The SageTEX package[*]

Dan Drake and others[†]

July 26, 2023

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 $26^3 \cdot 10^3 =$ **??** 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 v3.4, dated 2019/11/14.

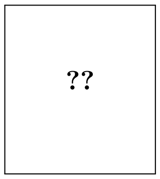
[†]Package's website: `github.com/sagemath/sagetex`.

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 3.1.9). 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 ⟨*filename*⟩`.tex`, the file ⟨*filename*⟩`.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).

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.

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

3.1 Inline Sage

sage $\boxed{\texttt{\textbackslash sage\{}\langle\textit{Sage code}\rangle\texttt{\}}}$ 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 **??**. (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 $\boxed{\texttt{\textbackslash sagestr\{}\langle\textit{Sage code}\rangle\texttt{\}}}$ 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; 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.

[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.

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.

3.1.1 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` $\boxed{\texttt{\textbackslash sageplot[}\langle\textit{ltx opts}\rangle\texttt{][}\langle\textit{fmt}\rangle\texttt{]\{}\langle\textit{graphics obj}\rangle,\ \langle\textit{keyword args}\rangle\texttt{\}}}$ 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}
```

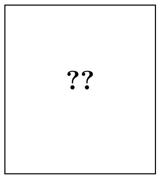
You can specify a file format if you like. This must be the *second* optional argument, so you must use empty brackets if you're not passing anything to `\includegraphics`:

```
\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:



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.

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]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.

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 **??**. 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 allows 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: ??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 allows 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 ??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 an "at" sign is escaped to LaTeX. In particular, you can use `\label` to mark line numbers in order to `\ref`erence 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 ?? 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: **??**

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`
`\sagetexunpause`

Use these macros to "pause" and "unpause" SageTEX. After issuing this macro, 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 `sagetex-run` 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 together, 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

CoCalc (`cocalc.com/`) 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 CoCalc 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 CoCalc, `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 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.)

[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.

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

```
\RequirePackage{verbatim}
\RequirePackage{fancyvrb}
```

and `listings` for the `sagecommandline` environment.

```
\RequirePackage{listings}
\RequirePackage{xcolor}
\lstdefinelanguage{Sage}[]{Python}
  {morekeywords={False,sage,True},sensitive=true}
\lstdefinelanguage{SageOutput}[]{}
  {morekeywords={False,True},sensitive=true}
\lstdefinestyle{DefaultSageInputOutput}{
  nolol,
  identifierstyle=,
  name=sagecommandline,
  xleftmargin=5pt,
  numbersep=5pt,
  aboveskip=0pt,
  belowskip=0pt,
  breaklines=true,
  numberstyle=\footnotesize,
  numbers=right
}
\lstdefinestyle{DefaultSageInput}{
  language=Sage,
  style=DefaultSageInputOutput,
  basicstyle={\ttfamily\bfseries},
  commentstyle={\ttfamily\color{dgreencolor}},
  keywordstyle={\ttfamily\color{dbluecolor}\bfseries},
  stringstyle={\ttfamily\color{dgraycolor}\bfseries},
}
\lstdefinestyle{DefaultSageOutput}{
  language=SageOutput,
  style=DefaultSageInputOutput,
  basicstyle={\ttfamily},
  commentstyle={\ttfamily\color{dgreencolor}},
  keywordstyle={\ttfamily\color{dbluecolor}},
  stringstyle={\ttfamily\color{dgraycolor}},
}
\lstdefinestyle{SageInput}{
  style=DefaultSageInput,
}
\lstdefinestyle{SageOutput}{
  style=DefaultSageOutput,
}
\providecolor{dbluecolor}{rgb}{0.01,0.02,0.7}
\providecolor{dgreencolor}{rgb}{0.2,0.4,0.0}
\providecolor{dgraycolor}{rgb}{0.30,0.3,0.30}
```

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

```
\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.

```
\RequirePackage{makecmds}
\RequirePackage{ifpdf}
\RequirePackage{ifthen}
\IfFileExists{ifxetex.sty}{
  \RequirePackage{ifxetex}
}{
  \newboolean{xetex}
  \setboolean{xetex}{false}}
```

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

```
\newcounter{ST@inline}
\newcounter{ST@plot}
\newcounter{ST@cmdline}
\setcounter{ST@inline}{0}
\setcounter{ST@plot}{0}
\setcounter{ST@cmdline}{0}
\newlength{\sagetexindent}
```

```
\setlength{\sagetexindent}{5ex}
\newif\ifST@paused
\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:

```
\AtBeginDocument{\@ifundefined{ST@final}{%
\newwrite\ST@sf%
\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 33 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`.

```
\newcommand{\ST@wsf}[1]{\immediate\write\ST@sf{#1}}%
\ST@wsf{%
# -*- encoding: utf-8 -*-^^J%
# This file (\jobname.sagetex.sage) was *autogenerated* from \jobname.tex with
sagetex.sty version \ST@ver.^^J%
import sagetex^^J%
_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.

```
{\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.

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

```
\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.

```
\DeclareOption{final}{%
  \newcommand{\ST@final}{x}%
  \IfFileExists{\jobname.sagetex.sout}{}{\AtEndDocument{\PackageWarningNoLine{sagetex}%
  {‘final’ option provided, but \jobname.sagetex.sout^^Jdoesn’t exist! No Sage
    input will appear in your document. Remove the ‘final’^^Joption and
    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.

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

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

```
\DeclareOption{epstopdf}{%
\AtBeginDocument{%
\@ifundefined{ST@final}{%
  \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.

```
\newcommand{\ST@versioncheck}{True}
\DeclareOption{noversioncheck}{%
  \renewcommand{\ST@versioncheck}{False}}
\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.

