The SageTEX package*

Dan Drake and others†

June 20, 2009

1 Introduction

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

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

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

and it will produce

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

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

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

*This document corresponds to SageTEX v2.2.1, dated 2009/06/17.

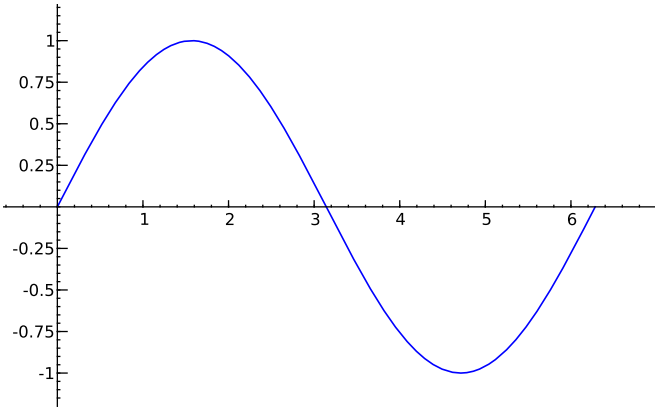
†Author's website: `mathsci.kaist.ac.kr/~drake/`.

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{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.)

Till Tantau has some good commentary on the use of graphics in section 6 of the PGF manual. 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

To install SageTEX, you need to do two things: make SageTEX known to Sage, and to LATEX. There are two basic methods to do those two things.

In what follows, "`$SAGE_ROOT`" refers to the root directory of your Sage installation.

2.1 As a Sage spkg

The easiest way to install SageTEX is by using Sage's own spkg installation facility; visit the optional packages page and run `sage -i` with the appropriate file name. This will let Sage know about SageTEX; you still need to let LATEX know about it.

The simplest way to "install" SageTEX for LATEX is to copy the file `sagetex.sty` from `$SAGE_ROOT/local/share/texmf` to the same directory as your document. This will always work, as LATEX always searches the current directory for files.

Rather than make lots of copies of `sagetex.sty`, you can keep it (and the rest of the SageTEX documentation) in a `texmf` directory. The easiest thing to do is to create a `texmf` directory in your home directory and use the `texhash` utility so that your TEX system can find the new files. See `www.tex.ac.uk/cgi-bin/texfaq2html?label=privinst` which describes the basic ideas, and also `www.tex.ac.uk/cgi-bin/texfaq2html?label=what-TDS` which has some information specific to MiKTEX. Linux/Unix users can use `$HOME/texmf` and users of MacTEX should use `$HOME/Library/texmf`.

To copy the files that LATEX needs into your `texmf` directory, simply do

```
cp -r $SAGE_ROOT/local/share/texmf/* HOMEPREFIX/texmf/
```

where `HOMEPREFIX` is the appropriate prefix for your own `texmf` tree. Then you need to make TEX aware of the new files by running

```
texhash HOMEPREFIX/texmf/
```

2.2 From CTAN

You can also get SageTEX from its CTAN page. This is not the recommended way to get SageTEX, but it will work.

If you get SageTEX from CTAN, you will need to make the `sagetex.sty` file available to LATEX using any of the methods described above, and you will also

need to make `sagetex.py` known to Sage. You can either keep a copy of that file in the same directory as your document or put it where Sage will find it. You can use the `$SAGE_PATH` environment variable (which is analogous to the `$PYTHONPATH` variable) to tell Sage where the file is, or manually copy `sagetex.py` into `$SAGE_ROOT/local/lib/python/site-packages`.

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 `sagetex.sty` and `sagetex.py` known to LATEX and Sage using any of the methods described above (although note that TEXShop includes copies of these files for you). You also might need to edit the `sage.engine` script to reflect the location of 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. 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 4.3 and section 4.4 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` file. That file contains LATEX code that, when you run LATEX on your source file again, will pull in all the results of Sage's computation.

All you really need to know is that to typeset your document, you need to run LATEX, then run Sage, then run LATEX again. You can even "run Sage" on a

computer that doesn't have Sage installed by using the `remote-sagetex.py` script; see section 5. Whenever this manual says "run Sage", you can either directly run Sage, or use the `remote-sagetex.py` script.

