The sagetex package[*]

Dan Drake (ddrake@member.ams.org) and others

March 12, 2008

1 Introduction

Why should the Haskell folks have all the fun?

Literate Haskell is a popular way to mix Haskell source code and LaTeX documents. (Well, 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.

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 $26^3*10^3 = \sage{26^3*10^3}$
license plates.
```

and it will produce

> There are 26 choices for each letter, and 10 choices for each digit, for a total of 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 file, and doing the \includegraphics business with the correct filename yourself. If you write this:

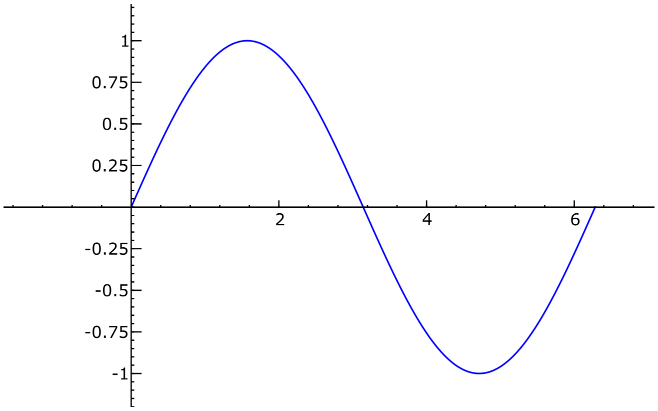
[*] This document corresponds to sagetex v1.4, dated 2008/03/12.

```
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

The simplest way to "install" sagetex is to copy the files `sagetex.sty` and `sagetex.py` into the same directory as your document. This will always work, as LaTeX and Python search the current directory for files. It is also convenient for zipping up a directory to send to a colleague who is not yet enlightened enough to be using sagetex.

Rather than make lots of copies of those files, you can keep them in one place and update the TEXINPUTS and PYTHONPATH environment variables appropriately.

Perhaps the best solution is to put the files into a directory searched by TeX and friends, and then edit the `sagetex.sty` file so that the `.sage` files we generate update Python's path appropriately—look for "Python path" in `sagetex.sty`. This is suitable for a system-wide installation, or if you are the kind of person who keeps a `texmf` tree in your home directory.

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. 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 which, 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.

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]

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

3.1 Inline Sage

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

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

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

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

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

in Sage.

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

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

Here, I'll do just that right now: the prime factorization of the current page plus 12 is 2^4.

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 subsection 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

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.

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

will run the following command in Sage:

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

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

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

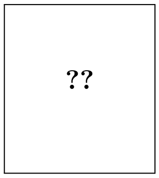
You can specify a file format if you like. This must be the *second* optional argument, so you must use empty brackets if you're not passing anything to `\includegraphics`:

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

The filename is automatically generated, and unless you specify a format, both EPS and PDF files will be generated. This allows you to freely switch between using, say, a DVI viewer (many of which have support for automatic reloading, source specials and make the writing process easier) and creating PDFs for posting on the web or emailing to colleagues.

If you ask for, say, a PNG file, keep in mind that ordinary `latex` and DVI files have no support for DVI files; sagetex detects this and will warn you that it cannot find a suitable file if using `latex`. If you use `pdflatex`, there will be no problems because PDF files can include PNG graphics.

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



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

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

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

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.[2] 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. (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 one option: `imagemagick`. 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.

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

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 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 = sin(x) - 1
   g = log(x)
   h = 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.

4 Other notes

Here are some other notes on using sagetex.

Using Beamer The BEAMER package does not play nicely with verbatim-like environments. 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.

Thanks to Franco Saliola for reporting this.

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!" (page 7) 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('plot = commands to make your plot')
mathematica('Export["%s/graphicsfile.eps", plot]' % 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.

5 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.

5.1 The style file

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

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 and the default indent.

```
\newcounter{ST@inline}
\newcounter{ST@plot}
\setcounter{ST@inline}{0}
\setcounter{ST@plot}{0}
\newlength{\sagetexindent}
\setlength{\sagetexindent}{5ex}
```

`\ST@epsim` By default, we don't use ImageMagick to create EPS files when a non-default format is specified.

```
\newcommand{\ST@epsim}{False}
```

The expansion of that macro gets put into a Python function call, so it works to have it be one of the strings "`True`" or "`False`".

Declare the `imagemagick` option and process it:

```
\DeclareOption{imagemagick}{\renewcommand{\ST@epsim}{True}}
\ProcessOptions\relax
```

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

It's time to deal with files. Open the `.sage` file:

```
\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. If you know what directory `sagetex.py` will be kept in, delete the `\iffalse` and `\fi` lines in the generated style file (*don't* do it in the `.dtx` file) and change the directory appropriately. This is useful if you have a `texmf` tree in your home directory or are installing `sagetex` system-wide; then you don't need to copy `sagetex.py` into the same directory as your document.

