: var('a, b, c');                                           8
sage: ( a*x^2+b*x+c ).solve(x)                                  9
```

$$\left[x = -\frac{b + \sqrt{b^2 - 4\,ac}}{2\,a}, x = -\frac{b - \sqrt{b^2 - 4\,ac}}{2\,a} \right]$$

The Sage input and output is typeset using the `listings` package with the styles `SageInput` and `SageOutput`, respectively. If you don't like the defaults you can change them. It is recommended to derive from `DefaultSageInput` and `DefaultSageOutput`, for example

```
    \lstdefinestyle{SageInput}{style=DefaultSageInput,
                              basicstyle={\color{red}}}
    \lstdefinestyle{SageOutput}{style=DefaultSageOutput,
                              basicstyle={\color{green}}}
```

makes things overly colorful:

```
sage: pi.n(100)                                                 10
3.1415926535897932384626433833                                 11
```

3.4 Pausing SageTeX

Sometimes when you are writing a document, you may wish to temporarily turn off or pause SageTeX to concentrate more on your document than on the Sage computations, or to simply have your document typeset faster. You can do this with the following commands.

\sagetexpause Use these macros to "pause" and "unpause" SageTeX. After issuing this macro,
\sagetexunpause SageTeX will simply skip over the corresponding calculations. Anywhere a \sage macro is used while paused, you will simply see "(SageTeX is paused)", and anywhere a \sageplot macro is used, you will see:

SageTeX is paused; no graphic

Anything in the verbatim-like environments of section 3.3 will be typeset or not as usual, but none of the Sage code will be executed.

Obviously, you use \sagetexunpause to unpause SageTeX and return to the usual state of affairs. Both commands are idempotent; issuing them twice or more in a row is the same as issuing them once. This means you don't need to precisely match pause and unpause commands: once paused, SageTeX stays paused until it sees \sagetexunpause and vice versa.

4 Other notes

Here are some other notes on using SageTeX.

4.1 Using the sage macro inside align (and similar) environments

The `align`, `align*`, and some other "fancy" math environments in the `amsmath` package do some special processing—in particular, they evaluate everything inside twice. This means that if you use \sage or \sagestr inside such an environment, it will be evaluated twice, and its argument will be put into the generated `.sage` file twice—and if that argument has side effects, those side effects will be executed twice! Doing something such as popping an element from a list will actually pop *two* elements and typeset the second. The solution is to do any processing that has side effects before the `align` environment (in a `sagesilent` environment, say) and to give \sage or \sagestr an argument with no side effects.

Thanks to Bruno Le Floch for reporting this.

4.2 Using `sage` inside `caption`

If you want to use \sage inside the \caption for a figure, you'll need to prepend
it with \protect, because (in TeX parlance) the \sage macro is not "robust".
For example:

```
\caption{this figure illustrates why $2 + 2 = \protect\sage{2 + 2}$.}
```

4.3 Using SageTeX with TeXworks

On Linux and OS X, it's easy to get the TeXworks editor to work with SageTeX.
You will define a new "typesetting engine" following these directions. First, in
some convenient location, create a shell script with these contents:

```
#!/bin/sh
pdflatex $1.tex
sage $1.sagetex.sage
pdflatex $1.tex
```

Name it `sagetex-engine` or something similar. Then, in the Edit → Preferences
menu, click the + button in the Processing tools section. Give the new tool a
good name, like "SageTeX". In the Program space, browse to your script. Click
the + for the Arguments and type `$basename` in the resulting box.

Then, back in your document, you should be able to select SageTex in the
typesetting engine dropdown menu and then typeset.

(You can also replace "sage" above with the `run-sagetex-if-necessary`
script—see section 7.1.)

4.4 Using Beamer

The BEAMER package does not play nicely with verbatim-like environments unless
you ask it to. To use code block environments in a BEAMER presentation, do:

```
\begin{frame}[fragile]
\begin{sageblock}
# sage stuff
# more stuff \end{sageblock}
\end{frame}
```

For some reason, BEAMER inserts an extra line break at the end of the environment;
if you put the \end{sageblock} on the same line as the last line of your code,
it works properly. See section 12.9, "Verbatim and Fragile Text", in the BEAMER
manual. (Thanks to Franco Saliola for reporting this.)

BEAMER's overlays and \sageplot also need some help in order to work to-
gether, as discussed in this sage-support thread. If you want a plot to only appear
in a certain overlay, you might try something like this in your frame:

```
\begin{itemize}
\item item 1
\item item 2
\item \sageplot[height=4cm][png]{(plot_slope_field(2*x,(x,-4,4),
    (y,-4,4))+(x^2-2).plot(-2,2))}
\end{itemize}
```

but the plot will appear on all the overlays, instead of the third. The solution is to use the \visible macro:

```
\begin{itemize}
\item item 1
\item item 2
\item \visible<3->{\sageplot[height=4cm][png]{(plot_slope_field(2*x,(x,-4,4),
    (y,-4,4))+(x^2-2).plot(-2,2))}}
\end{itemize}
```

Then the plot will only appear on the third (and later) overlays. (Thanks to Robert Mařík for this solution.)

4.5 Using the `rccol` package

If you are trying to use the \sage macro inside a table when using the `rccol` package, you need to use an extra pair of braces or typesetting will fail. That is, you need to do something like this:

```
abc & {\sage{foo.n()}} & {\sage{bar}} \\
```

with each "\sage{}" enclosed in an extra {}. Thanks to Sette Diop for reporting this.

4.6 Plotting from Mathematica, Maple, etc.

Sage can use Mathematica, Maple, and friends and can tell them to do plotting, but since it cannot get those plots into a Sage graphics object, you cannot use \sageplot to use such graphics. You'll need to use the method described in "But that's not good enough!" (section 3.2.2) with some additional bits to get the directory right—otherwise your file will get saved to someplace in a hidden directory.

For Mathematica, you can do something like this inside a sagesilent or sageblock environment:

```
mathematica('myplot = commands to make your plot')
mathematica('Export["%s/graphicsfile.eps", myplot]' % os.getcwd())
```

then put \includegraphics[opts]{graphicsfile} in your file.

For Maple, you'll need something like

```
maple('plotsetup(ps, plotoutput='%s/graphicsfile.eps', \
    plotoptions='whatever');' % os.getcwd())
maple('plot(function, x=1..whatever);')
```

and then `\includegraphics` as necessary.

These interfaces, especially when plotting, can be finicky. The above commands are just meant to be a starting point.

4.7 Sending SageTEX files to others who don't use Sage

4.7.1 The 21st century way

The SageMathCloud (`cloud.sagemath.com`) service has full support for SageTEX, is free to use, and has many fantastic collaboration features. If you somehow want to collaborate on a SageTEX-enabled TEXdocument with a colleague, using SageMathCloud is one of the best and easiest ways to do it. You and your collaborators simply create an account, then make a project for your collaboration, and add all the collaborators to the project. Then you can all enjoy realtime collaborative editing, PDF previews, chat, and more.

4.7.2 The Nth century way, for $N < 21$

Using anything other than SageMathCloud, `git`, or similar tools these days seems pretty primitive. If emailing `.tex` files back and forth seems perfectly reasonable to you, or if you have to send a file to a journal, the easiest way is to simply include with your document the following files:

1. `sagetex.sty`

2. the generated `.sout` and `.scmd` files

3. the `sage-plots-for-⟨filename⟩.tex` directory and its contents

As long as `sagetex.sty` is available, your document can be typeset using any reasonable LATEX system. Since it is very common to include graphics files with a paper submission, this is a solution that should always work. (In particular, it will work with arXiv submissions.)

There is another option, and that is to use the `makestatic.py` script included with SageTEX. This script has been unmaintained for some time and likely won't work. If you want or need to use it and run into trouble, email the `sage-support` group and let us know.

Use of the script is quite simple. Copy it and `sagetexparse.py` to the directory with your document, and run

```
python makestatic.py inputfile [outputfile]
```

where `inputfile` is your document. (You can also set the executable bit of `makestatic.py` and use `./makestatic.py`.) This script needs the pyparsing module to be installed.[4] You may optionally specify `outputfile`; if you do so, the

---

[4]If you don't have pyparsing installed, you can simply copy the file `$SAGE_ROOT/local/lib/python/matplotlib/pyparsing.py` into your directory.

results will be written to that file. If the file exists, it won't be overwritten unless you also specify the `-o` switch.

You will need to run this after you've compiled your document and run Sage on the `.sage` file. The script reads in the `.sout` file and replaces all the calls to `\sage` and `\sageplot` with their plain LaTeX equivalent, and turns the `sageblock` and `sageverbatim` environments into `verbatim` environments. Any `sagesilent` environment is turned into a `comment` environment. Any `sagecommandline` environment is turned into a `lstlisting` environment, typesetting the relevant part of the `.scmd` file. The resulting document should compile to something identical, or very nearly so, to the original file.

One large limitation of this script is that it can't change anything while SageTEX is paused, since Sage doesn't compute anything for such parts of your document. It also doesn't check to see if pause and unpause commands are inside comments or verbatim environments. If you're going to use `makestatic.py`, just remove all pause/unpause statements.

The parsing that `makestatic.py` does is pretty good, but not perfect. Right now it doesn't support having a comma-separated list of packages, so you can't have `\usepackage{sagetex, foo}`. You need to have just `\usepackage{sagetex}`. (Along with package options; those are handled correctly.) If you find other parsing errors, please let me know.

4.8 Extracting the Sage code from a document

(This script has the same unmaintained-and-probably-doesn't-work status as `makestatic.py`; see above.)

This next script is probably not so useful, but having done the above, this was pretty easy. The `extractsagecode.py` script does the opposite of `makestatic.py`, in some sense: given a document, it extracts all the Sage code and removes all the LaTeX.

Its usage is the same as `makestatic.py`.

Note that the resulting file will almost certainly *not* be a runnable Sage script, since there might be LaTeX commands in it, the indentation may not be correct, and the plot options just get written verbatim to the file. Nevertheless, it might be useful if you just want to look at the Sage code in a file.

5 Using SageTEX without Sage installed

(This script has the same unmaintained-and-probably-doesn't-work status as `makestatic.py`; see above.)

You may want to edit and typeset a SageTEX-ified file on a computer that doesn't have Sage installed. How can you do that? We need to somehow run Sage on the `.sage` file. The included script `remote-sagetex.py` takes advantage of Sage's network transparency and will use a remote server to do all the computations. Anywhere in this manual where you are told to "run Sage", instead of actually running Sage, you can run

```
python remote-sagetex.py filename.sage
```

The script will ask you for a server, username, and password, then process all your code and write a `.sout` file and graphics files exactly as if you had used a local copy of Sage to process the `.sage` script. (With some minor limitations and differences; see below.)

One important point: *the script requires Python 2.6.* It will not work with earlier versions. (It will work with Python 3.0 or later with some trivial changes.)

You can provide the server, username and password with the command-line switches `--server`, `--username`, and `--password`, or you can put that information into a file and use the `--file` switch to specify that file. The format of the file must be like the following:

```
# hash mark at beginning of line marks a comment
server = "http://example.com:1234"
username = 'my_user_name'
password = 's33krit'
```

As you can see, it's really just like assigning a string to a variable in Python. You can use single or double quotes and use hash marks to start comments. You can't have comments on the same line as an assignment, though. You can omit any of those pieces of information information; the script will ask for anything it needs to know. Information provided as a command line switch takes precedence over anything found in the file.

You can keep this file separate from your LaTeX documents in a secure location; for example, on a USB thumb drive or in an automatically encrypted directory (like `~/Private` in Ubuntu). This makes it much harder to accidentally upload your private login information to the arXiv, put it on a website, send it to a colleague, or otherwise make your private information public.

5.1 Limitations of `remote-sagetex.py`

The `remote-sagetex.py` script has several limitations. It completely ignores the `epstopdf` and `imagemagick` flags. The `epstopdf` flag is not a big deal, since it was originally introduced to work around a matplotlib bug which has since been fixed. Not having `imagemagick` support means that you cannot automatically convert 3D graphics to eps format; using `pdflatex` to make PDFs works around this issue.

5.2 Other caveats

Right now, the "simple server API" that `remote-sagetex.py` uses is not terribly robust, and if you interrupt the script, it's possible to leave an idle session running on the server. If many idle sessions accumulate on the server, it can use up a lot of memory and cause the server to be slow, unresponsive, or maybe even crash. For now, I recommend that you only run the script manually. It's probably best to not configure your TeX editing environment to automatically run `remote-sagetex.py`

whenever you typeset your document, at least not without showing you the output or alerting you about errors.

6   Implementation

There are two pieces to this package: a LaTeX style file, and a Python module. They are mutually interdependent, so it makes sense to document them both here.

6.1   The style file

All macros and counters intended for use internal to this package begin with "ST@".

6.1.1   Initialization

Let's begin by loading some packages. The key bits of `sageblock` and friends are stol—um, adapted from the `verbatim` package manual. So grab the `verbatim` package. We also need the `fancyvrb` package for the `sageexample` environment

```
1 \RequirePackage{verbatim}
2 \RequirePackage{fancyvrb}
```

and `listings` for the `sagecommandline` environment.