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*⟩`.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!

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

3.1 Inline Sage

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

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

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

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

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

in Sage.

Note that since LaTeX will do macro expansion on whatever you give to `\sage`, you can mix LaTeX variables and Sage variables! If you have defined the Sage

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

variable `foo` to be 12 (using, say, the `sageblock` environment), then you can do something like this:

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

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

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

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

3.2 Graphics and plotting

`\sageplot` `\sageplot[`⟨*ltx opts*⟩`][`⟨*fmt*⟩`]{`⟨*graphics obj*⟩, ⟨*keyword args*⟩`}` 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.

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}
```

[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.

Option	Description
⟨*ltx options*⟩	Any text here is passed directly into the optional arguments (between the square brackets) of an `\includegraphics` command. If not specified, "`width=.75\textwidth`" will be used.
⟨*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.
⟨*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.

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.

If you ask for, say, a PNG file, 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

[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.

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!

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.

Because of this, when producing 3D plots with `\sageplot`, *you must specify a file format.* The PNG format is compressed and lossless and is by far the best choice, so use that whenever possible. (Right now, it is always possible.) If you do not specify a file format, or specify one that Tachyon does not understand, it will produce files in the Targa format with an incorrect extension and LaTeX (both `latex` and `pdflatex`) will be profoundly confused. Don't do that.

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. (Running `pdflatex` on the same file will work, since PDF files can include PNG files.)

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

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

3.2.2 But that's not good enough!

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

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

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

3.3 Verbatim-like environments

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

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

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

and then anytime later write in your source file

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

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

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

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

`comment` Logically, we now need an environment that neither typesets nor executes your Sage code... but the `verbatim` package, which is always loaded when using

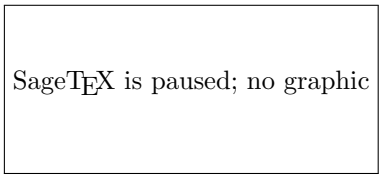
SageTEX, provides such an environment: `comment`. Another way to do this is to put stuff between `\iffalse` and `\fi`.

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

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:



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 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.

4.2 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.

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

What can you do when sending a LaTeX document that uses SageTEX to a colleague who doesn't use Sage?[4] The best option is to bring your colleague into the light and get him or her using Sage! But this may not be feasible, because some (most?) mathematicians are fiercely crotchety about their choice of computer algebra system, or you may be sending a paper to a journal or the arXiv, and such places will not run Sage just so they can typeset your paper—at least not until Sage is much closer to its goal of world domination.

How can you send your SageTEX-enabled document to someone else who doesn't use Sage? The easiest way is to simply include with your document the following files:

[4] Or who cannot use Sage, since currently SageTEX is not very useful on Windows.

1. `sagetex.sty`
2. the generated `.sout` file
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.

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.[5] 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. 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.

4.4 Extracting the Sage code from a document

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.

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

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

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.

```
\RequirePackage{verbatim}
```

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.

```
\RequirePackage{makecmds}
\RequirePackage{ifpdf}
\RequirePackage{ifthen}
```

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

```
\newcounter{ST@inline}
\newcounter{ST@plot}
```

```
\setcounter{ST@inline}{0}
\setcounter{ST@plot}{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.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 24 for discussion of file extensions. The "`(\jobname.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.

```
\newcommand{\ST@wsf}[1]{\immediate\write\ST@sf{#1}}%
\ST@wsf{# This file (\jobname.sage) was *autogenerated* from the file \jobname.tex.}%
\ST@wsf{import sagetex}%
\ST@wsf{_st_ = sagetex.SageTeXProcessor('\jobname')}}%
```

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}}}
```

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.sout}{}{\AtEndDocument{\PackageWarningNoLine{sagetex}%
  {`final' option provided, but \jobname.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. Then, process the options.