```
\InputIfFileExists{\jobname.sagetex.sout}{}
{\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.

```
\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.

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

The `inline` function of the Python module is documented on page 34. Back in LaTeX-land: if paused, say so.

```
\ifST@paused
  \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.

```
\else
  \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.

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

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

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

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

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

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

```
\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`.

```
\catcode`\%=12
\newcommand{\percent}{%}
\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.

```
\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.

```
\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.

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

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.

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

The Python `plot` function is documented on page 39.

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 Ti*k*Z 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:

```
\ifthenelse{\boolean{pdf} \or \boolean{xetex}}{
  \ifthenelse{\equal{#2}{notprovided}}%
    {\ST@inclgrfx{#1}{pdf}}%
    {\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.)

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

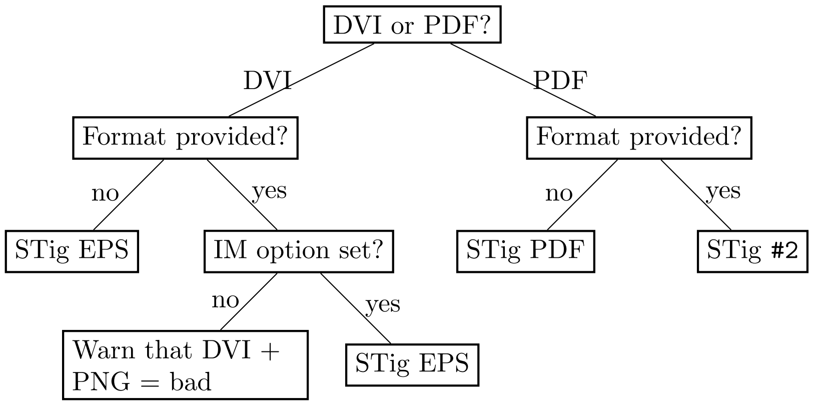


Figure 1: The logic tree that `\sageplot` uses to decide whether to run `\includegraphics` or to yell at the user. “Format” is the `#2` argument to `\sageplot`, “STig ext” means a call to `\ST@inclgrfx` with “ext” as the second argument, and “IM” is Imagemagick.

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.

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

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

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

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

```
\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.

```
\newcommand{\ST@inclgrfx}[2]{\ifST@paused
  \fbox{\rule[-1cm]{0cm}{2cm}Sage\TeX{} is paused; no graphic}
\else
```

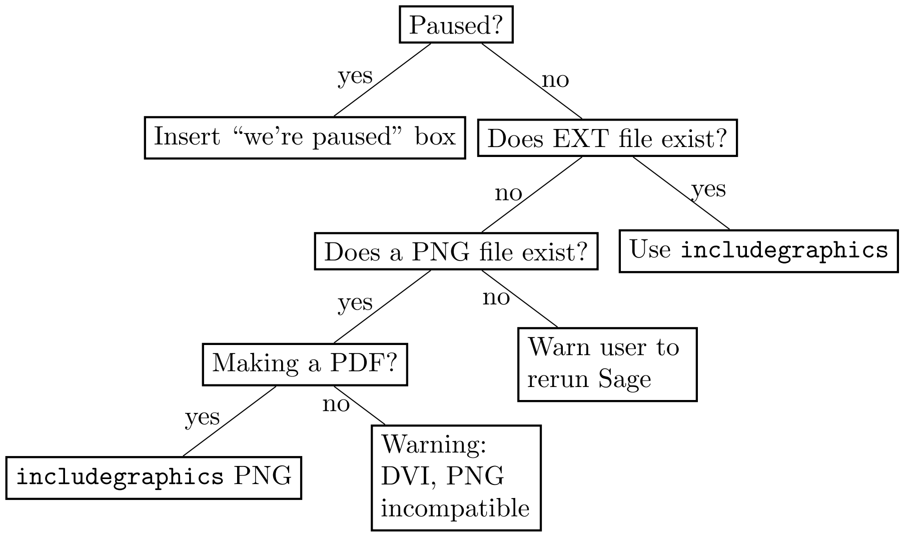


Figure 2: The logic used by the `\ST@inclgrfx` command.

```
\IfFileExists{\ST@plotdir/plot-\theST@plot.#2}%
  {\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.

```
  {\IfFileExists{\ST@plotdir/plot-\theST@plot.png}%
     {\ifpdf
         \ST@inclgrfx{#1}{png}
       \else
        \PackageWarning{sagetex}{Graphics file
        \ST@plotdir/plot-\theST@plot.png on page \thepage\space not
         supported; try using pdflatex. Plot command is}%
      \fi}%
     {\ST@missingfilebox%
     \PackageWarning{sagetex}{Graphics file
     \ST@plotdir/plot-\theST@plot.#2\space on page \thepage\space does not
     exist. Plot command is}%
     \gdef\ST@rerun{x}}}
\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 35. The last bit is some magic from the `verbatim` package manual that makes LaTeX respect line breaks.

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

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

```
\newcommand{\ST@endsfbl}{%
\ST@wsf{except:^^J
 _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.

```
\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.

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

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

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

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

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

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

```
\newenvironment{sagesilent}{\ST@beginsfbl%
\def\verbatim@processline{\ST@wsf{ \the\verbatim@line}}%
\verbatim@start}%
{\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.

```
\newenvironment{sageverbatim}{%
\def\verbatim@processline{\hspace{\sagetexindent}\the\verbatim@line\par}%
\verbatim}%
{\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.