```
 3 \RequirePackage{listings}
 4 \RequirePackage{color}
 5 \lstdefinelanguage{Sage}[]{Python}
 6   {morekeywords={False,sage,True},sensitive=true}
 7 \lstdefinelanguage{SageOutput}[]{}
 8   {morekeywords={False,True},sensitive=true}
 9 \lstdefinestyle{DefaultSageInputOutput}{
10   nolol,
11   identifierstyle=,
12   name=sagecommandline,
13   xleftmargin=5pt,
14   numbersep=5pt,
15   aboveskip=0pt,
16   belowskip=0pt,
17   breaklines=true,
18   numberstyle=\footnotesize,
19   numbers=right
20 }
21 \lstdefinestyle{DefaultSageInput}{
22   language=Sage,
23   style=DefaultSageInputOutput,
24   basicstyle={\ttfamily\bfseries},
25   commentstyle={\ttfamily\color{dgreencolor}},
26   keywordstyle={\ttfamily\color{dbluecolor}\bfseries},
27   stringstyle={\ttfamily\color{dgraycolor}\bfseries},
28 }
29 \lstdefinestyle{DefaultSageOutput}{
```

```
30    language=SageOutput,
31    style=DefaultSageInputOutput,
32    basicstyle={\ttfamily},
33    commentstyle={\ttfamily\color{dgreencolor}},
34    keywordstyle={\ttfamily\color{dbluecolor}},
35    stringstyle={\ttfamily\color{dgraycolor}},
36 }
37 \lstdefinestyle{SageInput}{
38    style=DefaultSageInput,
39 }
40 \lstdefinestyle{SageOutput}{
41    style=DefaultSageOutput,
42 }
43 \definecolor{dbluecolor}{rgb}{0.01,0.02,0.7}
44 \definecolor{dgreencolor}{rgb}{0.2,0.4,0.0}
45 \definecolor{dgraycolor}{rgb}{0.30,0.3,0.30}
```

Unsurprisingly, the \sageplot command works poorly without graphics support.

```
46 \RequirePackage{graphicx}
```

The makecmds package gives us a \provideenvironment which we need, and we use ifpdf and ifthen in \sageplot so we know what kind of files to look for. Since ifpdf doesn't detect running under XeTeX (which defaults to producing PDFs), we need ifxetex. Hopefully the ifpdf package will get support for this and we can drop ifxetex. We also work around ancient TeX distributions that don't have ifxetex and assume that they don't have XeTeX.

```
47 \RequirePackage{makecmds}
48 \RequirePackage{ifpdf}
49 \RequirePackage{ifthen}
50 \IfFileExists{ifxetex.sty}{
51    \RequirePackage{ifxetex}
52 }{
53    \newboolean{xetex}
54    \setboolean{xetex}{false}}
```

Next set up the counters, default indent, and flags.

```
55 \newcounter{ST@inline}
56 \newcounter{ST@plot}
57 \newcounter{ST@cmdline}
58 \setcounter{ST@inline}{0}
59 \setcounter{ST@plot}{0}
60 \setcounter{ST@cmdline}{0}
61 \newlength{\sagetexindent}
62 \setlength{\sagetexindent}{5ex}
63 \newif\ifST@paused
64 \ST@pausedfalse
```

Set up the file stuff, which will get run at the beginning of the document, after we know what's happening with the final option. First, we open the .sage file:

```
65 \AtBeginDocument{\@ifundefined{ST@final}{%
66 \newwrite\ST@sf%
```

67 `\immediate\openout\ST@sf=\jobname.sagetex.sage%`

`\ST@wsf` We will write a lot of stuff to that file, so make a convenient abbreviation, then use it to put the initial commands into the `.sage` file. The hash mark below gets doubled when written to the file, for some obscure reason related to parameter expansion. It's valid Python, though, so I haven't bothered figuring out how to get a single hash. We are assuming that the extension is `.tex`; see the `initplot` documentation on page 34 for discussion of file extensions. (There is now the `currfile` package (`http://www.ctan.org/pkg/currfile/`) which can figure out file extensions, apparently.) The "(`\jobname.sagetex.sage`)" business is there because the comment below will get pulled into the autogenerated `.py` file (second order autogeneration!) and I'd like to reduce possible confusion if someone is looking around in those files. Finally, we check for version mismatch and bail if the .py and .sty versions don't match and the user hasn't disabled checking. Note that we use `^^J` and not `^^J%` when we need indented lines. Also, `sagetex.py` now includes a `version` variable which eliminates all the irritating string munging below, and later we can remove this stuff and just use `sagetex.version`.

68 `\newcommand{\ST@wsf}[1]{\immediate\write\ST@sf{#1}}%`
69 `\ST@wsf{%`
70 `# -*- encoding: utf-8 -*-^^J%`
71 `# This file (\jobname.sagetex.sage) was *autogenerated* from \jobname.tex with`
72 `sagetex.sty version \ST@ver.^^J%`
73 `import sagetex^^J%`
74 `_st_ = sagetex.SageTeXProcessor('\jobname', version='\ST@ver', version_check=\ST@versioncheck)}`

On the other hand, if the `ST@final` flag is set, don't bother with any of the file stuff, and make `\ST@wsf` a no-op.

75 `{\newcommand{\ST@wsf}[1]{\relax}}}`

`\ST@dodfsetup` The `sageexample` environment writes stuff out to a different file formatted so that one can run doctests on it. We define a macro that only sets this up if necessary.

76 `\newcommand{\ST@dodfsetup}{%`
77 `\@ifundefined{ST@diddfsetup}{%`
78 `\newwrite\ST@df%`
79 `\immediate\openout\ST@df=\jobname_doctest.sage%`
80 `\immediate\write\ST@df{r"""^^J%`
81 `This file was *autogenerated* from \jobname.tex with sagetex.sty^^J%`
82 `version \ST@ver. It contains the contents of all the^^J%`
83 `sageexample environments from \jobname.tex. You should be able to^^J%`
84 `doctest this file with "sage -t \jobname_doctest.sage".^^J%`
85 `^^J%`
86 `It is always safe to delete this file; it is not used in typesetting your^^J%`
87 `document.^^J%`
88 `\AtEndDocument{\immediate\write\ST@df{"""}}%`
89 `\gdef\ST@diddfsetup{x}}%`
90 `{\relax}}`

`\ST@wdf` This is the compansion to `\ST@wsf`; it writes to the doctest file, assuming that

is has been set up. We ignore the `final` option here since nothing in this file is relevant to typesetting the document.

```
91 \newcommand{\ST@wdf}[1]{\immediate\write\ST@df{#1}}
```

Now we declare our options, which mostly just set flags that we check at the beginning of the document, and when running the `.sage` file.

The `final` option controls whether or not we write the `.sage` file; the `imagemagick` and `epstopdf` options both want to write something to that same file. So we put off all the actual file stuff until the beginning of the document—by that time, we'll have processed the `final` option (or not) and can check the `\ST@final` flag to see what to do. (We must do this because we can't specify code that runs if an option *isn't* defined.)

For `final`, we set a flag for other guys to check, and if there's no `.sout` file, we warn the user that something fishy is going on.

```
92 \DeclareOption{final}{%
93   \newcommand{\ST@final}{x}%
94   \IfFileExists{\jobname.sagetex.sout}{}{\AtEndDocument{\PackageWarningNoLine{sagetex}%
95   {'final' option provided, but \jobname.sagetex.sout^^Jdoesn't exist! No Sage
96     input will appear in your document. Remove the 'final'^^Joption and
97     rerun LaTeX on your document}}}}
```

For `imagemagick`, we set two flags: one for LaTeX and one for Sage. It's important that we set `ST@useimagmagick` *before* the beginning of the document, so that the graphics commands can check that. We do wait until the beginning of the document to do file writing stuff.

```
98 \DeclareOption{imagemagick}{%
99   \newcommand{\ST@useimagemagick}{x}%
100  \AtBeginDocument{%
101  \@ifundefined{ST@final}{%
102    \ST@wsf{_st_.useimagemagick = True}}{}}}
```

For `epstopdf`, we just set a flag for Sage.

```
103 \DeclareOption{epstopdf}{%
104 \AtBeginDocument{%
105 \@ifundefined{ST@final}{%
106  \ST@wsf{_st_.useepstopdf = True}}{}}}
```

By default, we check to see if the .py and .sty file versions match. But we let the user disable this.

```
107 \newcommand{\ST@versioncheck}{True}
108 \DeclareOption{noversioncheck}{%
109  \renewcommand{\ST@versioncheck}{False}}
110 \ProcessOptions\relax
```

The `\relax` is a little incantation suggested by the "LaTeX 2$_\varepsilon$ for class and package writers" manual, section 4.7.

Pull in the `.sout` file if it exists, or do nothing if it doesn't. I suppose we could do this inside an `AtBeginDocument` but I don't see any particular reason to do that. It will work whenever we load it. If the `.sout` file isn't found, print the

usual TeX-style message. This allows programs (`Latexmk`, for example) that read the `.log` file or terminal output to detect the need for another typesetting run to do so. If the "`No file foo.sout`" line doesn't work for some software package, please let me know and I can change it to use `PackageInfo` or whatever.

```
111 \InputIfFileExists{\jobname.sagetex.sout}{}
112 {\typeout{No file \jobname.sagetex.sout.}}
```

The user might load the `hyperref` package after this one (indeed, the `hyperref` documentation insists that it be loaded last) or not at all—so when we hit the beginning of the document, provide a dummy `NoHyper` environment if one hasn't been defined by the `hyperref` package. We need this for the `\sage` macro below.

```
113 \AtBeginDocument{\provideenvironment{NoHyper}{}{}}
```

6.1.2 The `\sage` and `\sagestr` macros

`\ST@sage` This macro combines `\ref`, `\label`, and Sage all at once. First, we use Sage to get a LaTeX representation of whatever you give this function. The Sage script writes a `\newlabel` line into the `.sout` file, and we read the output using the `\ref` command. Usually, `\ref` pulls in a section or theorem number, but it will pull in arbitrary text just as well.

The first thing it does it write its argument into the `.sage` file, along with a counter so we can produce a unique label. We wrap a try/except around the function call so that we can provide a more helpful error message in case something goes wrong. (In particular, we can tell the user which line of the `.tex` file contains the offending code.) Note the difference between `^^J` and `^^J%`: the newline immediately after the former puts a space into the output, and the percent sign in the latter supresses this.

```
114 \newcommand{\ST@sage}[1]{\ST@wsf{%
115 try:^^J
116   _st_.current_tex_line = \the\inputlineno^^J
117   _st_.inline(\theST@inline, #1)^^J%
118 except:^^J
119   _st_.goboom(\the\inputlineno)}%
```

The `inline` function of the Python module is documented on page 35. Back in LaTeX-land: if paused, say so.

```
120 \ifST@paused
121   \mbox{(Sage\TeX{} is paused)}%
```

Otherwise... our use of `\newlabel` and `\ref` seems awfully clever until you load the `hyperref` package, which gleefully tries to hyperlink the hell out of everything. This is great until it hits one of our special `\newlabel`s and gets deeply confused. Fortunately the `hyperref` folks are willing to accomodate people like us, and give us a `NoHyper` environment.

```
122 \else
123   \begin{NoHyper}\ref{@sageinline\theST@inline}\end{NoHyper}%
```

Now check if the label has already been defined. (The internal implementation of labels in LaTeX involves defining a macro called "r@@labelname".) If it hasn't, we

set a flag so that we can tell the user to run Sage on the `.sage` file at the end of the run.

```
124    \@ifundefined{r@@sageinline\theST@inline}{\gdef\ST@rerun{x}}{}%
125 \fi
```

In any case, the last thing to do is step the counter.

```
126 \stepcounter{ST@inline}}
```

\sage This is the user-visible macro; it runs Sage's `latex()` on its argument.

```
127 \newcommand{\sage}[1]{\ST@sage{latex(#1)}}
```

\sagestr Like above, but doesn't run `latex()` on its argument.

```
128 \newcommand{\sagestr}[1]{\ST@sage{#1}}
```

\percent A macro that inserts a percent sign. This is more-or-less stolen from the Docstrip manual; there they change the catcode inside a group and use `gdef`, but here we try to be more LATEXy and use `\newcommand`.

```
129 \catcode`\%=12
130 \newcommand{\percent}{%}
131 \catcode`\%=14
```

6.1.3   The \sageplot macro and friends

Plotting is rather more complicated, and requires several helper macros that accompany \sageplot.

\ST@plotdir A little abbreviation for the plot directory. We don't use \graphicspath because it's apparently slow—also, since we know right where our plots are going, no need to have LATEX looking for them.

```
132 \newcommand{\ST@plotdir}{sage-plots-for-\jobname.tex}
```

\ST@missingfilebox The code that makes the "file not found" box. This shows up in a couple places below, so let's just define it once.