```
\DeclareOption{epstopdf}{%
\AtBeginDocument{%
\@ifundefined{ST@final}{%
  \ST@wsf{_st_.useepstopdf = True}}{}}}
\ProcessOptions\relax
```

The `\relax` is a little incantation suggested by the "LaTeX 2_ε 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.sout}{}{\typeout{No file \jobname.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` macro

`\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.) We can use `^^J` to put linebreaks into the `.sage` file, but LaTeX wants to put a space after that, which is why we don't put the "except" on its own line here in the source.

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

The `inline` function of the Python module is documented on page 24. 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}}
```

`\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][width=.75\textwidth]{%
  \@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. We define a default size of 3/4 the textwidth, which seems reasonable. (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_.plot(\theST@plot, format='#2', _p_=#3)^^Jexcept:^^J
 _st_.goboom(\the\inputlineno)}%
```

The Python plot function is documented on page 25.

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:

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

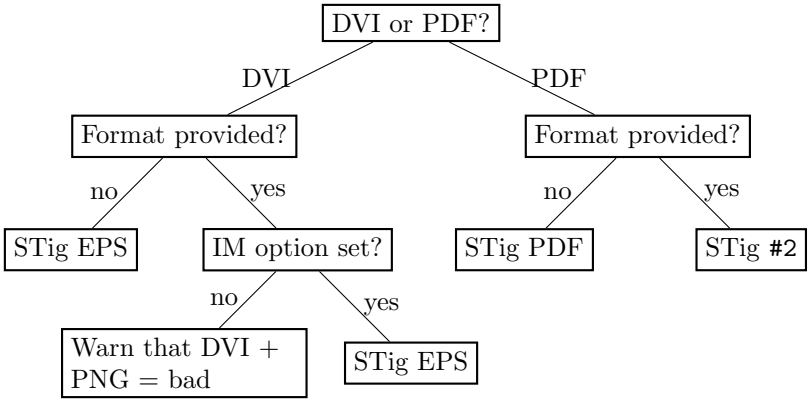


Figure 1: The logic tree that `\sageplot` uses to decide whether to run `\includegraphics` or to yell at the user. "Format" is the `#2` argument to `\sageplot`, "STig ext" means a call to `\ST@inclgrfx` with "ext" as the second argument, and "IM" is Imagemagick.

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.)

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

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

```
    {\@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}}}%
\fi
```

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

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

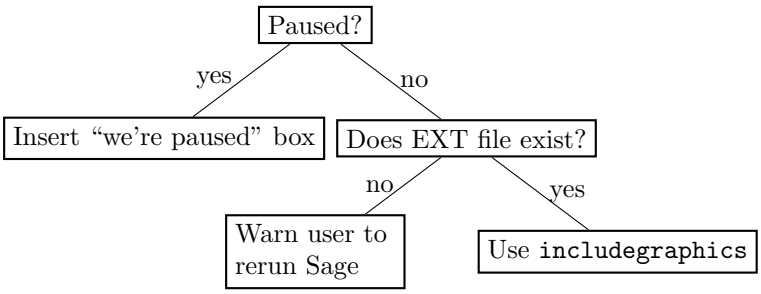


Figure 2: The logic used by the `\ST@inclgrfx` command.

`\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
  \IfFileExists{\ST@plotdir/plot-\theST@plot.#2}%
    {\includegraphics[#1]{\ST@plotdir/plot-\theST@plot.#2}}%
```

If the file doesn't exist, 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.

```
    {\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 25. The last bit is some magic from the `verbatim` package manual that makes LATEX respect line breaks.

```
\newcommand{\ST@beginsfbl}{%
  \@bsphack\ST@wsf{%
_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.

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 suppressed 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.

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

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: the `sagetex.py` file is intended to be used as a module and doesn't do anything useful when called directly, so if someone does that, warn them. We do this right away so that we print this and exit before trying to import any Sage modules; that way, this error message gets printed whether you run the script with Sage or with Python.

```
import sys
if __name__ == "__main__":
  print("""This file is part of the SageTeX package.
It is not meant to be called directly.

This file will be automatically used by Sage scripts generated from a
LaTeX document using the SageTeX package.""")
  sys.exit()
```

Import what we need:

```
from sage.misc.latex import latex
import os
import os.path
import hashlib
import traceback
import subprocess
import shutil
```

We define a 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():
  def __init__(self, jobname):
    self.progress('Processing Sage code for %s.tex...' % jobname)
    self.didinitplot = False
```

```
149     self.useimagemagick = False
150     self.useepstopdf = False
151     self.plotdir = 'sage-plots-for-' + jobname + '.tex'
152     self.filename = jobname
```

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.