```
\newcommand{\sageexampleincludetextoutput}{False}
\newenvironment{sageexample}{%
  \ST@wsf{%
try:^^J
 _st_.current_tex_line = \the\inputlineno^^J
 _st_.doctest(\theST@inline, r"""}%
  \ST@dodfsetup%
  \ST@wdf{Sage example, line \the\inputlineno::^^J}%
  \begingroup%
  \@bsphack%
  \let\do\@makeother\dospecials%
  \catcode`\^^M\active%
  \def\verbatim@processline{%
    \ST@wsf{\the\verbatim@line}%
    \ST@wdf{\the\verbatim@line}%
  }%
  \verbatim@start%
}
{
 \@esphack%
 \endgroup%
 \ST@wsf{%
   """, globals(), locals(), \sageexampleincludetextoutput)^^Jexcept:^^J
   _st_.goboom(\the\inputlineno)}%
 \ifST@paused%
   \mbox{(Sage\TeX{} is paused)}%
 \else%
   \begin{NoHyper}\ref{@sageinline\theST@inline}\end{NoHyper}%
   \@ifundefined{r@@sageinline\theST@inline}{\gdef\ST@rerun{x}}{}%
 \fi%
 \ST@wdf{}%
 \stepcounter{ST@inline}}
```

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

```
\newcommand{\sagecommandlinetextoutput}{True}
\newlength{\sagecommandlineskip}
\setlength{\sagecommandlineskip}{8pt}
\newenvironment{sagecommandline}{%
  \ST@wsf{%
try:^^J
 _st_.current_tex_line = \the\inputlineno^^J
 _st_.commandline(\theST@cmdline, r"""}%
  \ST@dodfsetup%
  \ST@wdf{Sage commandline, line \the\inputlineno::^^J}%
  \begingroup%
  \@bsphack%
  \let\do\@makeother\dospecials%
  \catcode`\^^M\active%
  \def\verbatim@processline{%
    \ST@wsf{\the\verbatim@line}%
    \ST@wdf{\the\verbatim@line}%
  }%
  \verbatim@start%
}
{
 \@esphack%
 \endgroup%
 \ST@wsf{%
   """, globals(), locals(), \sagecommandlinetextoutput)^^Jexcept:^^J
   _st_.goboom(\the\inputlineno)}%
 \ifST@paused%
   \mbox{(Sage\TeX{} is paused)}%
 \else%
   \begin{NoHyper}\ref{@sagecmdline\theST@cmdline}\end{NoHyper}%
   \@ifundefined{r@@sagecmdline\theST@cmdline}{\gdef\ST@rerun{x}}{}%
 \fi%
 \ST@wdf{}%
 \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.

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

`sagetexunpause` This is the obvious companion to `\sagetexpause`.

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

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

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

```
pyversion = ' '.join(__version__.strip('[').split()[0:2])
from sage.misc.latex import latex
from sage.repl.preparse import preparse
from six import PY3
import sys
import os
import os.path
import hashlib
import traceback
import subprocess
import shutil
import re
from collections import defaultdict
```

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

```
class VersionError(Exception):
    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.

```
class MyDict(defaultdict):
    def __init__(self, *args, **kwargs):
        defaultdict.__init__(self, *args, **kwargs)
        self.default_factory = lambda: -1

    def increment(self, key):
        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.

```
def joinone(j, xs_):
    if len(xs_) >= 2:
        xs = ([xs_[0].rstrip(j)] +
              [x.strip(j) for x in xs_[1:-1]] +
              [xs_[-1].lstrip(j)])
    else:
        xs = xs_
    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.

```
def strip_common_leading_spaces(s):
    lines = s.splitlines()
    lead = min(m.end() for m in
                   [re.match(' *\S', line) for line in lines]
                   if m is not None) - 1
    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.

```
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.

```
  def __init__(self, jobname, version=None, version_check=True):
    if version != pyversion:
      errstr = """versions of .sty and .py files do not match.
{0}.sagetex.sage was generated by sagetex.sty version "{1}", but
is being processed by sagetex.py version "{2}".
Please make sure that TeX is using the sagetex.sty
from your current version of Sage; see
http://doc.sagemath.org/html/en/tutorial/sagetex.html.""".format(jobname,
  version, pyversion)
      if version_check:
        raise VersionError(errstr)
      else:
        print('**** WARNING! Skipping version check for .sty and .py files, and')
      print(errstr)
    if ' ' in jobname:
      jobname = jobname.strip('"')
    self.progress('Processing Sage code for {0}.tex...'.format(jobname))
    self.didinitplot = False
    self.useimagemagick = False
    self.useepstopdf = False
    self.plotdir = 'sage-plots-for-' + jobname + '.tex'
    self.filename = jobname
    self.name = os.path.splitext(jobname)[0]
    autogenstr = """% This file was *autogenerated* from {0}.sagetex.sage with
 % 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.

```
    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.

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

```
    self.scmdtmp = open(self.filename + '.sagetex.scmd.tmp', 'w')
    self.scmdtmp.write(autogenstr)
    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.

```
  def progress(self, t,linebreak=True):
    if linebreak:
      print(t)
    else:
      sys.stdout.write(t)
      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.