```
133 \newcommand{\ST@missingfilebox}{\framebox[2cm]{\rule[-1cm]{0cm}{2cm}\textbf{??}}}
```

\sageplot This function is similar to \sage. The neat thing that we take advantage of is that commas aren't special for arguments to LATEX commands, so it's easy to capture a bunch of keyword arguments that get passed right into a Python function.

This macro has two optional arguments, which can't be defined using LATEX's \newcommand; we use Scott Pakin's brilliant newcommand package to create this macro; the options I fed to his script were similar to this:

```
          MACRO sageplot OPT[#1={width}] OPT[#2={notprovided}] #3
```

Observe that we are using a Python script to write LATEX code which writes Python code which writes LATEX code. Crazy!

Here's the wrapper command which does whatever magic we need to get two optional arguments.

```
134 \newcommand{\sageplot}[1][]{%
135    \@ifnextchar[{\ST@sageplot[#1]}{\ST@sageplot[#1][notprovided]}}
```
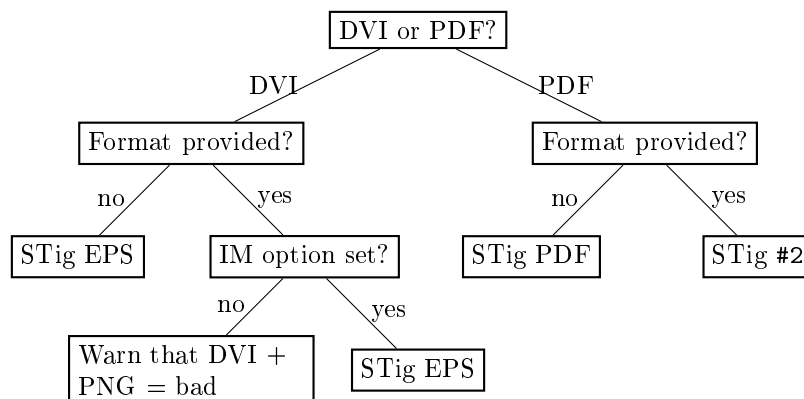
24

DVI or PDF?

DVI — Format provided?

PDF — Format provided?

no — STig EPS

yes — IM option set?

no — STig PDF

yes — STig #2

no — Warn that DVI + PNG = bad

yes — STig EPS

Figure 1: The logic tree that \sageplot uses to decide whether to run \includegraphics or to yell at the user. "Format" is the #2 argument to \sageplot, "STig ext" means a call to \ST@inclgrfx with "ext" as the second argument, and "IM" is Imagemagick.

The first optional argument #1 will get shoved right into the optional argument for \includegraphics, so the user has easy control over the LaTeX aspects of the plotting. (Perhaps a future version of SageTEX will allow the user to specify in the package options a set of default options to be used throughout.) The second optional argument #2 is the file format and allows us to tell what files to look for. It defaults to "notprovided", which tells the Python module to create EPS and PDF files. Everything in #3 gets put into the Python function call, so the user can put in keyword arguments there which get interpreted correctly by Python.

\ST@sageplot Let's see the real code here. We write a couple lines to the .sage file, including a counter, input line number, and all of the mandatory argument; all this is wrapped in another try/except.

```
136 \def\ST@sageplot[#1][#2]#3{\ST@wsf{try:^^J
137  _st_.current_tex_line = \the\inputlineno^^J
138  _st_.plot(\theST@plot, format='#2', _p_=#3)^^Jexcept:^^J
139  _st_.goboom(\the\inputlineno)}%
```

The Python plot function is documented on page 40.

Now we include the appropriate graphics file. Because the user might be producing DVI or PDF files, and have supplied a file format or not, and so on, the logic we follow is a bit complicated. Figure 1 shows what we do; for completeness—and because I think drawing trees with TikZ is really cool—we show what \ST@inclgrfx does in Figure 2. This entire complicated business is intended to avoid doing an \includegraphics command on a file that doesn't exist, and to issue warnings appropriate to the situation.

If we are creating a PDF, we check to see if the user asked for a different format, and use that if necessary:

```
140 \ifthenelse{\boolean{pdf} \or \boolean{xetex}}{
```

```
141    \ifthenelse{\equal{#2}{notprovided}}%
142      {\ST@inclgrfx{#1}{pdf}}%
143      {\ST@inclgrfx{#1}{#2}}}
```

Otherwise, we are creating a DVI file, which only supports EPS. If the user provided a format anyway, don't include the file (since it won't work) and warn the user about this. (Unless the file doesn't exist, in which case we do the same thing that `\ST@inclgrfx` does.)

```
144 { \ifthenelse{\equal{#2}{notprovided}}%
145      {\ST@inclgrfx{#1}{eps}}%
```

If a format is provided, we check to see if we're using the imagemagick option. If not, we're going to issue some sort of warning, depending on whether the file exists yet or not.

```
146      {\@ifundefined{ST@useimagemagick}%
147        {\IfFileExists{\ST@plotdir/plot-\theST@plot.#2}%
148          {\ST@missingfilebox%
149           \PackageWarning{sagetex}{Graphics file
150           \ST@plotdir/plot-\theST@plot.#2\space on page \thepage\space
151           cannot be used with DVI output. Use pdflatex or create an EPS
152           file. Plot command is}}%
153          {\ST@missingfilebox%
154           \PackageWarning{sagetex}{Graphics file
155           \ST@plotdir/plot-\theST@plot.#2\space on page \thepage\space
156           does not exist. Plot command is}%
157           \gdef\ST@rerun{x}}}%
```

Otherwise, we are using Imagemagick, so try to include an EPS file anyway.

```
158      {\ST@inclgrfx{#1}{eps}}}}
```

Step the counter and we're done with the usual work.

```
159 \stepcounter{ST@plot}}
```

\ST@inclgrfx   This command includes the requested graphics file (#2 is the extension) with the requested options (#1) if the file exists. Note that it just needs to know the extension, since we use a counter for the filename. If we are paused, it just puts in a little box saying so.

```
160 \newcommand{\ST@inclgrfx}[2]{\ifST@paused
161   \fbox{\rule[-1cm]{0cm}{2cm}Sage\TeX{} is paused; no graphic}
162 \else
163   \IfFileExists{\ST@plotdir/plot-\theST@plot.#2}%
164     {\includegraphics[#1]{\ST@plotdir/plot-\theST@plot.#2}}%
```

If the file doesn't exist, we try one more thing before giving up: the Python module will automatically fall back to saving as a PNG file if saving as an EPS or PDF file fails. So if making a PDF, we look for a PNG file.

If the file isn't there, we insert a little box to indicate it wasn't found, issue a warning that we didn't find a graphics file, then set a flag that, at the end of the run, tells the user to run Sage again.

```
165      {\IfFileExists{\ST@plotdir/plot-\theST@plot.png}%
```
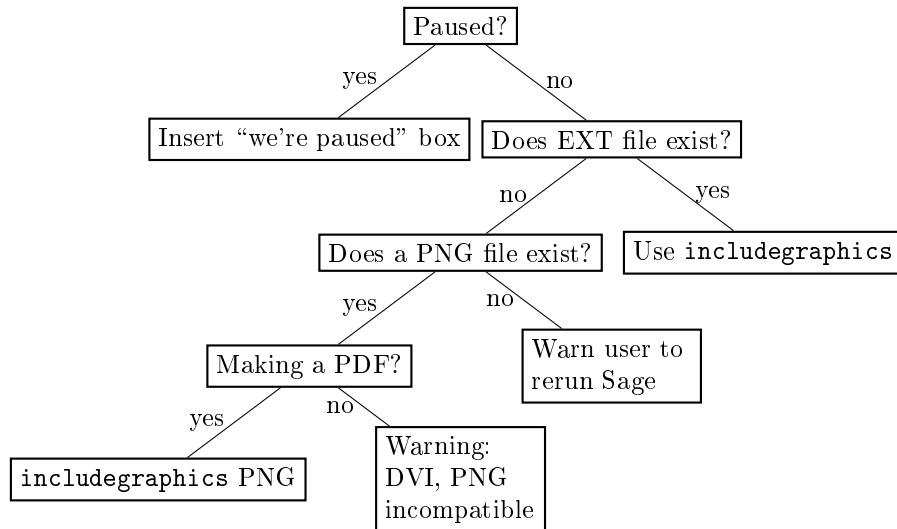
Figure 2: The logic used by the `\ST@inclgrfx` command.

```
166        {\ifpdf
167            \ST@inclgrfx{#1}{png}
168          \else
169            \PackageWarning{sagetex}{Graphics file
170            \ST@plotdir/plot-\theST@plot.png on page \thepage\space not
171             supported; try using pdflatex. Plot command is}%
172          \fi}%
173        {\ST@missingfilebox%
174        \PackageWarning{sagetex}{Graphics file
175        \ST@plotdir/plot-\theST@plot.#2\space on page \thepage\space does not
176        exist. Plot command is}%
177        \gdef\ST@rerun{x}}}
178 \fi}
```

Figure 2 makes this a bit clearer.

6.1.4   Verbatim-like environments

`\ST@beginsfbl`   This is "begin `.sage` file block", an internal-use abbreviation that sets things up when we start writing a chunk of Sage code to the `.sage` file. It begins with some TeX magic that fixes spacing, then puts the start of a try/except block in the `.sage` file—this not only allows the user to indent code without Sage/Python complaining about indentation, but lets us tell the user where things went wrong. The `blockbegin` and `blockend` functions are documented on page 36. The last bit is some magic from the `verbatim` package manual that makes LaTeX respect line breaks.

```
179 \newcommand{\ST@beginsfbl}{%
```

27

```
180   \@bsphack\ST@wsf{%
181 _st_.current_tex_line = \the\inputlineno^^J%
182 _st_.blockbegin()^^Jtry:}%
183   \let\do\@makeother\dospecials\catcode`\^^M\active}
```

\ST@endsfbl The companion to \ST@beginsfbl.

```
184 \newcommand{\ST@endsfbl}{%
185 \ST@wsf{except:^^J
186 _st_.goboom(\the\inputlineno)^^J_st_.blockend()}}
```

Now let's define the "verbatim-like" environments. There are four possibilities, corresponding to the two independent choices of typesetting the code or not, and writing to the .sage file or not.

sageblock This environment does both: it typesets your code and puts it into the .sage file for execution by Sage.

```
187 \newenvironment{sageblock}{\ST@beginsfbl%
```

The space between \ST@wsf{ and \the is crucial! It, along with the "try:", is what allows the user to indent code if they like. This line sends stuff to the .sage file.

```
188 \def\verbatim@processline{\ST@wsf{ \the\verbatim@line}%
```

Next, we typeset your code and start the verbatim environment.

```
189 \hspace{\sagetexindent}\the\verbatim@line\par}%
190 \verbatim}%
```

At the end of the environment, we put a chunk into the .sage file and stop the verbatim environment.

```
191 {\ST@endsfbl\endverbatim}
```

sagesilent This is from the verbatim package manual. It's just like the above, except we don't typeset anything.

```
192 \newenvironment{sagesilent}{\ST@beginsfbl%
193 \def\verbatim@processline{\ST@wsf{ \the\verbatim@line}}%
194 \verbatim@start}%
195 {\ST@endsfbl\@esphack}
```

sageverbatim The opposite of sagesilent. This is exactly the same as the verbatim environment, except that we include some indentation to be consistent with other typeset Sage code.

```
196 \newenvironment{sageverbatim}{%
197 \def\verbatim@processline{\hspace{\sagetexindent}\the\verbatim@line\par}%
198 \verbatim}%
199 {\endverbatim}
```

Logically, we now need an environment which neither typesets *nor* writes code to the .sage file. The verbatim package's comment environment does that.

sageexample Finally, we have an environment which is mostly-but-not-entirely verbatim; this is the example environment, which takes input like Sage doctests, and prints out the commands verbatim but nicely typesets the output of those commands. This and the corresponding Python function are originally due to Nicolas M. Thiéry.

```
200 \newcommand{\sageexampleincludetextoutput}{False}
201 \newenvironment{sageexample}{%
202   \ST@wsf{%
203 try:^^J
204 _st_.current_tex_line = \the\inputlineno^^J
205 _st_.doctest(\theST@inline, r"""}%
206   \ST@dodfsetup%
207   \ST@wdf{Sage example, line \the\inputlineno::^^J}%
208   \begingroup%
209   \@bsphack%
210   \let\do\@makeother\dospecials%
211   \catcode'\^^M\active%
212   \def\verbatim@processline{%
213     \ST@wsf{\the\verbatim@line}%
214     \ST@wdf{\the\verbatim@line}%
215   }%
216   \verbatim@start%
217 }
218 {
219   \@esphack%
220   \endgroup%
221   \ST@wsf{%
222     """, globals(), locals(), \sageexampleincludetextoutput)^^Jexcept:^^J
223     _st_.goboom(\the\inputlineno)}%
224   \ifST@paused%
225     \mbox{(Sage\TeX{} is paused)}%
226   \else%
227     \begin{NoHyper}\ref{@sageinline\theST@inline}\end{NoHyper}%
228     \@ifundefined{r@@sageinline\theST@inline}{\gdef\ST@rerun{x}}{}%
229   \fi%
230   \ST@wdf{}%
231   \stepcounter{ST@inline}}
```

sagecommandline This environment is similar to the sageexample environment, but typesets the Sage output as text with Python syntax highlighting.