```
\newcommand{\ST@wsf}[1]{\immediate\write\ST@sf{#1}}
\iffalse
%% To get .sage files to automatically change the Python path to find
%% sagetex.py, delete the \iffalse and \fi lines surrounding this and
%% change the directory below to where sagetex.py can be found.
\ST@wsf{import sys}
\ST@wsf{sys.path.insert(0, 'directory with sagetex.py')}
\fi
\ST@wsf{import sagetex}
\ST@wsf{sagetex.openout('\jobname')}
```

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.

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

Now let's define the cool stuff.

`\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.)

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

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.

```
\begin{NoHyper}\ref{@sagelabel\theST@inline}\end{NoHyper}%
```

Now check to see if the label has already been defined. (The internal implementation of labels in LaTeX involves defining a function "`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. Finally, step the counter.

```
\@ifundefined{r@@sagelabel\theST@inline}{\gdef\ST@rerun{x}}{}%
\stepcounter{ST@inline}}
```

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.

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

\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
```

\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}
```

\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]}%]
}
```

That percent sign followed by a square bracket seems necessary; I have no idea why.

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.

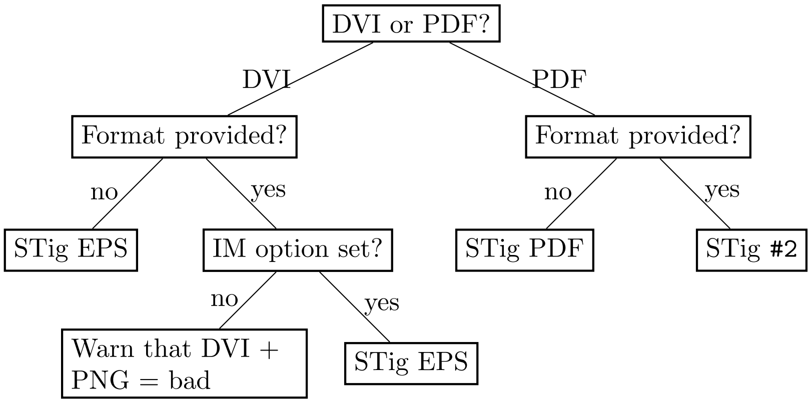


Figure 1: The logic tree that `\sageplot` uses to decide whether to run `\includegraphics` or to yell at the user. "Format" is the `#2` argument to `\sageplot`, "STig ext" means a call to `\ST@inclgrfx` with "ext" as the second argument, and "IM" is Imagemagick.

`\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. Note that the `\write` gobbles up line endings, so the `sageplot` bits below get written to the `.sage` file as one line.

```
\def\ST@sageplot[#1][#2]#3{%
\ST@wsf{try:}%
\ST@wsf{ sagetex.initplot('\jobname')}%
\ST@wsf{ sagetex.plot(\theST@plot, #3, format='#2', epsmagick=\ST@epsim)}%
\ST@wsf{except:}%
\ST@wsf{ sagetex.goboom(\the\inputlineno)}%
```

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, 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}}%
```

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 so, try to include an EPS file anyway.

```
    {\ifthenelse{\equal{\ST@epsim}{True}}
      {\ST@inclgrfx{#1}{eps}}%
```

If we're not using the imagemagick option, we're going to issue some sort of warning, depending on whether the file exists yet or not.

```
      {\IfFileExists{\ST@plotdir/plot-\theST@plot.#2}%
        {\framebox[2cm]{\rule[-1cm]{0cm}{2cm}\textbf{??}}%
         \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}}%
        {\framebox[2cm]{\rule[-1cm]{0cm}{2cm}\textbf{??}}%
         \PackageWarning{sagetex}{Graphics file
         \ST@plotdir/plot-\theST@plot.#2\space on page \thepage\space
         does not exist}%
         \gdef\ST@rerun{x}}}}%
\fi
```

Finally, step the counter and we're done.

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

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

```
\newcommand{\ST@inclgrfx}[2]{%
  \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.

```
    {\framebox[2cm]{\rule[-1cm]{0cm}{2cm}\textbf{??}}%
     \PackageWarning{sagetex}{Graphics file
     \ST@plotdir/plot-\theST@plot.#2\space on page \thepage\space does not
     exist}%
     \gdef\ST@rerun{x}}}
```

Figure 2 makes this a bit clearer.

\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.

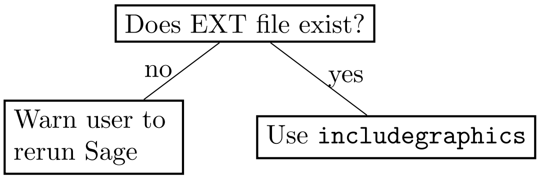


Figure 2: The logic used by the `\ST@inclgrfx` command.

The last bit is some magic from the `verbatim` package manual that makes LaTeX respect line breaks.

```
\newcommand{\ST@beginsfbl}{%
  \@bsphack%
  \ST@wsf{sagetex.blockbegin()}%
  \ST@wsf{try:}%
  \let\do\@makeother\dospecials\catcode`\^^M\active}
```

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

```
\newcommand{\ST@endsfbl}{%
\ST@wsf{except:}%
\ST@wsf{ sagetex.goboom(\the\inputlineno)}%
\ST@wsf{sagetex.blockend()}}
```

Now let's define the "verbatim-like" environments. There are four possibilities, corresponding to 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.