```
def initplot(self):
  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.)

```
  if os.path.isdir(self.plotdir):
    shutil.rmtree(self.plotdir)
  os.mkdir(self.plotdir)
  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

```
\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:

```
def inline(self, counter, s, labelname='sageinline'):
    if counter <= self.max_counter_seen[labelname]:
        return
    else:
        self.max_counter_seen.increment(labelname)
    if labelname == 'sageinline':
        self.progress('Inline formula {0} (line {1})'.format(counter, self.current_tex_line))
    elif labelname == 'sagecmdline':
        pass # output message already printed
    else:
        raise ValueError('inline() got a bad labelname "{0}"'.format(labelname))
    self.souttmp.write(r'\newlabel{@' + labelname + str(counter) +
                       '}{{%\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.

```
def savecmd(self, s):
    self.scmdtmp.write(s.rstrip() + "\n")
    begin = self.scmdpos
    end = begin + len(s.splitlines()) - 1
    self.scmdpos = end + 1
    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.

```
def blockbegin(self):
  self.progress('Code block (line {}) begin...'.format(self.current_tex_line), False)
def blockend(self):
  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:])
```

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

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

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

```
    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 `start`ing 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.

```
    outputs = []
    for i, j in zip(starts, starts[1:] + [len(s)]):
        k = i + re.match(prompt, s[i:j]).end()
        try:
            k += [m.end() for m in re.finditer(cont, s[k:j])][-1]
        except IndexError:
            pass
        end = s.find('\n', k)
        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.

```
        ret = []
        for start, end, g in zip(starts, outputs, split):
            lines = g.splitlines()
            cmd = lines[:1]
            for line in lines[1:]:
                has_old_cont = re.match(oldcont, line)
                has_cont = re.match(cont, line)
                if has_old_cont and not has_cont:
                    raise SyntaxError(""" SageTeX no longer supports "..." for line continuation
sagecommandline environments. Use "....:", which matches what the Sage
interpreter uses. See the documentation and example file in
SAGE_ROOT/venv/share/doc/sagetex.""")
                if has_cont:
                    cmd.append(line[has_cont.end():])
            ret.append((start, end, '\n'.join(cmd)))
        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.

```
    def doctest(self, counter, s, globals, locals, include_text_output):
        self.progress('Sage example {0} (line {1})'.format(counter, self.current_tex_line))
        splitup = self.split_sage_cmds(s)
        tex_strs = []
        for i in range(len(splitup)):
            boxname = '@sageinline{}-code{}'.format(counter, i)
            to_tmp = [r'\begin{SaveVerbatim}{' + boxname + '}',
                      s[splitup[i][0]:splitup[i][1]]]
            if include_text_output:
                try:
                    to_tmp.append(s[splitup[i][1]:splitup[i+1][0]])
                except IndexError:
                    to_tmp.append(s[splitup[i][1]:])
            to_tmp.append('\\end{SaveVerbatim}\n')
            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".

```
        tex_strs.append(r'\UseVerbatim{' + boxname + '}')
        try:
            result = eval(preparse(splitup[i][2]), globals, locals)
            tex_strs += [r'\abovedisplayskip=0pt plus 3pt ',
                         r'\abovedisplayshortskip=0pt plus 3pt ',
                         r'\begin{displaymath}',
                         latex(result),
                         r' \end{displaymath}']
        except SyntaxError:
            exec(preparse(splitup[i][2]), globals, locals)
    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.

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

    splitup = self.split_sage_cmds(s)
    skip = r'\vspace{\sagecommandlineskip}'
    tex_strs = [skip]
    lstinput = r'\lstinputlisting[firstline={0},lastline={1},firstnumber={2},style=SageInput{
    for i in range(len(splitup)):
        orig_input = s[splitup[i][0]:splitup[i][1]]
        begin, end = self.savecmd(strip_common_leading_spaces(orig_input.strip('\n')))
        if '#@' in orig_input:
            escapeoption = ',escapeinside={\\#@}{\\^^M}'
        else:
            escapeoption = ''
        tex_strs.append(lstinput.format(begin, end, begin - 2, scmd_fn, escape=escapeoption))
        try:
            result = eval(preparse(splitup[i][2]), globals, locals)
            if text_output:
                begin, end = self.savecmd(str(result))
                tex_strs.append(lstinput.format(begin, end, begin - 2, scmd_fn, escape=''))
            else:
                tex_strs.append(r'\begin{displaymath}' +
                                latex(result) +
                                r'\end{displaymath}')
        except SyntaxError:
            exec(preparse(splitup[i][2]), globals, locals)
    if 'displaymath' not in tex_strs[-1]:
        tex_strs.append(skip)
    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.

```
def plot(self, counter, _p_, format='notprovided', **kwargs):
    if not self.didinitplot:
        self.initplot()
    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]

```
    if format == 'notprovided':
        formats = ['eps', 'pdf']
    else:
        formats = [format]
    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.

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

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

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

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

[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.

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

```
 def goboom(self, line):
   print('\n**** Error in Sage code on line {0} of {1}.tex! Traceback\
follows.'.format(line, self.filename))
   traceback.print_exc()
   print('\n**** Running Sage on {0}.sage failed! Fix {0}.tex and try\
again.'.format(self.filename))
   self.souttmp.close()
   os.remove(self.filename + '.sagetex.sout.tmp')
   self.scmdtmp.close()
   os.remove(self.filename + '.sagetex.scmd.tmp')
   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 `sagetex-run` uses this mechanism to, well, only run Sage when necessary; see section 7.1.

```
def endofdoc(self):
  sagef = open(self.filename + '.sagetex.sage', 'r')
  m = hashlib.md5()
  for line in sagef:
    if not line.startswith((" _st_.goboom",
                            "print('SageT",
                            "_st_.current_tex_line",
                            " _st_.current_tex_line")):
        if PY3:
            m.update(bytearray(line,'utf8'))
        else:
            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.)

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

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

7 Included Python scripts

Here we describe the Python code for SageTEX scripts, for running Sage only if necessary, substituting in Sage outputs to produce a "static" file, and extracting all Sage code from a `.tex` file.

7.1 sagetex-run

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

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

you can do