```
153     self.souttmp = open(self.filename + '.sout.tmp', 'w')
154     s = '% This file was *autogenerated* from the file ' + \
155         os.path.splitext(jobname)[0] + '.sage.\n'
156     self.souttmp.write(s)
```

`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.

```
157   def progress(self, t,linebreak=True):
158     if linebreak:
159       print(t)
160     else:
161       sys.stdout.write(t)
162       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.

```
163   def initplot(self):
164     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.

```
165     if os.path.isdir(self.plotdir):
166       shutil.rmtree(self.plotdir)
167     os.mkdir(self.plotdir)
168     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 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`.

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 very short function:

```
169  def inline(self, counter, s):
170    self.progress('Inline formula %s' % counter)
171    self.souttmp.write('\\newlabel{@sageinline' + str(counter) + '}{{' + \
172                 latex(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.

`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.

```
173  def blockbegin(self):
174    self.progress('Code block begin...', False)
175  def blockend(self):
176    self.progress('end')
```

`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.

```
177  def plot(self, counter, _p_, format='notprovided', **kwargs):
178    if not self.didinitplot:
179      self.initplot()
180    self.progress('Plot %s' % counter)
```

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.[6]

```
181    if format == 'notprovided':
182      formats = ['eps', 'pdf']
183    else:
184      formats = [format]
185    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.

```
186      if fmt == 'pdf' and self.useepstopdf:
187        epsfile = os.path.join(self.plotdir, 'plot-%s.eps' % counter)
```

[6] 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.

```
188         self.progress('Calling epstopdf to convert plot-%s.eps to PDF' % \
189             counter)
190         subprocess.check_call(['epstopdf', epsfile])
191         continue
192       plotfilename = os.path.join(self.plotdir, 'plot-%s.%s' % (counter, fmt))
193       _p_.save(filename=plotfilename, **kwargs)
```

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

```
194       if format != 'notprovided' and self.useimagemagick:
195         self.progress('Calling Imagemagick to convert plot-%s.%s to EPS' % \
196           (counter, format))
197         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.

```
198   def toeps(self, counter, ext):
199     subprocess.check_call(['convert',\
200       '%s/plot-%s.%s' % (self.plotdir, counter, ext), \
201       '%s/plot-%s.eps' % (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` file.

```
202   def goboom(self, line):
203     print('\n**** Error in Sage code on line %s of %s.tex! Traceback\
204  follows.' % (line, self.filename))
205     traceback.print_exc()
206     print('\n**** Running Sage on %s.sage failed! Fix %s.tex and try\
207  again.' % ((self.filename,) * 2))
208     self.souttmp.close()
209     os.remove(self.filename + '.sout.tmp')
210     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 calls to `goboom` and pause/unpause lines, so we will strip those lines out. What we do below is exactly equivalent to running

```
egrep '^( _st_.goboom|print .SageT)' filename.sage | md5sum
```

in a shell.

```
211  def endofdoc(self):
212    sagef = open(self.filename + '.sage', 'r')
213    m = hashlib.md5()
214    for line in sagef:
215      if line[0:12] != " _st_.goboom" and line[0:12] != "print 'SageT":
216        m.update(line)
217    s = '%' + m.hexdigest() + '% md5sum of corresponding .sage file\
218 (minus "goboom" and pause/unpause lines)\n'
219    self.souttmp.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.

```
220    self.souttmp.close()
221    os.rename(self.filename + '.sout.tmp', self.filename + '.sout')
222    self.progress('Sage processing complete. Run LaTeX on %s.tex again.' %\
223              self.filename)
```