```
232 \newcommand{\sagecommandlinetextoutput}{True}
233 \newlength{\sagecommandlineskip}
234 \setlength{\sagecommandlineskip}{8pt}
235 \newenvironment{sagecommandline}{%
236   \ST@wsf{%
237 try:^^J
238 _st_.current_tex_line = \the\inputlineno^^J
239 _st_.commandline(\theST@cmdline, r"""}%
240   \ST@dodfsetup%
241   \ST@wdf{Sage commandline, line \the\inputlineno::^^J}%
```

```
242    \begingroup%
243    \@bsphack%
244    \let\do\@makeother\dospecials%
245    \catcode`\^^M\active%
246    \def\verbatim@processline{%
247      \ST@wsf{\the\verbatim@line}%
248      \ST@wdf{\the\verbatim@line}%
249    }%
250    \verbatim@start%
251 }
252 {
253    \@esphack%
254    \endgroup%
255    \ST@wsf{%
256      """, globals(), locals(), \sagecommandlinetextoutput)^^Jexcept:^^J
257      _st_.goboom(\the\inputlineno)}%
258    \ifST@paused%
259      \mbox{(Sage\TeX{} is paused)}%
260    \else%
261      \begin{NoHyper}\ref{@sagecmdline\theST@cmdline}\end{NoHyper}%
262      \@ifundefined{r@@sagecmdline\theST@cmdline}{\gdef\ST@rerun{x}}{}%
263    \fi%
264    \ST@wdf{}%
265    \stepcounter{ST@cmdline}}
```

6.1.5 Pausing SageTeX

How can one have Sage to stop processing SageTeX output for a little while, and then start again? At first I thought I would need some sort of "goto" statement in Python, but later realized that there's a dead simple solution: write triple quotes to the .sage file to comment out the code. Okay, so this isn't *really* commenting out the code; PEP 8 says block comments should use "#" and Sage will read in the "commented-out" code as a string literal. For the purposes of SageTeX, I think this is a good decision, though, since (1) the pausing mechanism is orthogonal to everything else, which makes it easier to not screw up other code, and (2) it will always work.

This illustrates what I really like about SageTeX: it mixes LaTeX and Sage/Python, and often what is difficult or impossible in one system is trivial in the other.

sagetexpause   This macro pauses SageTeX by effectively commenting out code in the .sage file. When running the corresponding .sage file, Sage will skip over any commands issued while SageTeX is paused.

```
266 \newcommand{\sagetexpause}{\ifST@paused\relax\else
267 \ST@wsf{print('SageTeX paused on \jobname.tex line \the\inputlineno')^^J"""}
268 \ST@pausedtrue
269 \fi}
```

sagetexunpause   This is the obvious companion to \sagetexpause.

```
270 \newcommand{\sagetexunpause}{\ifST@paused
271 \ST@wsf{"""^^Jprint('SageTeX unpaused on \jobname.tex line \the\inputlineno')}
272 \ST@pausedfalse
273 \fi}
```

6.1.6   End-of-document cleanup

We tell the Sage script to write some information to the .sout file, then check to see if ST@rerun ever got defined. If not, all the inline formulas and plots worked, so do nothing. We check to see if we're paused first, so that we can finish the triple-quoted string in the .sage file.

```
274 \AtEndDocument{\ifST@paused
275 \ST@wsf{"""^^Jprint('SageTeX unpaused at end of \jobname.tex')}
276 \fi
277 \ST@wsf{_st_.endofdoc()}%
278 \@ifundefined{ST@rerun}{}%
```

Otherwise, we issue a warning to tell the user to run Sage on the .sage file. Part of the reason we do this is that, by using \ref to pull in the inlines, LaTeX will complain about undefined references if you haven't run the Sage script—and for many LaTeX users, myself included, the warning "there were undefined references" is a signal to run LaTeX again. But to fix these particular undefined references, you need to run *Sage*. We also suppress file-not-found errors for graphics files, and need to tell the user what to do about that.

At any rate, we tell the user to run Sage if it's necessary.

```
279 {\typeout{****************************************************************************}
280 \PackageWarningNoLine{sagetex}{there were undefined Sage formulas and/or
281 plots.^^JRun Sage on \jobname.sagetex.sage, and then run LaTeX on \jobname.tex
282 again}}
283 \typeout{****************************************************************************}}
```

6.2   The Python module

The style file writes things to the .sage file and reads them from the .sout file. The Python module provides functions that help produce the .sout file from the .sage file.

A note on Python and Docstrip   There is one tiny potential source of confusion when documenting Python code with Docstrip: the percent sign. If you have a long line of Python code which includes a percent sign for string formatting and you break the line with a backslash and begin the next line with a percent sign, that line *will not* be written to the output file. This is only a problem if you *begin* the line with a (single) percent sign; there are no troubles otherwise.

On to the code. Munge the version string (which we get from sagetex.dtx) to extract what we want, then import what we need:

31

```
284 pyversion = ' '.join(__version__.strip('[').split()[0:2])
285 from sage.misc.latex import latex
286 from sage.repl.preparse import preparse
287 from six import PY3
288 import sys
289 import os
290 import os.path
291 import hashlib
292 import traceback
293 import subprocess
294 import shutil
295 import re
296 from collections import defaultdict
```

Define an exception class for version mismatches. I suppose I could just use `ValueError`, but this is easy enough:

```
297 class VersionError(Exception):
298     pass
```

Sometimes our macros that write things to the `.sout` file get evaluated twice, most commonly in the "fancy" AMS environments such as `align` and `multline`. So we need to keep track of the counters we've seen so we don't write labels to the `.sout` file more than once. We have more than one kind of label, so a dictionary is the natural way to store the counters we've seen for each kind of label. For convenience let's make a dictionary subclass for which (1) values default to -1, and (2) there's an `increment(key)` function that just increments the value corresponding to the key.

```
299 class MyDict(defaultdict):
300     def __init__(self, *args, **kwargs):
301         defaultdict.__init__(self, *args, **kwargs)
302         self.default_factory = lambda: -1
303
304     def increment(self, key):
305         self[key] = self[key] + 1
```

Here's a helper function used by `doctest`; it works ike `j.join(xs)`, but ensures exactly one copy of `j` between the strings in `xs`. Intended for `j` to be a single character, particularly newline so that you can join things with no extra blank lines.

```
306 def joinone(j, xs_):
307     if len(xs_) >= 2:
308         xs = ([xs_[0].rstrip(j)] +
309               [x.strip(j) for x in xs_[1:-1]] +
310               [xs_[-1].lstrip(j)])
311     else:
312         xs = xs_
313     return j.join(xs)
```

Another helper, used by `commandline` (and maybe, someday, `doctest`?). In each line, we look for a possibly empty sequence of spaces followed by a non-whitespace

character, so we can distinguish between whitespace-only lines (which we ignore) and lines that have no leading spaces.

One tiny possible problem: you might have a line of only, say, two spaces, but perhaps the "real" lines all start with at least three spaces. Then you would, for that line, do `line[2:]`. That seems like it might raise an error, since the line only has indices 0 and 1, but Python's indexing handles this perfectly: in that case, `line[2:]` will be the empty string, which is fine for our purposes.

```
314 def strip_common_leading_spaces(s):
315     lines = s.splitlines()
316     lead = min(m.end() for m in
317                   [re.match(' *\S', line) for line in lines]
318                   if m is not None) - 1
319     return '\n'.join(line[lead:] for line in lines)
```

6.2.1 The SageTeXProcessor class

The star of the show, as it were. We define a `SageTeXProcessor` class so that it's a bit easier to carry around internal state. We used to just have some global variables and a bunch of functions, but this seems a bit nicer and easier.

```
320 class SageTeXProcessor():
```

If the original `.tex` file has spaces in its name, the `\jobname` we get is surrounded by double quotes, so fix that. Technically, it is possible to have double quotes in a legitimate filename, but dealing with that sort of quoting is unpleasant. And yes, we're ignoring the possibility of tabs and other whitespace in the filename. Patches for handling pathological filenames welcome.

```
321   def __init__(self, jobname, version=None, version_check=True):
322     if version != pyversion:
323       errstr = """versions of .sty and .py files do not match.
324 {0}.sagetex.sage was generated by sagetex.sty version "{1}", but
325 is being processed by sagetex.py version "{2}".
326 Please make sure that TeX is using the sagetex.sty
327 from your current version of Sage; see
328 http://doc.sagemath.org/html/en/tutorial/sagetex.html.""".format(jobname,
329   version, pyversion)
330       if version_check:
331         raise VersionError(errstr)
332       else:
333         print('**** WARNING! Skipping version check for .sty and .py files, and')
334         print(errstr)
335     if ' ' in jobname:
336       jobname = jobname.strip('"')
337     self.progress('Processing Sage code for {0}.tex...'.format(jobname))
338     self.didinitplot = False
339     self.useimagemagick = False
340     self.useepstopdf = False
341     self.plotdir = 'sage-plots-for-' + jobname + '.tex'
342     self.filename = jobname
```

```
343    self.name = os.path.splitext(jobname)[0]
344    autogenstr = """% This file was *autogenerated* from {0}.sagetex.sage with
345  % sagetex.py version {1}\n""".format(self.name, version)
```

Don't remove the space before the percent sign above!

LaTeX environments such as `align` evaluate their arguments twice after doing `\savecounters@`, so if you do `\sage` inside such an environment, it will result in two labels with the same name in the `.sout` file and the user sees a warning when typesetting. So we keep track of the largest label we've seen so that we don't write two labels with the same name.

```
346    self.max_counter_seen = MyDict()
```

Open a `.sout.tmp` file and write all our output to that. Then, when we're done, we move that to `.sout`. The "autogenerated" line is basically the same as the lines that get put at the top of preparsed Sage files; we are automatically generating a file with Sage, so it seems reasonable to add it. Add in the version to help debugging version mismatch problems.

```
347    self.souttmp = open(self.filename + '.sagetex.sout.tmp', 'w')
348    self.souttmp.write(autogenstr)
```

In addition to the `.sout` file, the `sagecommandline` also needs a `.scmd` file. As before, we use a `.scmd.tmp` file and rename it later on. We store the position so that `commandline` can tell the `listings` package what lines in the `.scmd` file to pull in.

```
349    self.scmdtmp = open(self.filename + '.sagetex.scmd.tmp', 'w')
350    self.scmdtmp.write(autogenstr)
351    self.scmdpos = 3
```

progress This function just prints stuff. It allows us to not print a linebreak, so you can get "`start...`" (little time spent processing) "`end`" on one line.

```
352    def progress(self, t,linebreak=True):
353      if linebreak:
354        print(t)
355      else:
356        sys.stdout.write(t)
357        sys.stdout.flush()
```

initplot We only want to create the plots directory if the user actually plots something. This function creates the directory and sets the `didinitplot` flag after doing so. We make a directory based on the LaTeX file being processed so that if there are multiple `.tex` files in a directory, we don't overwrite plots from another file.

```
358    def initplot(self):
359      self.progress('Initializing plots directory')
```

We hard-code the `.tex` extension, which is fine in the overwhelming majority of cases, although it does cause minor confusion when building the documentation. If it turns out lots of people use, say, a `ltx` extension or whatever, We could find out the correct extension, but it would involve a lot of irritating mucking around—on

`comp.text.tex`, the best solution I found for finding the file extension is to look through the `.log` file. (Although see the `currfile` package.)