```
sagetex-run document.sagetex.sage
```

and it will use the MD5 mechanism described in the `endofdoc` macro (page 41). With this, you can set up your editor (TEXShop, TEXWorks, etc) to typeset your document with a script that does

```
pdflatex $1
sagetex-run $1
```

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

```
"""
Given a filename f, examines f.sagetex.sage and f.sagetex.sout and
runs Sage if necessary.
"""

import hashlib
import sys
import os
import re
import subprocess
import shutil
import argparse

def argparser():
    p = argparse.ArgumentParser(description=__doc__.strip())
    p.add_argument('--sage', action='store', default=find_sage(),
                   help="Location of the Sage executable")
    p.add_argument('src', help="Input file name (basename or .sagetex.sage)")
    return p

def find_sage():
    return shutil.which('sage') or 'sage'

def run(args):
    src = args.src
    path_to_sage = args.sage

    if src.endswith('.sagetex.sage'):
        src = src[:-13]
    else:
        src = os.path.splitext(src)[0]
```

```

# Ensure results are output in the same directory as the source files
os.chdir(os.path.dirname(src))
src = os.path.basename(src)

usepackage = r'usepackage(\[.*\])?{sagetex}'
uses_sagetex = False

# If it does not use sagetex, obviously running sage is unnecessary.
if os.path.isfile(src + '.tex'):
    with open(src + '.tex') as texf:
        for line in texf:
            if line.strip().startswith(r'\usepackage') and re.search(usepackage, line):
                uses_sagetex = True
                break
else:
    # The .tex file might not exist if LaTeX output was put to a different
    # directory, so in that case just assume we need to build.
    uses_sagetex = True

if not uses_sagetex:
    print(src + ".tex doesn't seem to use SageTeX, exiting.", file=sys.stderr)
    sys.exit(1)

# if something goes wrong, assume we need to run Sage
run_sage = True
ignore = r"^( _st_.goboom|print\('SageT| ?_st_.current_tex_line)"

try:
    with open(src + '.sagetex.sage', 'r') as sagef:
        h = hashlib.md5()
        for line in sagef:
            if not re.search(ignore, line):
                h.update(bytearray(line,'utf8'))
except IOError:
    print('{0}.sagetex.sage not found, I think you need to typeset {0}.tex first.'
          ''.format(src), file=sys.stderr)
    sys.exit(1)

try:
    with open(src + '.sagetex.sout', 'r') as outf:
        for line in outf:
            m = re.match('%([0-9a-f]+)% md5sum', line)
            if m:
                print('computed md5:', h.hexdigest())
                print('sagetex.sout md5:', m.group(1))
                if h.hexdigest() == m.group(1):
                    run_sage = False
                    break
except IOError:
        pass

    if run_sage:
        print('Need to run Sage on {0}.'.format(src))
        sys.exit(subprocess.call([path_to_sage, src + '.sagetex.sage']))
    else:
        print('Not necessary to run Sage on {0}.'.format(src))

if __name__ == "__main__":
    run(argparser().parse_args())
```

7.2 sagetex-makestatic

Now the sagetex-makestatic:

```
"""
Removes SageTeX macros from 'inputfile' and replaces them with the
Sage-computed results to make a "static" file. You'll need to have run
Sage on 'inputfile' already.

'inputfile' can include the .tex extension or not. If you provide
'outputfile', the results will be written to a file of that name.
Specify '-o' or '--overwrite' to overwrite the file if it exists.

See the SageTeX documentation for more details.
"""
import sys
import time
import os.path
import argparse

from sagetexparse import DeSageTex

def argparser():
    p = argparse.ArgumentParser(description=__doc__.strip())
    p.add_argument('inputfile', help="Input file name (basename or .tex)")
    p.add_argument('outputfile', nargs='?', default=None, help="Output file name")
    p.add_argument('-o', '--overwrite', action="store_true", default=False,
                   help="Overwrite output file if it exists")
    p.add_argument('-s', '--sout', action="store", default=None,
                   help="Location of the .sagetex.sout file")
    return p

def run(args):
    src, dst, overwrite = args.inputfile, args.outputfile, args.overwrite

    if dst is not None and (os.path.exists(dst) and not overwrite):
        print('Error: %s exists and overwrite option not specified.' % dst,
              file=sys.stderr)
        sys.exit(1)

    src, ext = os.path.splitext(src)
    texfn = src + '.tex'
    soutfn = args.sout if args.sout is not None else src + '.sagetex.sout'
```

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

```
    desagetexed = DeSageTex(texfn, soutfn)
```

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!

```
    header = ("%% SageTeX commands have been automatically removed from this file and\n"
              "%% replaced with plain LaTeX. Processed %s.\n"
              "" % time.strftime('%a %d %b %Y %H:%M:%S', time.localtime()))

    if dst is not None:
        dest = open(dst, 'w')
    else:
        dest = sys.stdout

    dest.write(header)
    dest.write(desagetexed.result)

if __name__ == "__main__":
    run(argparser().parse_args())
```

7.3 sagetex-extract

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

```
"""
Extracts Sage code from `inputfile'.

`inputfile' can include the .tex extension or not. If you provide
`outputfile', the results will be written to a file of that name,
otherwise the result will be printed to stdout.