7 Included Python scripts

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

7.1 makestatic.py

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

```
224 import sys
225 import time
226 import getopt
227 import os.path
228 from sagetexparse import DeSageTex
229
230 def usage():
231   print("""Usage: %s [-h|--help] [-o|--overwrite] inputfile [outputfile]
232
233 Removes SageTeX macros from ‘inputfile’ and replaces them with the
234 Sage-computed results to make a "static" file. You’ll need to have run
235 Sage on ‘inputfile’ already.
236
237 ‘inputfile’ can include the .tex extension or not. If you provide
238 ‘outputfile’, the results will be written to a file of that name.
239 Specify ‘-o’ or ‘--overwrite’ to overwrite the file if it exists.
240
241 See the SageTeX documentation for more details.""" % sys.argv[0])
242
243 try:
244   opts, args = getopt.getopt(sys.argv[1:], ’ho’, [’help’, ’overwrite’])
245 except getopt.GetoptError, err:
246   print str(err)
247   usage()
248   sys.exit(2)
249
250 overwrite = False
251 for o, a in opts:
252   if o in (’-h’, ’--help’):
253     usage()
254     sys.exit()
255   elif o in (’-o’, ’--overwrite’):
256     overwrite = True
257
258 if len(args) == 0 or len(args) > 2:
259   print(’Error: wrong number of arguments. Make sure to specify options first.\n’)
260   usage()
261   sys.exit(2)
262
263 if len(args) == 2 and (os.path.exists(args[1]) and not overwrite):
264   print(’Error: %s exists and overwrite option not specified.’ % args[1])
265   sys.exit(1)
266
267 src, ext = os.path.splitext(args[0])
```

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

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

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

```
header = """\
%% SageTeX commands have been automatically removed from this file and
%% replaced with plain LaTeX. Processed %s.

""" % time.strftime('%a %d %b %Y %H:%M:%S', time.localtime())

if len(args) == 2:
  dest = open(args[1], 'w')
else:
  dest = sys.stdout

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

7.2 extractsagecode.py

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

```
import sys
import time
import getopt
import os.path
from sagetexparse import SageCodeExtractor

def usage():
  print("""Usage: %s [-h|--help] [-o|--overwrite] inputfile [outputfile]

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.""" % sys.argv[0])

try:
  opts, args = getopt.getopt(sys.argv[1:], 'ho', ['help', 'overwrite'])
except getopt.GetoptError, err:
  print str(err)
  usage()
  sys.exit(2)

```

```
overwrite = False
for o, a in opts:
  if o in ('-h', '--help'):
    usage()
    sys.exit()
  elif o in ('-o', '--overwrite'):
    overwrite = True

if len(args) == 0 or len(args) > 2:
  print('Error: wrong number of arguments. Make sure to specify options first.\n')
  usage()
  sys.exit(2)

if len(args) == 2 and (os.path.exists(args[1]) and not overwrite):
  print('Error: %s exists and overwrite option not specified.' % args[1])
  sys.exit(1)

src, ext = os.path.splitext(args[0])
sagecode = SageCodeExtractor(src)
header = """\
# This file contains Sage code extracted from %s%s.
# Processed %s.

""" % (src, ext, time.strftime('%a %d %b %Y %H:%M:%S', time.localtime()))

if len(args) == 2:
  dest = open(args[1], 'w')
else:
  dest = sys.stdout

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

7.3 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
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)
  nest.setParseAction(lambda l, s, t: l[s:getTokensEndLoc()])
```

```
  return nest

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

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

```
sagemacroparser = r'\sage' + 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 %s; exiting. Does your .sout file exist?' % 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.

```
371   def newlabel(self, s, l, t):
372     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.

```
373 class DeSageTex():
374   def __init__(self, fn):
375     self.sagen = 0
376     self.plotn = 0
377     self.fn = fn
378     self.sout = SoutParser(fn + '.sout')
```

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

```
379     smacro = sagemacroparser
380     smacro.setParseAction(self.sage)
```

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

```
381     usepackage = (r'\usepackage'
382                   + Optional(squarebrackets)
383                   + '{sagetex}')
384     usepackage.setParseAction(replaceWith(r"""% "\usepackage{sagetex}" line was here:
385 \RequirePackage{verbatim}
386 \RequirePackage{graphicx}
387 \newcommand{\sagetexpause}{\relax}
388 \newcommand{\sagetexunpause}{\relax}"""))
```

Parse `\sageplot` macros.