Now we deal with some end-of-file 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.

```
\AtEndDocument{\ST@wsf{sagetex.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}%
\PackageWarningNoLine{sagetex}{Run Sage on \jobname.sage, and then run
LaTeX on \jobname.tex again}}}
```

5.2 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 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 used by Sage scripts generated from a LaTeX document
using the sagetex package. Keep it somewhere where Sage and Python can
find it and it will automatically be imported.""")
  sys.exit()
```

We start with some imports and definitions of our global variables. This is a relatively specialized use of Sage, so using global variables isn't a bad idea. Plus I think when we import this module, they will all stay inside the `sagetex` namespace anyway.

```
from sage.misc.latex import latex
import os
import os.path
import hashlib
import traceback
import subprocess
import shutil
initplot_done = False
dirname       = None
filename      = ""
```

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

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

`openout` This function opens a `.sout.tmp` file and writes 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.

```
def openout(f):
  global filename
  filename = f
  global _file_
  _file_ = open(f + ’.sout.tmp’, ’w’)
  s = ’% This file was *autogenerated* from the file ’ + \
        os.path.splitext(filename)[0] + ’.sage.\n’
  _file_.write(s)
  progress(’Processing Sage code for %s.tex...’ % filename)
```

initplot

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

```
def initplot(f):
  global initplot_done
  if not initplot_done:
    progress('Initializing plots directory')
    global dirname
```

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, I think we could find out the correct extension, but it would involve a lot of irritating mucking around.

```
    dirname = 'sage-plots-for-' + f + '.tex'
    if os.path.isdir(dirname):
      shutil.rmtree(dirname)
    os.mkdir(dirname)
    initplot_done = True
```

inline

This function works with \sage from the style file 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, 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:

```
def inline(counter, s):
  progress('Inline formula %s' % counter)
  _file_.write('\\newlabel{@sagelabel' + str(counter) + '}{{' + \
               latex(s) + '}{}{}{}{}}\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.

```
def blockbegin():
  progress('Code block begin...', False)
def blockend():
  progress('end')
```

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

```
def plot(counter, p, format=’notprovided’, epsmagick=False, **kwargs):
  global dirname
  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.

```
  if format == ’notprovided’:
    formats = [’eps’, ’pdf’]
  else:
    formats = [format]
  for fmt in formats:
    plotfilename = os.path.join(dirname, ’plot-%s.%s’ % (counter, fmt))
    #print(’  plotting %s with args %s’ % (plotfilename, kwargs))
    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.

```
    if format != ’notprovided’ and epsmagick is True:
      print(’Calling Imagemagick to convert plot-%s.%s to EPS’ % \
        (counter, format))
      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.

```
def toeps(counter, ext):
  global dirname
  subprocess.check_call([’convert’,\
    ’%s/plot-%s.%s’ % (dirname, counter, ext), \
    ’%s/plot-%s.eps’ % (dirname, 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 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.

```
def goboom(line):
  global filename
  print('\n**** Error in Sage code on line %s of %s.tex! Traceback\
 follows.' % (line, filename))
  traceback.print_exc()
  print('\n**** Running Sage on %s.sage failed! Fix %s.tex and try\
 again.' % (filename, filename))
  os.remove(filename + '.sout.tmp')
  sys.exit(1)
```

`endofdoc` When we’re done processing, we have a couple little cleanup tasks. We want to put the MD5 sm 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 tell 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, say, a line break 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 calls to Sage have 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`, so we will strip those lines out. Basically we are doing

```
grep -v '^ sagetex.goboom' filename.sage | md5sum
```

(In fact, what we do below produces exactly the same sum.)

```
def endofdoc():
  global filename
  sagef = open(filename + '.sage', 'r')
  m = hashlib.md5()
  for line in sagef:
    if line[0:15] != ' sagetex.goboom':
      m.update(line)
  s = '%' + m.hexdigest() + '% md5sum of .sage file (minus "goboom" \
lines) that produced this\n'
  _file_.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.)

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

```
_file_.close()
os.rename(filename + '.sout.tmp', filename + '.sout')
progress('Sage processing complete. Run LaTeX on %s.tex again.' %\
        filename)
```

6 Credits and acknowledgements

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 almost zero original code there), made significant changes to the Python module, put both files into Docstrip format, and wrote all the documentation.

Many thanks to Jason Grout for his numerous comments, suggestions, and feedback.

7 Copying and licenses

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

v1.0

General: Initial version 1

v1.1

General: Wrapped user-provided Sage code in try/except clauses; plotting now has optional format argument. 1

v1.2

General: Imagemagick option; better documentation 1

v1.3

`\sageplot`: Iron out warnings, cool Ti*k*Z flowchart 11

v1.3.1

General: Internal variables renamed; fixed typos 1

v1.4

General: MD5 fix, percent sign macro, CTAN upload 1

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.