```
360    if os.path.isdir(self.plotdir):
361        shutil.rmtree(self.plotdir)
362    os.mkdir(self.plotdir)
363    self.didinitplot = True
```

`inline` This function works with `\sage` from the style file (see section 6.1.2) to put Sage output into your LaTeX file. Usually, when you use `\label`, it writes a line such as

$$\text{\textbackslash newlabel\{labelname\}\{\{section number\}\{page number\}\}}$$

to the `.aux` file. When you use the `hyperref` package, there are more fields in the second argument, but the first two fields are the same. The `\ref` command just pulls in what's in the first field of the second argument, so we can hijack this mechanism for our own nefarious purposes. The function writes a `\newlabel` line with a label made from a counter and the text from running Sage on `s`.

When the user does `\sage` inside certain displayed math environments (`align` is the most common culprit) this function will get called twice with exactly the same arguments. We check to see what labels we've seen and immediately bail if we've written this label before.

The `labelname` defaults to the the name used by the usual `\sage` inline macro, but this function is also used by the `sagecommandline` environment. It's important to keep the corresponding labels separate, because `\sage` macros often (for example) appear inside math mode, and the labels from `sagecommandline` contain a `lstlistings` environment—pulling such an environment into math mode produces strange, unrecoverable errors, and if you can't typeset your file, you can't produce an updated `.sagetex.sage` file to run Sage on to produce a reasonable `.sagetext.sout` file that will fix the label problem. So it works much better to use distinct labels for such things.

We print out the line number so if something goes wrong, the user can more easily track down the offending `\sage` command in the source file.

That's a lot of explanation for a short function:

```
364  def inline(self, counter, s, labelname='sageinline'):
365      if counter <= self.max_counter_seen[labelname]:
366          return
367      else:
368          self.max_counter_seen.increment(labelname)
369      if labelname == 'sageinline':
370          self.progress('Inline formula {0} (line {1})'.format(counter, self.current_tex_line))
371      elif labelname == 'sagecmdline':
372          pass # output message already printed
373      else:
374          raise ValueError('inline() got a bad labelname "{0}"'.format(labelname))
375      self.souttmp.write(r'\newlabel{@' + labelname + str(counter) +
376                        '}{{%\n' + s.rstrip() + '}{}{}{}{}}\n')
```

We are using five fields, just like `hyperref` does, because that works whether or not `hyperref` is loaded. Using two fields, as in plain LaTeX, doesn't work if `hyperref` is loaded.

savecmd Analogous to `inline`, this method saves the input string `s` to the `souttmp` file. It returns the first and last line of the newly-added output so that `commandline` can tell the `listings` package where to get stuff.

```
377    def savecmd(self, s):
378        self.scmdtmp.write(s.rstrip() + "\n")
379        begin = self.scmdpos
380        end = begin + len(s.splitlines()) - 1
381        self.scmdpos = end + 1
382        return begin, end
```

blockbegin
blockend This function and its companion used to write stuff to the `.sout` file, but now they just update the user on our progress evaluating a code block. The verbatim-like environments of section 6.1.4 use these functions.

```
383    def blockbegin(self):
384      self.progress('Code block (line {}) begin...'.format(self.current_tex_line), False)
385    def blockend(self):
386      self.progress('end')
```

splitsagecmds Given a string `s` of doctest-like Sage code, this function returns a list of tuples `(i, j, cmd)`, where `cmd` is a string representing a Sage command, with the initial prompt and continuation lines stripped, `i` is the position in `s` where `cmd` starts, and `j` is the starting position in `s` of the purported output from the command that was included in `s`.

This is used by `doctest` and `commandline`, below.

For example, this turns the string

```
'''
    sage: 1+1
    2
    sage: y = 1729
    sage: 10^3 + 9^3 == 12^3 + 1^3 == y
    sage: gcd(9999999,
    ....: 123456)
    3
    sage: factor(x^2 + 2*x + 1)
    (x + 1)^2
'''
```

into

```
[(0, 18, '1+1'),
 (28, 51, 'y = 1729'),
 (51, 95, '10^3 + 9^3 == 12^3 + 1^3 == y'),
 (95, 144, 'gcd(9999999,\n123456)'),
 (154, 190, 'factor(x^2 + 2*x + 1)')]
```

You can reconstruct the sequence of commands and their output with something like

```
splitup = split_sage_cmds(s)
oldout = splitup[0][1]
print('=' * 50)
print('==== Command:')
print(s[splitup[0][0]:oldout])
for start, out, _ in splitup[1:]:
    print('==== Given output:')
    print(s[oldout:start])
    print('=' * 50)
    print('==== Command:')
    print(s[start:out])
    oldout = out
print('==== Given output:')
print(s[oldout:])
```

```
387  def split_sage_cmds(self, s):
388      prompt = '\n' + r'\s*sage: '
389      oldcont = r'\s*\.\.\.'
390      cont = r'\s*\.\.\.\.: '
```

Prepending a newline to `s` ensures that the list from `re.split()` begins with something we can ignore—and so that the `re.split()` returns $n+1$ groups for n matches, so therefore `split` and `starts` match up.

```
391      split = re.split(prompt, '\n' + s)[1:]
392      starts = [m.start() - 1 for m in re.finditer(prompt, '\n' + s)]
```

The prepended newline messes up the first element of `starts`, fix that:

```
393      starts[0] = re.search(prompt, s).start()
```

Now find where the outputs start. We need this because `doctest()` may or may not print the outputs. The idea is: for each `starting` position, advance over the prompt that we know is there, then look for the rightmost continuation marker between the current prompt and the next one, and *then* look for the newline following that. That position is where the output begins.

```
394      outputs = []
395      for i, j in zip(starts, starts[1:] + [len(s)]):
396          k = i + re.match(prompt, s[i:j]).end()
397          try:
398              k += [m.end() for m in re.finditer(cont, s[k:j])][-1]
399          except IndexError:
400              pass
401          end = s.find('\n', k)
402          outputs.append(end)
```

Now we take each command group, split it up, and look for all the continuation lines. We append those, stripping off the continuation marks. We also do some

error checking so that users with documents that use the old ... continuation marks get a reasonable error message.

```
403          ret = []
404          for start, end, g in zip(starts, outputs, split):
405              lines = g.splitlines()
406              cmd = lines[:1]
407              for line in lines[1:]:
408                  has_old_cont = re.match(oldcont, line)
409                  has_cont = re.match(cont, line)
410                  if has_old_cont and not has_cont:
411                      raise SyntaxError(""" SageTeX no longer supports "..." for line continuation
412 sagecommandline environments. Use "....:", which matches what the Sage
413 interpreter uses. See the documentation and example file in
414 SAGE_ROOT/local/share/doc/sagetex.""")
415                  if has_cont:
416                      cmd.append(line[has_cont.end():])
417              ret.append((start, end, '\n'.join(cmd)))
418          return ret
```

doctest This function handles the `sageexample` environment, which typesets Sage code and its output. We call it `doctest` because the format is just like that for doctests in the Sage library.

The idea is:

1. Get the literal text for each command, wrap that in SaveVerbatim, write that (possibly with its associated output from the .tex file) to the sout file.

2. Accumulate a corresponding UseVerbatim and typeset output so that we can call `inline()` at the end and pull in all this stuff.

3. For the output: try to `eval()` the processed command (the one with the prompts and continuation marks stripped). If that succeeds, we run `latex()` on it and display that below the verbatim text from above. If that fails, it's because you have a statement and not an expression—there's no output from such a thing (well, none that we can capture, anyway) so no need to typeset output.

```
419  def doctest(self, counter, s, globals, locals, include_text_output):
420      self.progress('Sage example {0} (line {1})'.format(counter, self.current_tex_line))
421      splitup = self.split_sage_cmds(s)
422      tex_strs = []
423      for i in range(len(splitup)):
424          boxname = '@sageinline{}-code{}'.format(counter, i)
425          to_tmp = [r'\begin{SaveVerbatim}{' + boxname + '}',
426                      s[splitup[i][0]:splitup[i][1]]]
427          if include_text_output:
428              try:
429                  to_tmp.append(s[splitup[i][1]:splitup[i+1][0]])
430              except IndexError:
431                  to_tmp.append(s[splitup[i][1]:])
```

```
432            to_tmp.append('\\end{SaveVerbatim}\n')
433            self.souttmp.write(joinone('\n', to_tmp))
```

Now we build up something that we can send to `inline()`, which will pull it into the document using its usual label mechanism.

The verbatim stuff seems to end with a bit of vertical space, so don't start the displaymath environment with unnecessary vertical space—the displayskip stuff is from §11.5 of Herbert Voß's "Math Mode".

```
434            tex_strs.append(r'\UseVerbatim{' + boxname + '}')
435            try:
436                result = eval(preparse(splitup[i][2]), globals, locals)
437                tex_strs += [r'\abovedisplayskip=0pt plus 3pt ',
438                             r'\abovedisplayshortskip=0pt plus 3pt ',
439                             r'\begin{displaymath}',
440                             latex(result),
441                             r' \end{displaymath}']
442            except SyntaxError:
443                exec(preparse(splitup[i][2]), globals, locals)
444        self.inline(counter, '\n'.join(tex_strs))
```

`commandline` This function handles the `commandline` environment, which typesets Sage code, computes its output, and typesets that too. This is very similar to `doctest` and I hope to someday combine them into one.

Even if I can't refactor these two functions (and their associated environments) into one, I would like to eliminate the `.scmd` file that this function uses, since exactly the same bits of Sage code get written to both the `.scmd` file and the `_doctest.sage` file. The reason this isn't trivial is because we need to keep track of which line number we're on so that we can give the `listings` package a start and end line to extract, and right now the `_doctest.sage` file is written to by LaTeX and we can't track the line number.

In any case, here's what we do: after splitting up the provided string using `split_sage_cmds`, we iterate over each of the commands and:

1. Put the original input command into the `.scmd` file with `savecmd`.

2. Use the `begin` and `end` line numbers to append a `lstinputlisting` command to the TeX commands we'll eventually hand off to `inline`.

3. Evaluate the command using `eval()` or `exec`, as necessary. If we're doing plain text format, we send the output to the `.scmd` file and add a TeX command to pull that back in—if we need typeset output, then we just hit the output with `latex()` and add that to the list of TeX commands.

Observe that we detect spaces in the filename and quote that for TeX if we need to.

```
445 def commandline(self, counter, s, globals, locals, text_output):
446     self.progress('Sage commandline {0} (line {1})'.format(counter, self.current_tex_line))
447     scmd_fn = self.name + '.sagetex.scmd'
448     if ' ' in scmd_fn:
```

```
449            scmd_fn = '"{}"'.format(scmd_fn)
450
451        splitup = self.split_sage_cmds(s)
452        skip = r'\vspace{\sagecommandlineskip}'
453        tex_strs = [skip]
454        lstinput = r'\lstinputlisting[firstline={0},lastline={1},firstnumber={2},style=SageInput{
455        for i in range(len(splitup)):
456            orig_input = s[splitup[i][0]:splitup[i][1]]
457            begin, end = self.savecmd(strip_common_leading_spaces(orig_input.strip('\n')))
458            if '#@' in orig_input:
459                escapeoption = ',escapeinside={\\#@}{\\^^M}'
460            else:
461                escapeoption = ''
462            tex_strs.append(lstinput.format(begin, end, begin - 2, scmd_fn, escape=escapeoption))
463            try:
464                result = eval(preparse(splitup[i][2]), globals, locals)
465                if text_output:
466                    begin, end = self.savecmd(str(result))
467                    tex_strs.append(lstinput.format(begin, end, begin - 2, scmd_fn, escape=''))
468                else:
469                    tex_strs.append(r'\begin{displaymath}' +
470                                    latex(result) +
471                                    r'\end{displaymath}')
472            except SyntaxError:
473                exec(preparse(splitup[i][2]), globals, locals)
474        if 'displaymath' not in tex_strs[-1]:
475            tex_strs.append(skip)
476        self.inline(counter, '\n'.join(tex_strs), labelname='sagecmdline')
```

plot I hope it's obvious that this function does plotting. It's the Python counterpart of \ST@sageplot described in section 6.1.3. As mentioned in the \sageplot code, we're taking advantage of two things: first, that LaTeX doesn't treat commas and spaces in macro arguments specially, and second, that Python (and Sage plotting functions) has nice support for keyword arguments. The #3 argument to \sageplot becomes _p_ and **kwargs below.

```
477    def plot(self, counter, _p_, format='notprovided', **kwargs):
478        if not self.didinitplot:
479            self.initplot()
480        self.progress('Plot {0} (line {1})'.format(counter, self.current_tex_line))
```

If the user says nothing about file formats, we default to producing PDF and EPS. This allows the user to transparently switch between using a DVI previewer (which usually automatically updates when the DVI changes, and has support for source specials, which makes the writing process easier) and making PDFs.[5]

```
481        if format == 'notprovided':
482            formats = ['eps', 'pdf']
483        else:
```

---

[5]Yes, there's pdfsync, but full support for that is still rare in Linux, so producing EPS and PDF is the best solution for now.

```
484              formats = [format]
485         for fmt in formats:
```

If we're making a PDF and have been told to use `epstopdf`, do so, then skip the rest of the loop.

```
486              if fmt == 'pdf' and self.useepstopdf:
487                  epsfile = os.path.join(self.plotdir, 'plot-{0}.eps'.format(counter))
488                  self.progress('Calling epstopdf to convert plot-{0}.eps to PDF'.format(
489                               counter))
490                  subprocess.check_call(['epstopdf', epsfile])
491                  continue
```

Some plot objects (mostly 3-D plots) do not support saving to EPS or PDF files (yet), but everything can be saved to a PNG file. For the user's convenience, we catch the error when we run into such an object, save it to a PNG file, then exit the loop.

```
492              plotfilename = os.path.join(self.plotdir, 'plot-{0}.{1}'.format(counter, fmt))
493              try:
494                  _p_.save(filename=plotfilename, **kwargs)
495              except ValueError as inst:
496                  if re.match('filetype .*not supported by save', str(inst)):
497                      newfilename = plotfilename[:-3] + 'png'
498                      print('  saving {0} failed; saving to {1} instead.'.format(
499                                               plotfilename, newfilename))
500                      _p_.save(filename=newfilename, **kwargs)
501                      break
502                  else:
503                      raise
```

If the user provides a format *and* specifies the `imagemagick` option, we try to convert the newly-created file into EPS format.