Specify `-o' or `--overwrite' to overwrite the file if it exists.

See the SageTeX documentation for more details.
"""
import sys
import time
import os.path
import argparse

from sagetexparse import SageCodeExtractor

def argparser():
    p = argparse.ArgumentParser(description=__doc__.strip())
    p.add_argument('inputfile', help="Input file name (basename or .tex)")
    p.add_argument('outputfile', nargs='?', default=None, help="Output file name")
    p.add_argument('-o', '--overwrite', action="store_true", default=False,
                   help="Overwrite output file if it exists")
    p.add_argument('--no-inline', action="store_true", default=False,
                   help="Extract code only from Sage environments")
    return p

def run(args):
    src, dst, overwrite = args.inputfile, args.outputfile, args.overwrite

    if dst is not None and (os.path.exists(dst) and not overwrite):
        print('Error: %s exists and overwrite option not specified.' % dst,
              file=sys.stderr)
        sys.exit(1)

    src, ext = os.path.splitext(src)
    sagecode = SageCodeExtractor(src + '.tex', inline=not args.no_inline)
    header = ("#> This file contains Sage code extracted from %s%s.\n"
              "#> Processed %s.\n"
              "" % (src, ext, time.strftime('%a %d %b %Y %H:%M:%S', time.localtime())))

    if dst is not None:
        dest = open(dst, 'w')
    else:
        dest = sys.stdout

    dest.write(header)
    dest.write(sagecode.result)

if __name__ == "__main__":
    run(argparser().parse_args())
```

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.

```
import sys
import os
import textwrap
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.

```
def skipToMatching(opener, closer):
  nest = nestedExpr(opener, closer)
  return originalTextFor(nest)

curlybrackets = skipToMatching('{', '}')
squarebrackets = skipToMatching('[', ']')
```

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

```
sagemacroparser = r'\sage' + curlybrackets('code')
sagestrmacroparser = r'\sagestr' + curlybrackets('code')
sageplotparser = (r'\sageplot'
                 + Optional(squarebrackets)('opts')
                 + Optional(squarebrackets)('format')
                 + curlybrackets('code'))
sagetexpause = Literal(r'\sagetexpause')
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!

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

A label line looks like

`\newlabel{@sageinline`⟨*integer*⟩`}{{`⟨*bunch of LaTeX code*⟩`}{}{}{}{}}`

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.

```
    parselabel = (r'\newlabel{@sageinline'
                 + Word(nums)('num')
                 + '}{'
                 + curlybrackets('result')
                 + '{}{}{}{}}')
```

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

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

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

```
    try:
      OneOrMore(parselabel).parseFile(fn)
    except IOError:
      print('Error accessing {}; exiting. Does your .sout file exist?'.format(fn))
      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.

```
  def newlabel(self, s, l, t):
    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.

```
class DeSageTex():
  def __init__(self, texfn, soutfn):
    self.sagen = 0
    self.plotn = 0
    self.fn = os.path.basename(texfn)
    self.sout = SoutParser(soutfn)
```

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.

```
    strmacro = sagestrmacroparser
    smacro = sagemacroparser
    smacro.setParseAction(self.sage)
    strmacro.setParseAction(self.sage)
```

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

```
    usepackage = (r'\usepackage'
                  + Optional(squarebrackets)
                  + '{sagetex}')
    usepackage.setParseAction(replaceWith(r"""% "\usepackage{sagetex}" line was here:
\RequirePackage{verbatim}
\RequirePackage{graphicx}
\newcommand{\sagetexpause}{\relax}
\newcommand{\sagetexunpause}{\relax}"""))
```

Parse `\sageplot` macros.

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

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

```
    beginorend = oneOf('begin end')
    blockorverb = 'sage' + oneOf('block verbatim')
    blockorverb.setParseAction(replaceWith('verbatim'))
    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.

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

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

```
    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.

```
    doit = smacro | senv | ssilent | usepackage | splot | stexindent |strmacro
    doit.ignore('%' + restOfLine)
    doit.ignore(r'\begin{verbatim}' + SkipTo(r'\end{verbatim}'))
    doit.ignore(r'\begin{comment}' + SkipTo(r'\end{comment}'))
    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.

```
    str = ''.join(open(texfn, 'r').readlines())
    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.

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

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

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

```
class SageCodeExtractor():
  def __init__(self, texfn, inline=True):
    smacro = sagemacroparser
    smacro.setParseAction(self.macroout)

    splot = sageplotparser
    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.

```
    env_names = oneOf('sageblock sageverbatim sagesilent')
    senv = r'\begin{' + env_names('env') + '}' + SkipTo(
           r'\end{' + matchPreviousExpr(env_names) + '}')('code')
    senv.leaveWhitespace()
    senv.setParseAction(self.envout)

    spause = sagetexpause
    spause.setParseAction(self.pause)

    sunpause = sagetexunpause
    sunpause.setParseAction(self.unpause)

    if inline:
        doit = smacro | splot | senv | spause | sunpause
    else:
        doit = senv | spause | sunpause
    doit.ignore('%' + restOfLine)

    str = ''.join(open(texfn, 'r').readlines())
    self.result = ''

    doit.transformString(str)

  def macroout(self, s, l, t):
    self.result += '#> \\sage{} from line %s\n' % lineno(l, s)
    self.result += textwrap.dedent(t.code[1:-1]) + '\n\n'

  def plotout(self, s, l, t):
    self.result += '#> \\sageplot{} from line %s:\n' % lineno(l, s)
    if t.format != '':
      self.result += '#> format: %s' % t.format[0][1:-1] + '\n'
    self.result += textwrap.dedent(t.code[1:-1]) + '\n\n'

  def envout(self, s, l, t):
    self.result += '#> %s environment from line %s:' % (t.env,
      lineno(l, s))
    self.result += textwrap.dedent(''.join(t.code)) + '\n'

  def pause(self, s, l, t):
    self.result += ('#> SageTeX (probably) paused on input line %s.\n\n' %
                    (lineno(l, s)))

  def unpause(self, s, l, t):
    self.result += ('#> SageTeX (probably) unpaused on input line %s.\n\n' %
                    (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.

