The sympytex package[*]

Tim Molteno (tim@physics.otago.ac.nz) and others

May 20, 2014

1 Introduction

The sympytex package allows you to embed the symbolic python package Sympy (see http://www.sympy.org) and LaTeX.

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 $26^3*10^3 = \sympy{(26**3 * 10**3)}$
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. Sympy 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 sympytex, you can concentrate on the mathematical structure ("I need the product of 26^3 and 10^3") and let Sympy deal with the base-10 presentation of the number.

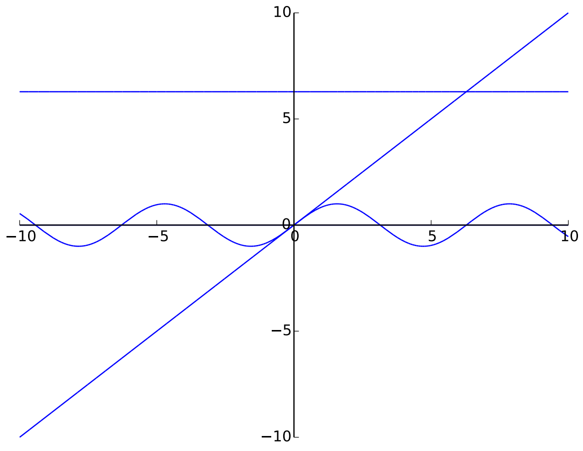
A less trivial, and perhaps more useful example is plotting. You can include a plot of the sine curve without manually producing a plot, saving an EPS or PDF file, and doing the `\includegraphics` business with the correct filename yourself. If you write this:

```
Here is a lovely graph of the sine curve:
\sympyplot{plot(sin(x), x, 0, 2*pi, show=False)}
```

in your LaTeX file, it produces

Here is a lovely graph of the sine curve:

[*]This document corresponds to sympytex v0.3, dated 2014/05/16.



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 sympytex takes care of the gritty details of producing the file and sourcing it into your document.

But `\sympyplot` isn't magic I just tried to convince you that sympytex 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 sympytex does is make it easy to to create graphics; it doesn't magically make your graphics good, appropriate, or useful.

2 Installation

The simplest way to "install" sympytex is to copy the files `sympytex.sty` and `sympytex.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 sympytex.

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 `sympytex.sty` file so that the `.sympy` files we generate update Python's path appropriately—look for "Python path" in `sympytex.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 `sympytex` works. Naturally the very first step is to put `\usepackage{sympytex}` 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 `.sympy` file is written. This is a python source file that uses the `sympytex.py` Python module from this package and when execute the python code in 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 Sympy's computation.

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

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

Now that you know that, let's describe what macros `sympytex` 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]

3.1 Inline Sympy

`\sympy` `\sympy{`⟨*Sympy code*⟩`}`

takes whatever Sympy code you give it, runs Sympy's `latex` function on it, and puts the result into your document.

For example, if you do `\sympy{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 Sympy.

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

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

```
The prime factorization of the current page number
times $2^{17} + 1$  is
$\sympy{factorint(\thepage*2**17+1)}$.
```

Here, I'll do just that right now: the prime factorization of the current page number times $2^{17}+1$ is $\left\{3 : 1, \quad 174763 : 1\right\}$.

The `\sympy` 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 `\sympy` (or `\sympyplot`), 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 `.sympy` file and Sympy 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 `.sympy` file.

3.2 Graphics and plotting

`\sympyplot` $\boxed{\texttt{\textbackslash sympyplot[}\langle \textit{ltx opts}\rangle\texttt{][}\langle \textit{fmt}\rangle\texttt{]\{}\langle \textit{graphics obj}\rangle,\ \langle \textit{keyword args}\rangle\texttt{\}}}$ plots the given Sympy 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 Sympy graphics object. The options are described in Table 1.

Option	Description
⟨*ltx options*⟩	Any text here is passed directly into the optional arguments (between the square brackets) of an `\includegraphics` command. If not specified, "`width=.75\textwidth`" will be used.
⟨*fmt*⟩	You can optionally specify a file extension here; Sympy will then try to save the graphics object to a file with extension *fmt*. If not specified, sympytex will save to EPS and PDF files.
⟨*graphics obj*⟩	A Sympy 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 `\sympyplot` command.

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

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

will run the following command in Sympy:

```
sympy: 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}
```

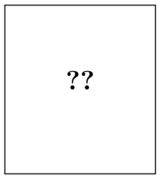
You can specify a file format if you like. This must be the *second* optional argument, so you must use empty brackets if you're not passing anything to `\includegraphics`:

```
\sympyplot[][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; sympytex 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 sympytex 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 `sympy-plots-for-filename.tex`. You can safely delete that directory anytime; if sympytex can't find the files, it will warn you to run Sympy to regenerate them.

WARNING! When you run Sympy on your `.sympy` file, all files in the `sympy-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.

3.3 Verbatim-like environments

The sympytex package provides several environments for typesetting and executing Sympy code.

sympyblock

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

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

and then anytime later write in your source file

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

and the `\sympy` call will get correctly replaced by $\log(x)\cos(x) + (\sin(x) - 1)/x$. You can use any Sympy or Python commands inside a `sympyblock`; all the commands get sent directly to Sympy.

sympysilent

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

sympyverbatim

This environment is the opposite of the one above: whatever you type will be typeset, but not written into the `.sympy` 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 Sympy code... but the `verbatim` package, which is always loaded when using sympytex, provides such an environment: `comment`. Another way to do this is to put stuff between `\iffalse` and `\fi`.

\sympytexindent

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

4 Other notes

Here are some other notes on using sympytex.

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{sympyblock}
```

```
# sympy stuff
# more stuff \end{sympyblock}
\end{frame}
```

For some reason, BEAMER inserts an extra line break at the end of the environment; if you put the `\end{sympyblock}` on the same line as the last line of your code, it works properly.