```
504              if format != 'notprovided' and self.useimagemagick:
505                  self.progress('Calling Imagemagick to convert plot-{0}.{1} to EPS'.format(
506                     counter, format))
507                  self.toeps(counter, format)
```

toeps   This function calls the Imagmagick utility `convert` to, well, convert something into EPS format. This gets called when the user has requested the "imagemagick" option to the SageTeX style file and is making a graphic file with a nondefault extension.

```
508   def toeps(self, counter, ext):
509      subprocess.check_call(['convert',\
510        '{0}/plot-{1}.{2}'.format(self.plotdir, counter, ext), \
511        '{0}/plot-{1}.eps'.format(self.plotdir, counter)])
```

We are blindly assuming that the `convert` command exists and will do the conversion for us; the `check_call` function raises an exception which, since all these calls get wrapped in try/excepts in the `.sage` file, should result in a reasonable error message if something strange happens.

goboom When a chunk of Sage code blows up, this function bears the bad news to the user. Normally in Python the traceback is good enough for this, but in this case, we start with a `.sage` file (which is autogenerated) which itself autogenerates a `.py` file—and the tracebacks the user sees refer to that file, whose line numbers are basically useless. We want to tell them where in the LaTeX file things went bad, so we do that, give them the traceback, and exit after removing the `.sout.tmp` and `.scmd.tmp` file.

```
512  def goboom(self, line):
513    print('\n**** Error in Sage code on line {0} of {1}.tex! Traceback\
514  follows.'.format(line, self.filename))
515    traceback.print_exc()
516    print('\n**** Running Sage on {0}.sage failed! Fix {0}.tex and try\
517  again.'.format(self.filename))
518    self.souttmp.close()
519    os.remove(self.filename + '.sagetex.sout.tmp')
520    self.scmdtmp.close()
521    os.remove(self.filename + '.sagetex.scmd.tmp')
522    sys.exit(int(1))
```

We use `int(1)` above to make sure `sys.exit` sees a Python integer; see ticket #2861.

endofdoc When we're done processing, we have some cleanup tasks. We want to put the MD5 sum of the `.sage` file that produced the `.sout` file we're about to write into the `.sout` file, so that external programs that build LaTeX documents can determine if they need to call Sage to update the `.sout` file. But there is a problem: we write line numbers to the `.sage` file so that we can provide useful error messages—but that means that adding non-SageTeX text to your source file will change the MD5 sum, and your program will think it needs to rerun Sage even though none of the actual SageTeX macros changed.

How do we include line numbers for our error messages but still allow a program to discover a "genuine" change to the `.sage` file?

The answer is to only find the MD5 sum of *part* of the `.sage` file. By design, the source file line numbers only appear in (1) calls to `goboom`, (2) lines with `_st_.current_tex_line`, and (3) pause/unpause lines, so we will strip those lines out. What we do below is exactly equivalent to running

```
egrep -v '^( _st_.goboom| ?_st_.current_tex_line|print(.SageT)' filename.sage | md5sum
```

in a shell. The included `run-sagetex-if-necessary` uses this mechanism to, well, only run Sage when necessary; see section 7.1.

```
523  def endofdoc(self):
524    sagef = open(self.filename + '.sagetex.sage', 'r')
525    m = hashlib.md5()
526    for line in sagef:
527      if not line.startswith((" _st_.goboom",
528                              "print('SageT",
529                              "_st_.current_tex_line",
```

42

```
530                                    " _st_.current_tex_line")):
531             if PY3:
532                 m.update(bytearray(line,'utf8'))
533             else:
534                 m.update(bytearray(line))
```

(The `current_tex_line` thing appears twice because it may appear indented one space or not, depending on whether it's used before `blockbegin` or not.)

```
535     s = '%' + m.hexdigest() + '% md5sum of corresponding .sage file\
536  (minus "goboom", "current_tex_line", and pause/unpause lines)\n'
537     self.souttmp.write(s)
538     self.scmdtmp.write(s)
```

Now, we do issue warnings to run Sage on the `.sage` file and an external program might look for those to detect the need to rerun Sage, but those warnings do not quite capture all situations. (If you've already produced the `.sout` file and change a `\sage` call, no warning will be issued since all the `\refs` find a `\newlabel`.) Anyway, I think it's easier to grab an MD5 sum out of the end of the file than parse the output from running `latex` on your file. (The regular expression `^%[0-9a-f]{32}%` will find the MD5 sum. Note that there are percent signs on each side of the hex string.)

Now we are done with the `.sout.tmp` file. Close it, rename it, and tell the user we're done.

```
539     self.souttmp.close()
540     os.rename(self.filename + '.sagetex.sout.tmp', self.filename + '.sagetex.sout')
541     self.scmdtmp.close()
542     os.rename(self.filename + '.sagetex.scmd.tmp', self.filename + '.sagetex.scmd')
543     self.progress('Sage processing complete. Run LaTeX on {0}.tex again.'.format(
544             self.filename))
```

7 Included Python scripts

Here we describe the Python code for `run-sagetex-if-necessary`, and also `makestatic.py`, which removes SageTeX commands to produce a "static" file, and `extractsagecode.py`, which extracts all the Sage code from a `.tex` file.

7.1 run-sagetex-if-necessary

When working on a document that uses SageTeX, running Sage every time you typeset your document may take too long, especially since it often won't be necessary. This script is a drop-in replacement for Sage: instead of

```
sage document.sagetex.sage
```

you can do

```
run-sagetex-if-necessary.py document.sagetex.sage
```

43

and it will use the MD5 mechanism described in the `endofdoc` macro (page 42).
With this, you can set up your editor (TEXShop, TEXWorks, etc) to typeset your
document with a script that does

```
pdflatex $1
run-sagetex-if-necessary.py $1
```

which will only, of course, run Sage when necessary.

```
545
546 # given a filename f, examines f.sagetex.sage and f.sagetex.sout and
547 # runs Sage if necessary.
548
549 import hashlib
550 import sys
551 import os
552 import re
553 import subprocess
554 from six import PY3
555
556 # CHANGE THIS AS APPROPRIATE
557 path_to_sage = os.path.expanduser('~/bin/sage')
558 # or try to auto-find it:
559 # path_to_sage = subprocess.check_output(['which', 'sage']).strip()
560 # or just tell me:
561 # path_to_sage = '/usr/local/bin/sage'
562
563 if sys.argv[1].endswith('.sagetex.sage'):
564     src = sys.argv[1][:-13]
565 else:
566     src = os.path.splitext(sys.argv[1])[0]
567
568 usepackage = r'usepackage(\[.*\])?{sagetex}'
569 uses_sagetex = False
570
571 # if it does not use sagetex, obviously running sage is unnecessary
572 with open(src + '.tex') as texf:
573     for line in texf:
574         if line.strip().startswith(r'\usepackage') and re.search(usepackage, line):
575             uses_sagetex = True
576             break
577
578 if not uses_sagetex:
579     print(src + ".tex doesn't seem to use SageTeX, exiting.")
580     sys.exit(0)
581
582 # if something goes wrong, assume we need to run Sage
583 run_sage = True
584 ignore = r"^( _st_.goboom|print\('SageT| ?_st_.current_tex_line)"
585
586 try:
```

```
587      with open(src + '.sagetex.sage', 'r') as sagef:
588          h = hashlib.md5()
589          for line in sagef:
590              if not re.search(ignore, line):
591                  if PY3:
592                      h.update(bytearray(line,'utf8'))
593                  else:
594                      h.update(bytearray(line))
595  except IOError:
596      print('{0}.sagetex.sage not found, I think you need to typeset {0}.tex first.'.format(src))
597      sys.exit(1)
598
599  try:
600      with open(src + '.sagetex.sout', 'r') as outf:
601          for line in outf:
602              m = re.match('%([0-9a-f]+)% md5sum', line)
603              if m:
604                  print('computed md5:', h.hexdigest())
605                  print('sagetex.sout md5:', m.group(1))
606                  if h.hexdigest() == m.group(1):
607                      run_sage = False
608                      break
609  except IOError:
610      pass
611
612  if run_sage:
613      print('Need to run Sage on {0}.'.format(src))
614      sys.exit(subprocess.call([path_to_sage, src + '.sagetex.sage']))
615  else:
616      print('Not necessary to run Sage on {0}.'.format(src))
```

7.2 makestatic.py

Now the `makestatic.py` script. It's about the most basic, generic Python script taking command-line arguments that you'll find. The `#!/usr/bin/env python` line is provided for us by the `.ins` file's preamble, so we don't put it here.

```
617  import sys
618  import time
619  import getopt
620  import os.path
621  from sagetexparse import DeSageTex
622
623  def usage():
624    print("""Usage: %s [-h|--help] [-o|--overwrite] inputfile [outputfile]
625
626  Removes SageTeX macros from 'inputfile' and replaces them with the
627  Sage-computed results to make a "static" file. You'll need to have run
628  Sage on 'inputfile' already.
629
```

```
630 'inputfile' can include the .tex extension or not. If you provide
631 'outputfile', the results will be written to a file of that name.
632 Specify '-o' or '--overwrite' to overwrite the file if it exists.
633
634 See the SageTeX documentation for more details.""" % sys.argv[0])
635
636 try:
637   opts, args = getopt.getopt(sys.argv[1:], 'ho', ['help', 'overwrite'])
638 except getopt.GetoptError as err:
639   print(str(err))
640   usage()
641   sys.exit(2)
642
643 overwrite = False
644 for o, a in opts:
645   if o in ('-h', '--help'):
646     usage()
647     sys.exit()
648   elif o in ('-o', '--overwrite'):
649     overwrite = True
650
651 if len(args) == 0 or len(args) > 2:
652   print('Error: wrong number of arguments. Make sure to specify options first.\n')
653   usage()
654   sys.exit(2)
655
656 if len(args) == 2 and (os.path.exists(args[1]) and not overwrite):
657   print('Error: %s exists and overwrite option not specified.' % args[1])
658   sys.exit(1)
659
660 src, ext = os.path.splitext(args[0])
```

All the real work gets done in the line below. Sorry it's not more exciting-looking.

```
661 desagetexed = DeSageTex(src)
```

This part is cool: we need double percent signs at the beginning of the line because
Python needs them (so they get turned into single percent signs) *and* because
Docstrip needs them (so the line gets passed into the generated file). It's perfect!

```
662 header = "%% SageTeX commands have been automatically removed from this file and\n%% replaced w
663
664 if len(args) == 2:
665   dest = open(args[1], 'w')
666 else:
667   dest = sys.stdout
668
669 dest.write(header)
670 dest.write(desagetexed.result)
```

7.3 extractsagecode.py

Same idea as `makestatic.py`, except this does basically the opposite thing.

```
671 import sys
672 import time
673 import getopt
674 import os.path
675 from sagetexparse import SageCodeExtractor
676
677 def usage():
678   print("""Usage: %s [-h|--help] [-o|--overwrite] inputfile [outputfile]
679
680 Extracts Sage code from 'inputfile'.
681
682 'inputfile' can include the .tex extension or not. If you provide
683 'outputfile', the results will be written to a file of that name,
684 otherwise the result will be printed to stdout.
685
686 Specify '-o' or '--overwrite' to overwrite the file if it exists.
687
688 See the SageTeX documentation for more details.""" % sys.argv[0])
689
690 try:
691   opts, args = getopt.getopt(sys.argv[1:], 'ho', ['help', 'overwrite'])
692 except getopt.GetoptError as err:
693   print(str(err))
694   usage()
695   sys.exit(2)
696
697 overwrite = False
698 for o, a in opts:
699   if o in ('-h', '--help'):
700     usage()
701     sys.exit()
702   elif o in ('-o', '--overwrite'):
703     overwrite = True
704
705 if len(args) == 0 or len(args) > 2:
706   print('Error: wrong number of arguments. Make sure to specify options first.\n')
707   usage()
708   sys.exit(2)
709
710 if len(args) == 2 and (os.path.exists(args[1]) and not overwrite):
711   print('Error: %s exists and overwrite option not specified.' % args[1])
712   sys.exit(1)
713
714 src, ext = os.path.splitext(args[0])
715 sagecode = SageCodeExtractor(src)
716 header = """\
717 # This file contains Sage code extracted from %s%s.
```

```
718 # Processed %s.
719
720 """ % (src, ext, time.strftime('%a %d %b %Y %H:%M:%S', time.localtime())))
721
722 if len(args) == 2:
723   dest = open(args[1], 'w')
724 else:
725   dest = sys.stdout
726
727 dest.write(header)
728 dest.write(sagecode.result)
```

7.4   The parser module

Here's the module that does the actual parsing and replacing. It's really quite simple, thanks to the awesome Pyparsing module. The parsing code below is nearly self-documenting! Compare that to fancy regular expressions, which sometimes look like someone sneezed punctuation all over the screen.