```
from __future__ import print_function
import json
import sys
import time
import re
import urllib
import hashlib
import os
import os.path
import shutil
import getopt
from contextlib import closing

#########################################################################
# You can provide a filename here and the script will read your login   #
# information from that file. The format must be:                       #
#                                                                       #
# server = 'http://foo.com:8000'                                        #
# username = 'my_name'                                                  #
# password = 's33krit'                                                  #
#                                                                       #
# You can omit one or more of those lines, use " quotes, and put hash   #
# marks at the beginning of a line for comments. Command-line args      #
# take precedence over information from the file.                       #
#########################################################################
login_info_file = None       # e.g. '/home/foo/Private/sagetex-login.txt'


usage = """Process a SageTeX-generated .sage file using a remote Sage server.

Usage: {0} [options] inputfile.sage

Options:

    -h, --help:         print this message
    -s, --server:       the Sage server to contact
    -u, --username:     username on the server
    -p, --password:     your password
    -f, --file:         get login information from a file

If the server does not begin with the four characters `http', then
`https://' will be prepended to the server name.
```

```

You can hard-code the filename from which to read login information into
the remote-sagetex script. Command-line arguments take precedence over
the contents of that file. See the SageTeX documentation for formatting
details.

If any of the server, username, and password are omitted, you will be
asked to provide them.

See the SageTeX documentation for more details on usage and limitations
of remote-sagetex.""".format(sys.argv[0])

server, username, password = (None,) * 3

try:
    opts, args = getopt.getopt(sys.argv[1:], 'hs:u:p:f:',
                    ['help', 'server=', 'user=', 'password=', 'file='])
except getopt.GetoptError as err:
    print(str(err), usage, sep='\n\n')
    sys.exit(2)

for o, a in opts:
    if o in ('-h', '--help'):
        print(usage)
        sys.exit()
    elif o in ('-s', '--server'):
        server = a
    elif o in ('-u', '--user'):
        username = a
    elif o in ('-p', '--password'):
        password = a
    elif o in ('-f', '--file'):
        login_info_file = a

if len(args) != 1:
    print('Error: must specify exactly one file. Please specify options first.',
          usage, sep='\n\n')
    sys.exit(2)

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 traceback 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.

```
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.

```
def parsedotsage(fn):
    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.

```
        inline = re.compile(r" _st_.inline\((?P<num>\d+), (?P<code>.*)\)")
        plot = re.compile(r" _st_.plot\((?P<num>\d+), (?P<code>.*)\)")
        goboom = re.compile(r" _st_.goboom\((?P<num>\d+)\)")
        pausemsg = re.compile(r"print.'(?P<msg>SageTeX (un)?paused.*)'")
        blockbegin = re.compile(r"_st_.blockbegin\(\)")
        ignore = re.compile(r"(try:)|(except):")
        in_comment = False
        in_block = False
        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`.
- `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.

```
        for line in f.readlines():
            if line.startswith('"""'):
                in_comment = not in_comment
            elif not in_comment:
                m = pausemsg.match(line)
                if m:
                    cmds.append({'type': 'pause',
                                 'msg': m.group('msg')})
                m = inline.match(line)
                if m:
                    cmds.append({'type': 'inline',
                                 'num': m.group('num'),
                                'code': m.group('code')})
                m = plot.match(line)
                if m:
                    cmds.append({'type': 'plot',
                                 'num': m.group('num'),
                                 '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.

```
                m = goboom.match(line)
                if m:
                    cmds[-1]['goboom'] = m.group('num')
                    if in_block:
                        in_block = False
                if in_block and not ignore.match(line):
                    cmds[-1]['code'] += line
                if blockbegin.match(line):
                    cmds.append({'type': 'block',
                                 'code': ''})
                    in_block = True
    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`.

```
debug = False
class RemoteSage:
    def __init__(self, server, user, password):
        self._srv = server.rstrip('/')
        sep = '___S_A_G_E___'
        self._response = re.compile('(?P<header>.*)' + sep +
                                    '\n*(?P<output>.*)', re.DOTALL)
        self._404 = re.compile('404 Not Found')
        self._session = self._get_url('login',
                                    urllib.urlencode({'username': user,
                                    'password':
                                    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.

```
        self._codewrap = """try:
{{0}}
except:
    print('{0}')
    traceback.print_exc()""".format(traceback_str)
        self.do_block("""
    import traceback
    def __st_plot__(counter, _p_, format='notprovided', **kwargs):
        if format == 'notprovided':
            formats = ['eps', 'pdf']
        else:
            formats = [format]
        for fmt in formats:
            plotfilename = 'plot-%s.%s' % (counter, fmt)
            _p_.save(filename=plotfilename, **kwargs)""")

    def _encode(self, d):
        return 'session={0}&'.format(self._session) + urllib.urlencode(d)

    def _get_url(self, action, u):
        with closing(urllib.urlopen(self._srv + '/simple/' + action +
                                    '?' + u)) as h:
            data = self._response.match(h.read())
            result = json.loads(data.group('header'))
            result['output'] = data.group('output').rstrip()
        return result

    def _get_file(self, fn, cell, ofn=None):
        with closing(urllib.urlopen(self._srv + '/simple/' + 'file' + '?' +
                     self._encode({'cell': cell, 'file': fn}))) as h:
            myfn = ofn if ofn else fn
            data = h.read()
            if not self._404.search(data):
                with open(myfn, 'w') as f:
                    f.write(data)
            else:
                print('Remote server reported {0} could not be found:'.format(
                      fn))
                print(data)
```