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

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

```
391     beginorend = oneOf('begin end')
392     blockorverb = 'sage' + oneOf('block verbatim')
393     blockorverb.setParseAction(replaceWith('verbatim'))
394     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.

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

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

```
398     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.

```
399     doit = smacro | senv | ssilent | usepackage | splot | stexindent
400     doit.ignore('%' + restOfLine)
401     doit.ignore(r'\begin{verbatim}' + SkipTo(r'\end{verbatim}'))
402     doit.ignore(r'\begin{comment}' + SkipTo(r'\end{comment}'))
403     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.

```
404     str = ''.join(open(fn + '.tex', 'r').readlines())
405     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.

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

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

```
409   def plot(self, s, l, t):
410     self.plotn += 1
411     if len(t.opts) == 0:
412       opts = r'[width=.75\textwidth]'
413     else:
414       opts = t.opts[0]
415     return (r'\includegraphics%s{sage-plots-for-%s.tex/plot-%s}' %
416       (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.

```
417 class SageCodeExtractor():
418   def __init__(self, fn):
419     smacro = sagemacroparser
420     smacro.setParseAction(self.macroout)
421
422     splot = sageplotparser
423     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.

```
424     env_names = oneOf('sageblock sageverbatim sagesilent')
425     senv = r'\begin{' + env_names('env') + '}' + SkipTo(
426            r'\end{' + matchPreviousExpr(env_names) + '}')('code')
427     senv.leaveWhitespace()
428     senv.setParseAction(self.envout)
429
430     spause = sagetexpause
431     spause.setParseAction(self.pause)
432
433     sunpause = sagetexunpause
434     sunpause.setParseAction(self.unpause)
435
436     doit = smacro | splot | senv | spause | sunpause
437
438     str = ''.join(open(fn + '.tex', 'r').readlines())
439     self.result = ''
440
441     doit.transformString(str)
442
443   def macroout(self, s, l, t):
444     self.result += '# \\sage{} from line %s\n' % lineno(l, s)
445     self.result += t.code[1:-1] + '\n\n'
446
447   def plotout(self, s, l, t):
448     self.result += '# \\sageplot{} from line %s:\n' % lineno(l, s)
449     if t.format is not '':
450       self.result += '# format: %s' % t.format[0][1:-1] + '\n'
451     self.result += t.code[1:-1] + '\n\n'
452
453   def envout(self, s, l, t):
454     self.result += '# %s environment from line %s:' % (t.env,
455       lineno(l, s))
456     self.result += t.code[0] + '\n'
457
458   def pause(self, s, l, t):
459     self.result += ('# SageTeX (probably) paused on input line %s.\n\n' %
460                     (lineno(l, s)))
461
462   def unpause(self, s, l, t):
463     self.result += ('# SageTeX (probably) unpaused on input line %s.\n\n' %
464                     (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 pream-

ble, 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.

```
548 def parsedotsage(fn):
549     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.

```
550         inline = re.compile(r" _st_.inline\((?P<num>\d+), (?P<code>.*)\)")
551         plot = re.compile(r" _st_.plot\((?P<num>\d+), (?P<code>.*)\)")
552         goboom = re.compile(r" _st_.goboom\((?P<num>\d+)\)")
553         pausemsg = re.compile(r"print.'(?P<msg>SageTeX (un)?paused.*)'")
554         blockbegin = re.compile(r"_st_.blockbegin\(\)")
555         ignore = re.compile(r"(try:)|(except):")
556         in_comment = False
557         in_block = False
558         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 26) 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.