```
729 import sys
730 from pyparsing import *
```

First, we define this very helpful parser: it finds the matching bracket, and doesn't parse any of the intervening text. It's basically like hitting the percent sign in Vim. This is useful for parsing LATEX stuff, when you want to just grab everything enclosed by matching brackets.

```
731 def skipToMatching(opener, closer):
732   nest = nestedExpr(opener, closer)
733   nest.setParseAction(lambda l, s, t: l[s:getTokensEndLoc()])
734   return nest
735
736 curlybrackets = skipToMatching('{', '}')
737 squarebrackets = skipToMatching('[', ']')
```

Next, parser for \sage, \sageplot, and pause/unpause calls:

```
738 sagemacroparser = r'\sage' + curlybrackets('code')
739 sageplotparser = (r'\sageplot'
740                   + Optional(squarebrackets)('opts')
741                   + Optional(squarebrackets)('format')
742                   + curlybrackets('code'))
743 sagetexpause = Literal(r'\sagetexpause')
744 sagetexunpause = Literal(r'\sagetexunpause')
```

With those defined, let's move on to our classes.

SoutParser   Here's the parser for the generated .sout file. The code below does all the parsing of the .sout file and puts the results into a list. Notice that it's on the order of 10 lines of code—hooray for Pyparsing!

```
745 class SoutParser():
746   def __init__(self, fn):
747     self.label = []
```

48

A label line looks like

$$\texttt{\textbackslash newlabel\{@sageinline}\langle integer\rangle \texttt{\}\{\{}\langle bunch\ of\ \LaTeX\ code\rangle \texttt{\}\{\}\{\}\{\}\{\}\}}$$

which makes the parser definition below pretty obvious. We assign some names to the interesting bits so the `newlabel` method can make the ⟨*integer*⟩ and ⟨*bunch of LaTeX code*⟩ into the keys and values of a dictionary. The `DeSageTeX` class then uses that dictionary to replace bits in the `.tex` file with their Sage-computed results.

```
748     parselabel = (r'\newlabel{@sageinline'
749                   + Word(nums)('num')
750                   + '}{'
751                   + curlybrackets('result')
752                   + '{}{}{}{}}')
```

We tell it to ignore comments, and hook up the list-making method.

```
753     parselabel.ignore('%' + restOfLine)
754     parselabel.setParseAction(self.newlabel)
```

A `.sout` file consists of one or more such lines. Now go parse the file we were given.

```
755     try:
756         OneOrMore(parselabel).parseFile(fn)
757     except IOError:
758         print('Error accessing {}; exiting. Does your .sout file exist?'.format(fn))
759         sys.exit(1)
```

Pyparser's parse actions get called with three arguments: the string that matched, the location of the beginning, and the resulting parse object. Here we just add a new key-value pair to the dictionary, remembering to strip off the enclosing brackets from the "result" bit.

```
760   def newlabel(self, s, l, t):
761       self.label.append(t.result[1:-1])
```

DeSageTeX   Now we define a parser for LaTeX files that use SageTeX commands. We assume that the provided `fn` is just a basename.

```
762 class DeSageTex():
763   def __init__(self, fn):
764       self.sagen = 0
765       self.plotn = 0
766       self.fn = fn
767       self.sout = SoutParser(fn + '.sagetex.sout')
```

Parse `\sage` macros. We just need to pull in the result from the `.sout` file and increment the counter—that's what `self.sage` does.

```
768     smacro = sagemacroparser
769     smacro.setParseAction(self.sage)
```

Parse the `\usepackage{sagetex}` line. Right now we don't support comma-separated lists of packages.

```
770     usepackage = (r'\usepackage'
```

```
771                    + Optional(squarebrackets)
772                    + '{sagetex}')
773     usepackage.setParseAction(replaceWith(r"""% "\usepackage{sagetex}" line was here:
774 \RequirePackage{verbatim}
775 \RequirePackage{graphicx}
776 \newcommand{\sagetexpause}{\relax}
777 \newcommand{\sagetexunpause}{\relax}"""))
```

Parse \sageplot macros.

```
778     splot = sageplotparser
779     splot.setParseAction(self.plot)
```

The printed environments (sageblock and sageverbatim) get turned into verbatim environments.

```
780     beginorend = oneOf('begin end')
781     blockorverb = 'sage' + oneOf('block verbatim')
782     blockorverb.setParseAction(replaceWith('verbatim'))
783     senv = '\\' + beginorend + '{' + blockorverb + '}'
```

The non-printed sagesilent environment gets commented out. We could remove all the text, but this works and makes going back to SageTEX commands (de-de-SageTEXing?) easier.

```
784     silent = Literal('sagesilent')
785     silent.setParseAction(replaceWith('comment'))
786     ssilent = '\\' + beginorend + '{' + silent + '}'
```

The \sagetexindent macro is no longer relevant, so remove it from the output ("suppress", in Pyparsing terms).

```
787     stexindent = Suppress(r'\setlength{\sagetexindent}' + curlybrackets)
```

Now we define the parser that actually goes through the file. It just looks for any one of the above bits, while ignoring anything that should be ignored.

```
788     doit = smacro | senv | ssilent | usepackage | splot | stexindent
789     doit.ignore('%' + restOfLine)
790     doit.ignore(r'\begin{verbatim}' + SkipTo(r'\end{verbatim}'))
791     doit.ignore(r'\begin{comment}' + SkipTo(r'\end{comment}'))
792     doit.ignore(r'\sagetexpause' + SkipTo(r'\sagetexunpause'))
```

We can't use the parseFile method, because that expects a "complete grammar" in which everything falls into some piece of the parser. Instead we suck in the whole file as a single string, and run transformString on it, since that will just pick out the interesting bits and munge them according to the above definitions.

```
793     str = ''.join(open(fn + '.tex', 'r').readlines())
794     self.result = doit.transformString(str)
```

That's the end of the class constructor, and it's all we need to do here. You access the results of parsing via the result string.

We do have two methods to define. The first does the same thing that \ref does in your LaTeX file: returns the content of the label and increments a counter.

```
795 def sage(self, s, l, t):
796     self.sagen += 1
797     return self.sout.label[self.sagen - 1]
```

50

The second method returns the appropriate \includegraphics command. It does need to account for the default argument.

```
798    def plot(self, s, l, t):
799      self.plotn += 1
800      if len(t.opts) == 0:
801        opts = r'[width=.75\textwidth]'
802      else:
803        opts = t.opts[0]
804      return (r'\includegraphics%s{sage-plots-for-%s.tex/plot-%s}' %
805        (opts, self.fn, self.plotn - 1))
```

SageCodeExtractor This class does the opposite of the first: instead of removing Sage stuff and leaving only LATEX, this removes all the LATEX and leaves only Sage.

```
806 class SageCodeExtractor():
807   def __init__(self, fn):
808     smacro = sagemacroparser
809     smacro.setParseAction(self.macroout)
810
811     splot = sageplotparser
812     splot.setParseAction(self.plotout)
```

Above, we used the general parsers for \sage and \sageplot. We have to redo the environment parsers because it seems too hard to define one parser object that will do both things we want: above, we just wanted to change the environment name, and here we want to suck out the code. Here, it's important that we find matching begin/end pairs; above it wasn't. At any rate, it's not a big deal to redo this parser.

```
813     env_names = oneOf('sageblock sageverbatim sagesilent')
814     senv = r'\begin{' + env_names('env') + '}' + SkipTo(
815            r'\end{' + matchPreviousExpr(env_names) + '}')('code')
816     senv.leaveWhitespace()
817     senv.setParseAction(self.envout)
818
819     spause = sagetexpause
820     spause.setParseAction(self.pause)
821
822     sunpause = sagetexunpause
823     sunpause.setParseAction(self.unpause)
824
825     doit = smacro | splot | senv | spause | sunpause
826
827     str = ''.join(open(fn + '.tex', 'r').readlines())
828     self.result = ''
829
830     doit.transformString(str)
831
832   def macroout(self, s, l, t):
833     self.result += '# \\sage{} from line %s\n' % lineno(l, s)
834     self.result += t.code[1:-1] + '\n\n'
```

```
835
836  def plotout(self, s, l, t):
837    self.result += '# \\sageplot{} from line %s:\n' % lineno(l, s)
838    if t.format is not '':
839      self.result += '# format: %s' % t.format[0][1:-1] + '\n'
840    self.result += t.code[1:-1] + '\n\n'
841
842  def envout(self, s, l, t):
843    self.result += '# %s environment from line %s:' % (t.env,
844      lineno(l, s))
845    self.result += t.code[0] + '\n'
846
847  def pause(self, s, l, t):
848    self.result += ('# SageTeX (probably) paused on input line %s.\n\n' %
849                    (lineno(l, s)))
850
851  def unpause(self, s, l, t):
852    self.result += ('# SageTeX (probably) unpaused on input line %s.\n\n' %
853                    (lineno(l, s)))
```

8   The `remote-sagetex` script

Here we describe the Python code for `remote-sagetex.py`. Since its job is to replicate the functionality of using Sage and `sagetex.py`, there is some overlap with the Python module.

The `#!/usr/bin/env python` line is provided for us by the `.ins` file's preamble, so we don't put it here.

```
854 from __future__ import print_function
855 import json
856 import sys
857 import time
858 import re
859 import urllib
860 import hashlib
861 import os
862 import os.path
863 import shutil
864 import getopt
865 from contextlib import closing
866
867 ##############################################################################
868 # You can provide a filename here and the script will read your login    #
869 # information from that file. The format must be:                        #
870 #                                                                        #
871 # server = 'http://foo.com:8000'                                         #
872 # username = 'my_name'                                                   #
873 # password = 's33krit'                                                   #
874 #                                                                        #
```

```
875 # You can omit one or more of those lines, use " quotes, and put hash    #
876 # marks at the beginning of a line for comments. Command-line args       #
877 # take precedence over information from the file.                        #
878 ###########################################################################
879 login_info_file = None       # e.g. '/home/foo/Private/sagetex-login.txt'
880
881
882 usage = """Process a SageTeX-generated .sage file using a remote Sage server.
883
884 Usage: {0} [options] inputfile.sage
885
886 Options:
887
888     -h, --help:         print this message
889     -s, --server:       the Sage server to contact
890     -u, --username:     username on the server
891     -p, --password:     your password
892     -f, --file:         get login information from a file
893
894 If the server does not begin with the four characters 'http', then
895 'https://' will be prepended to the server name.
896
897 You can hard-code the filename from which to read login information into
898 the remote-sagetex script. Command-line arguments take precedence over
899 the contents of that file. See the SageTeX documentation for formatting
900 details.
901
902 If any of the server, username, and password are omitted, you will be
903 asked to provide them.
904
905 See the SageTeX documentation for more details on usage and limitations
906 of remote-sagetex.""".format(sys.argv[0])
907
908 server, username, password = (None,) * 3
909
910 try:
911     opts, args = getopt.getopt(sys.argv[1:], 'hs:u:p:f:',
912                     ['help', 'server=', 'user=', 'password=', 'file='])
913 except getopt.GetoptError as err:
914     print(str(err), usage, sep='\n\n')
915     sys.exit(2)
916
917 for o, a in opts:
918     if o in ('-h', '--help'):
919         print(usage)
920         sys.exit()
921     elif o in ('-s', '--server'):
922         server = a
923     elif o in ('-u', '--user'):
924         username = a
```

```
925    elif o in ('-p', '--password'):
926        password = a
927    elif o in ('-f', '--file'):
928        login_info_file = a
929
930 if len(args) != 1:
931     print('Error: must specify exactly one file. Please specify options first.',
932            usage, sep='\n\n')
933     sys.exit(2)
934
935 jobname = os.path.splitext(args[0])[0]
```

When we send things to the server, we get everything back as a string, including
tracebacks. We can search through output using regexps to look for typical trace-
back strings, but there's a more robust way: put in a special string that changes
every time and is printed when there's an error, and look for that. Then it is
massively unlikely that a user's code could produce output that we'll mistake for
an actual traceback. System time will work well enough for these purposes. We
produce this string now, and we it when parsing the .sage file (we insert it into
code blocks) and when parsing the output that the remote server gives us.

```
936 traceback_str = 'Exception in SageTeX session {0}:'.format(time.time())
```

parsedotsage   To figure out what commands to send the remote server, we actually read in the
.sage file as strings and parse it. This seems a bit strange, but since we know
exactly what the format of that file is, we can parse it with a couple flags and a
handful of regexps.

```
937 def parsedotsage(fn):
938     with open(fn, 'r') as f:
```

Here are the regexps we use to snarf the interesting bits out of the .sage file.
Below we'll use the re module's match function so we needn't anchor any of these
at the beginning of the line.