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

```
    def _do_cell(self, code):
        realcode = self._codewrap.format(code)
        result = self._get_url('compute', self._encode({'code': realcode}))
        if result['status'] == 'computing':
            cell = result['cell_id']
            while result['status'] == 'computing':
                sys.stdout.write('working...')
                sys.stdout.flush()
                time.sleep(10)
                result = self._get_url('status', self._encode({'cell': cell}))
        if debug:
            print('cell: <<<', realcode, '>>>', 'result: <<<',
                result['output'], '>>>', sep='\n')
        return result

    def do_inline(self, code):
        return self._do_cell(' print(latex({0}))'.format(code))

    def do_block(self, code):
        result = self._do_cell(code)
        for fn in result['files']:
            self._get_file(fn, result['cell_id'])
        return result

    def do_plot(self, num, code, plotdir):
        result = self._do_cell(' __st_plot__({0}, {1})'.format(num, code))
        for fn in result['files']:
            self._get_file(fn, result['cell_id'], os.path.join(plotdir, fn))
        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.

```
    def close(self):
        sys.stdout.write('Logging out of {0}...'.format(server))
        sys.stdout.flush()
        self._get_url('logout', self._encode({}))
        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.

```
def do_plot_setup(plotdir):
    printc('initializing plots directory...')
    if os.path.isdir(plotdir):
        shutil.rmtree(plotdir)
    os.mkdir(plotdir)
    return True

did_plot_setup = False
plotdir = 'sage-plots-for-' + jobname + '.tex'

def labelline(n, s):
    return r'\newlabel{@sageinline' + str(n) + '}{{' + s  + '}{}{}{}{}}\n'

def printc(s):
    print(s, end='')
    sys.stdout.flush()

error = re.compile("(^" + traceback_str + ")|(^Syntax Error:)", re.MULTILINE)

```

```
def check_for_error(string, line):
    if error.search(string):
        print("""
**** Error in Sage code on line {0} of {1}.tex!
{2}
**** Running Sage on {1}.sage failed! Fix {1}.tex and try again.""".format(
              line, jobname, string))
        sys.exit(1)
```

Now let's actually start doing stuff.

```
print('Processing Sage code for {0}.tex using remote Sage server.'.format(
      jobname))

if login_info_file:
    with open(login_info_file, 'r') as f:
        print('Reading login information from {0}.'.format(login_info_file))
        get_val = lambda x: x.split('=')[1].strip().strip('\'"')
        for line in f:
            print(line)
            if not line.startswith('#'):
                if line.startswith('server') and not server:
                    server = get_val(line)
                if line.startswith('username') and not username:
                    username = get_val(line)
                if line.startswith('password') and not password:
                    password = get_val(line)

if not server:
    server = raw_input('Enter server: ')

if not server.startswith('http'):
    server = 'https://' + server

if not username:
    username = raw_input('Enter username: ')

if not password:
    from getpass import getpass
    password = getpass('Please enter password for user {0} on {1}: '.format(
        username, server))

printc('Parsing {0}.sage...'.format(jobname))
cmds = parsedotsage(jobname + '.sage')
print('done.')

sout = '% This file was *autogenerated* from the file {0}.sage.\n'.format(
    os.path.splitext(jobname)[0])

printc('Logging into {0} and starting session...'.format(server))
with closing(RemoteSage(server, username, password)) as sage:
    print('done.')
    for cmd in cmds:
        if cmd['type'] == 'inline':
            printc('Inline formula {0}...'.format(cmd['num']))
            result = sage.do_inline(cmd['code'])
            check_for_error(result['output'], cmd['goboom'])
            sout += labelline(cmd['num'], result['output'])
            print('done.')
        if cmd['type'] == 'block':
            printc('Code block begin...')
            result = sage.do_block(cmd['code'])
            check_for_error(result['output'], cmd['goboom'])
            print('end.')
        if cmd['type'] == 'plot':
            printc('Plot {0}...'.format(cmd['num']))
            if not did_plot_setup:
                did_plot_setup = do_plot_setup(plotdir)
            result = sage.do_plot(cmd['num'], cmd['code'], plotdir)
            check_for_error(result['output'], cmd['goboom'])
            print('done.')
        if cmd['type'] == 'pause':
            print(cmd['msg'])
        if int(time.time()) % 2280 == 0:
            printc('Unscheduled offworld activation; closing iris...')
            time.sleep(1)
            print('end.')

with open(jobname + '.sage', 'r') as sagef:
    h = hashlib.md5()
    for line in sagef:
        if (not line.startswith(' _st_.goboom') and
            not line.startswith("print('SageT")):
            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.

```
    sout += """%{0}% md5sum of corresponding .sage file
{1} (minus "goboom" and pause/unpause lines)
""".format(h.hexdigest(), '%')

printc('Writing .sout file...')
with open(jobname + '.sout', 'w') as soutf:
    soutf.write(sout)
    print('done.')
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:

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.