```
559         for line in f.readlines():
560             if line.startswith('"""'):
561                 in_comment = not in_comment
562             elif not in_comment:
563                 m = pausemsg.match(line)
564                 if m:
565                     cmds.append({'type': 'pause',
566                                  'msg': m.group('msg')})
567                 m = inline.match(line)
568                 if m:
569                     cmds.append({'type': 'inline',
570                                  'num': m.group('num'),
571                                  'code': m.group('code')})
572                 m = plot.match(line)
573                 if m:
574                     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
```

```
608     def __st_plot__(counter, _p_, format='notprovided', **kwargs):
609         if format == 'notprovided':
610             formats = ['eps', 'pdf']
611         else:
612             formats = [format]
613         for fmt in formats:
614             plotfilename = 'plot-%s.%s' % (counter, fmt)
615             _p_.save(filename=plotfilename, **kwargs)""")
616 
617     def _encode(self, d):
618         return 'session={0}&'.format(self._session) + urllib.urlencode(d)
619 
620     def _get_url(self, action, u):
621         with closing(urllib.urlopen(self._srv + '/simple/' + action +
622                                     '?' + u)) as h:
623             data = self._response.match(h.read())
624             result = json.loads(data.group('header'))
625             result['output'] = data.group('output').rstrip()
626         return result
627 
628     def _get_file(self, fn, cell, ofn=None):
629         with closing(urllib.urlopen(self._srv + '/simple/' + 'file' + '?' +
630                      self._encode({'cell': cell, 'file': fn}))) as h:
631             myfn = ofn if ofn else fn
632             data = h.read()
633             if not self._404.search(data):
634                 with open(myfn, 'w') as f:
635                     f.write(data)
636             else:
637                 print('Remote server reported {0} could not be found:'.format(
638                       fn))
639                 print(data)
```

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

```
640     def _do_cell(self, code):
641         realcode = self._codewrap.format(code)
642         result = self._get_url('compute', self._encode({'code': realcode}))
643         if result['status'] == 'computing':
644             cell = result['cell_id']
645             while result['status'] == 'computing':
646                 sys.stdout.write('working...')
647                 sys.stdout.flush()
648                 time.sleep(10)
649                 result = self._get_url('status', self._encode({'cell': cell}))
650         if debug:
651             print('cell: <<<', realcode, '>>>', 'result: <<<',
652                   result['output'], '>>>', sep='\n')
653         return result
654 
655     def do_inline(self, code):
```

```
656         return self._do_cell(' print(latex({0}))'.format(code))
657
658     def do_block(self, code):
659         result = self._do_cell(code)
660         for fn in result['files']:
661             self._get_file(fn, result['cell_id'])
662         return result
663
664     def do_plot(self, num, code, plotdir):
665         result = self._do_cell(' __st_plot__({0}, {1})'.format(num, code))
666         for fn in result['files']:
667             self._get_file(fn, result['cell_id'], os.path.join(plotdir, fn))
668         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.

```
669     def close(self):
670         sys.stdout.write('Logging out of {0}...'.format(server))
671         sys.stdout.flush()
672         self._get_url('logout', self._encode({}))
673         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.

```
674 def do_plot_setup(plotdir):
675     printc('initializing plots directory...')
676     if os.path.isdir(plotdir):
677         shutil.rmtree(plotdir)
678     os.mkdir(plotdir)
679     return True
680
681 did_plot_setup = False
682 plotdir = 'sage-plots-for-' + jobname + '.tex'
683
684 def labelline(n, s):
685     return r'\newlabel{@sageinline' + str(n) + '}{{' + s  + '}{}{}{}{}}\n'
686
687 def printc(s):
688     print(s, end='')
689     sys.stdout.flush()
690
691 error = re.compile("(^" + traceback_str + ")|(^Syntax Error:)", re.MULTILINE)
692
693 def check_for_error(string, line):
694     if error.search(string):
695         print("""
696 **** 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(line)
```

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. Almost all 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.

10 Copying and licenses

If you are unnaturally curious about the current state of the SageTEX package, you can visit `http://www.bitbucket.org/ddrake/sagetex/`. There is a Mercurial repository and other stuff there.

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

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

The *documentation* of the SageTEX package is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-sa/3.0/` or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Change History

Index

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