```
939        inline = re.compile(r" _st_.inline\((?P<num>\d+), (?P<code>.*)\)")
940        plot = re.compile(r" _st_.plot\((?P<num>\d+), (?P<code>.*)\)")
941        goboom = re.compile(r" _st_.goboom\((?P<num>\d+)\)")
942        pausemsg = re.compile(r"print.'(?P<msg>SageTeX (un)?paused.*)'")
943        blockbegin = re.compile(r"_st_.blockbegin\(\)")
944        ignore = re.compile(r"(try:)|(except):")
945        in_comment = False
946        in_block = False
947        cmds = []
```

Okay, let's go through the file. We're going to make a list of dictionaries. Each
dictionary corresponds to something we have to do with the remote server, except
for the pause/unpause ones, which we only use to print out information for the
user. All the dictionaries have a type key, which obviously tells you type they
are. The pause/unpause dictionaries then just have a msg which we toss out to
the user. The "real" dictionaries all have the following keys:

- type: one of inline, plot, and block.

54

- **goboom**: used to help the user pinpoint errors, just like the **goboom** function (page 41) does.

- **code**: the code to be executed.

Additionally, the **inline** and **plot** dicts have a **num** key for the label we write to the **.sout** file.

Here's the whole parser loop. The interesting bits are for parsing blocks because there we need to accumulate several lines of code.

```
948        for line in f.readlines():
949            if line.startswith('"""'):
950                in_comment = not in_comment
951            elif not in_comment:
952                m = pausemsg.match(line)
953                if m:
954                    cmds.append({'type': 'pause',
955                                 'msg': m.group('msg')})
956                m = inline.match(line)
957                if m:
958                    cmds.append({'type': 'inline',
959                                 'num': m.group('num'),
960                                 'code': m.group('code')})
961                m = plot.match(line)
962                if m:
963                    cmds.append({'type': 'plot',
964                                 'num': m.group('num'),
965                                 'code': m.group('code')})
```

The order of the next three "if"s is important, since we need the "goboom" line and the "blockbegin" line to *not* get included into the block's code. Note that the lines in the **.sage** file already have some indentation, which we'll use when sending the block to the server—we wrap the text in a try/except.

```
966                m = goboom.match(line)
967                if m:
968                    cmds[-1]['goboom'] = m.group('num')
969                    if in_block:
970                        in_block = False
971                if in_block and not ignore.match(line):
972                    cmds[-1]['code'] += line
973                if blockbegin.match(line):
974                    cmds.append({'type': 'block',
975                                 'code': ''})
976                    in_block = True
977    return cmds
```

Parsing the **.sage** file is simple enough so that we can write one function and just do it. Interacting with the remote server is a bit more complicated, and requires us to carry some state, so let's make a class.

RemoteSage   This is pretty simple; it's more or less a translation of the examples in
sage/server/simple/twist.py.

```
978 debug = False
979 class RemoteSage:
980     def __init__(self, server, user, password):
981         self._srv = server.rstrip('/')
982         sep = '___S_A_G_E___'
983         self._response = re.compile('(?P<header>.*)' + sep +
984                                     '\n*(?P<output>.*)', re.DOTALL)
985         self._404 = re.compile('404 Not Found')
986         self._session = self._get_url('login',
987                                 urllib.urlencode({'username': user,
988                                 'password':
989                                 password}))['session']
```

In the string below, we want to do "partial formatting": we format in the traceback
string now, and want to be able to format in the code later. The double braces
get ignored by `format()` now, and are picked up by `format()` when we use this
later.

```
990         self._codewrap = """try:
991 {{0}}
992 except:
993     print('{0}')
994     traceback.print_exc()""".format(traceback_str)
995         self.do_block("""
996     import traceback
997     def __st_plot__(counter, _p_, format='notprovided', **kwargs):
998         if format == 'notprovided':
999             formats = ['eps', 'pdf']
1000        else:
1001            formats = [format]
1002        for fmt in formats:
1003            plotfilename = 'plot-%s.%s' % (counter, fmt)
1004            _p_.save(filename=plotfilename, **kwargs)""")
1005
1006    def _encode(self, d):
1007        return 'session={0}&'.format(self._session) + urllib.urlencode(d)
1008
1009    def _get_url(self, action, u):
1010        with closing(urllib.urlopen(self._srv + '/simple/' + action +
1011                                    '?' + u)) as h:
1012            data = self._response.match(h.read())
1013            result = json.loads(data.group('header'))
1014            result['output'] = data.group('output').rstrip()
1015        return result
1016
1017    def _get_file(self, fn, cell, ofn=None):
1018        with closing(urllib.urlopen(self._srv + '/simple/' + 'file' + '?' +
1019                    self._encode({'cell': cell, 'file': fn}))) as h:
1020            myfn = ofn if ofn else fn
```

```
1021              data = h.read()
1022              if not self._404.search(data):
1023                  with open(myfn, 'w') as f:
1024                      f.write(data)
1025              else:
1026                  print('Remote server reported {0} could not be found:'.format(
1027                      fn))
1028                  print(data)
```

The `code` below gets stuffed between a try/except, so make sure it's indented!

```
1029      def _do_cell(self, code):
1030          realcode = self._codewrap.format(code)
1031          result = self._get_url('compute', self._encode({'code': realcode}))
1032          if result['status'] == 'computing':
1033              cell = result['cell_id']
1034              while result['status'] == 'computing':
1035                  sys.stdout.write('working...')
1036                  sys.stdout.flush()
1037                  time.sleep(10)
1038                  result = self._get_url('status', self._encode({'cell': cell}))
1039          if debug:
1040              print('cell: <<<', realcode, '>>>', 'result: <<<',
1041                  result['output'], '>>>', sep='\n')
1042          return result
1043
1044      def do_inline(self, code):
1045          return self._do_cell(' print(latex({0}))'.format(code))
1046
1047      def do_block(self, code):
1048          result = self._do_cell(code)
1049          for fn in result['files']:
1050              self._get_file(fn, result['cell_id'])
1051          return result
1052
1053      def do_plot(self, num, code, plotdir):
1054          result = self._do_cell(' __st_plot__({0}, {1})'.format(num, code))
1055          for fn in result['files']:
1056              self._get_file(fn, result['cell_id'], os.path.join(plotdir, fn))
1057          return result
```

When using the simple server API, it's important to log out so the server doesn't accumulate idle sessions that take up lots of memory. We define a `close()` method and use this class with the `closing` context manager that always calls `close()` on the way out.

```
1058      def close(self):
1059          sys.stdout.write('Logging out of {0}...'.format(server))
1060          sys.stdout.flush()
1061          self._get_url('logout', self._encode({}))
1062          print('done')
```

Next we have a little pile of miscellaneous functions and variables that we want to

have at hand while doing our work. Note that we again use the traceback string in the error-finding regular expression.

```
1063 def do_plot_setup(plotdir):
1064     printc('initializing plots directory...')
1065     if os.path.isdir(plotdir):
1066         shutil.rmtree(plotdir)
1067     os.mkdir(plotdir)
1068     return True
1069
1070 did_plot_setup = False
1071 plotdir = 'sage-plots-for-' + jobname + '.tex'
1072
1073 def labelline(n, s):
1074     return r'\newlabel{@sageinline' + str(n) + '}{{' + s  + '}{}{}{}{}}\n'
1075
1076 def printc(s):
1077     print(s, end='')
1078     sys.stdout.flush()
1079
1080 error = re.compile("(^" + traceback_str + ")|(^Syntax Error:)", re.MULTILINE)
1081
1082 def check_for_error(string, line):
1083     if error.search(string):
1084         print("""
1085 **** Error in Sage code on line {0} of {1}.tex!
1086 {2}
1087 **** Running Sage on {1}.sage failed! Fix {1}.tex and try again.""".format(
1088             line, jobname, string))
1089         sys.exit(1)
```

Now let's actually start doing stuff.

```
1090 print('Processing Sage code for {0}.tex using remote Sage server.'.format(
1091     jobname))
1092
1093 if login_info_file:
1094     with open(login_info_file, 'r') as f:
1095         print('Reading login information from {0}.'.format(login_info_file))
1096         get_val = lambda x: x.split('=')[1].strip().strip('\'"')
1097         for line in f:
1098             print(line)
1099             if not line.startswith('#'):
1100                 if line.startswith('server') and not server:
1101                     server = get_val(line)
1102                 if line.startswith('username') and not username:
1103                     username = get_val(line)
1104                 if line.startswith('password') and not password:
1105                     password = get_val(line)
1106
1107 if not server:
1108     server = raw_input('Enter server: ')
```

```
1109
1110 if not server.startswith('http'):
1111     server = 'https://' + server
1112
1113 if not username:
1114     username = raw_input('Enter username: ')
1115
1116 if not password:
1117     from getpass import getpass
1118     password = getpass('Please enter password for user {0} on {1}: '.format(
1119         username, server))
1120
1121 printc('Parsing {0}.sage...'.format(jobname))
1122 cmds = parsedotsage(jobname + '.sage')
1123 print('done.')
1124
1125 sout = '% This file was *autogenerated* from the file {0}.sage.\n'.format(
1126     os.path.splitext(jobname)[0])
1127
1128 printc('Logging into {0} and starting session...'.format(server))
1129 with closing(RemoteSage(server, username, password)) as sage:
1130     print('done.')
1131     for cmd in cmds:
1132         if cmd['type'] == 'inline':
1133             printc('Inline formula {0}...'.format(cmd['num']))
1134             result = sage.do_inline(cmd['code'])
1135             check_for_error(result['output'], cmd['goboom'])
1136             sout += labelline(cmd['num'], result['output'])
1137             print('done.')
1138         if cmd['type'] == 'block':
1139             printc('Code block begin...')
1140             result = sage.do_block(cmd['code'])
1141             check_for_error(result['output'], cmd['goboom'])
1142             print('end.')
1143         if cmd['type'] == 'plot':
1144             printc('Plot {0}...'.format(cmd['num']))
1145             if not did_plot_setup:
1146                 did_plot_setup = do_plot_setup(plotdir)
1147             result = sage.do_plot(cmd['num'], cmd['code'], plotdir)
1148             check_for_error(result['output'], cmd['goboom'])
1149             print('done.')
1150         if cmd['type'] == 'pause':
1151             print(cmd['msg'])
1152         if int(time.time()) % 2280 == 0:
1153             printc('Unscheduled offworld activation; closing iris...')
1154             time.sleep(1)
1155             print('end.')
1156
1157 with open(jobname + '.sage', 'r') as sagef:
1158     h = hashlib.md5()
```

```
1159    for line in sagef:
1160        if (not line.startswith(' _st_.goboom') and
1161            not line.startswith("print('SageT")):
1162            h.update(bytearray(line,'utf8'))
```

Putting the `{1}` in the string, just to replace it with %, seems a bit weird, but if I put a single percent sign there, Docstrip won't put that line into the resulting `.py` file—and if I put two percent signs, it replaces them with `\MetaPrefix` which is `##` when this file is generated. This is a quick and easy workaround.

```
1163    sout += """%{0}% md5sum of corresponding .sage file
1164 {1} (minus "goboom" and pause/unpause lines)
1165 """.format(h.hexdigest(), '%')
1166
1167 printc('Writing .sout file...')
1168 with open(jobname + '.sout', 'w') as soutf:
1169    soutf.write(sout)
1170    print('done.')
1171 print('Sage processing complete. Run LaTeX on {0}.tex again.'.format(jobname))
```

9   Credits and acknowledgments

According to the original README file, this system was originally done by Gonzalo Tornaria and Joe Wetherell. Later Harald Schilly made some improvements and modifications. Many of the examples in the `example.tex` file are from Harald.

Dan Drake rewrote and extended the style file (there is effectively zero original code there), made significant changes to the Python module, put both files into Docstrip format, and wrote all the documentation and extra Python scripts.

Many thanks to Jason Grout for his numerous comments, suggestions, and feedback. Thanks to Nicolas Thiéry for the initial code and contributions to the `sageexample` environment and Volker Braun for the `sagecommandline` environment.

10   Copying and licenses

If you are unnaturally curious about the current state of the SageTeX package, you can visit `https://github.com/sagemath/sagetex`. (The old Bitbucket and Github dandrake repositories are deprecated.)

As for the terms and conditions under which you can copy and modify SageTeX:

The *source code* of the SageTeX package may be redistributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation, either version 2 of the License, or (at your option) any later version. To view a copy of this license, see `http://www.gnu.org/licenses/` or send a letter to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

The *documentation* of the SageTeX package is licensed under the Creative Commons Attribution-Share Alike 3.0 License. To view a copy of this license,

visit `http://creativecommons.org/licenses/by-sa/3.0/` or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

I am not terribly dogmatic about these licenses, so if you would like to do something with SageTeX that's not possible under these license conditions, please contact me. I will likely be receptive to suggestions.

Change History

Mercurial revision in version
string. 21

`inline`: check label name when
comparing against maximum
counter seen; trac ticket 12267   35

v3.0

`doctest`: sageexample and
sagecommandline now require
"...:" for continuation lines,
not "..."; matches the actual
Sage interpreter 39

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.

65