Thanks to Franco Saliola for reporting this.

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 `sympyblock` and friends are stol—um, adapted from the `verbatim` package manual. So grab the `verbatim` package.

```
\RequirePackage{verbatim}
```

Unsurprisingly, the `\sympyplot` 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 `\sympyplot` 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{\sympytexindent}
\setlength{\sympytexindent}{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 .sympy file:

```
\newwrite\ST@sf
\immediate\openout\ST@sf=\jobname.sympy
```

\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 .sympy file. If you know what directory sympytex.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 sympytex system-wide; then you don't need to copy sympytex.py into the same directory as your document.

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

\sympy This macro combines \ref, \label, and Sympy all at once. First, we use Sympy to get a LaTeX representation of whatever you give this function. The Sympy 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 .sympy 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{\sympy}[1]{%
\ST@wsf{try:}%
\ST@wsf{ sympytex.inline(\theST@inline, #1)}%
\ST@wsf{except:}%
\ST@wsf{ sympytex.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{@sympylabel\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 Sympy on the `.sympy` file at the end of the run. Finally, step the counter.

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

`\sympyplain` This macro combines `\ref`, `\label`, and Sympy all at once. First, we use Sympy to get a plain representation of whatever you give this function. The Sympy 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 `.sympy` 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{\sympyplain}[1]{%
\ST@wsf{try:}%
\ST@wsf{ sympytex.inlineplain(\theST@inline, #1)}%
\ST@wsf{except:}%
\ST@wsf{ sympytex.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{@sympylabel\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 Sympy on the `.sympy` file at the end of the run. Finally, step the counter.

```
\@ifundefined{r@@sympylabel\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}{sympy-plots-for-\jobname.tex}
```

`\sympyplot` This function is similar to `\sympy`. 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 sympyplot 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{\sympyplot}[1][width=.75\textwidth]{%
  \@ifnextchar[{\ST@sympyplot[#1]}{\ST@sympyplot[#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 sympytex 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@sympyplot` Let's see the real code here. We write a couple lines to the `.sympy` 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 `sympyplot` bits below get written to the `.sympy` file as one line.

```
\def\ST@sympyplot[#1][#2]#3{%
\ST@wsf{try:}%
\ST@wsf{ sympytex.initplot('\jobname')}%
```

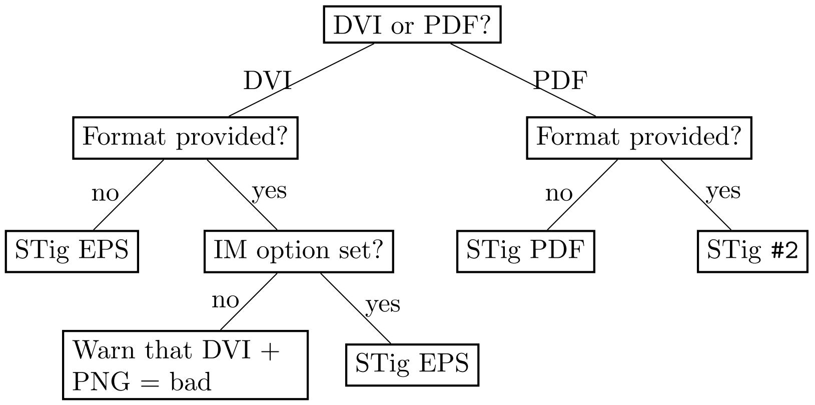


Figure 1: The logic tree that `\sympyplot` uses to decide whether to run `\includegraphics` or to yell at the user. "Format" is the `#2` argument to `\sympyplot`, "STig ext" means a call to `\ST@inclgrfx` with "ext" as the second argument, and "IM" is Imagemagick.

```
\ST@wsf{ sympytex.plot(\theST@plot, #3, format='#2', epsmagick=\ST@epsim)}%
\ST@wsf{except:}%
\ST@wsf{ sympytex.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{#2}{eps}}
     {\ST@inclgrfx{#1}{eps}}%
```

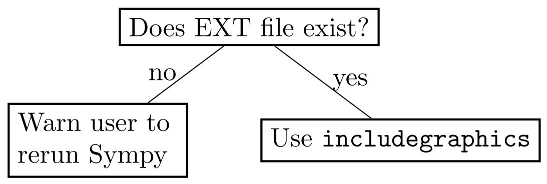


Figure 2: The logic used by the `\ST@inclgrfx` command.

```
     {\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{sympytex}{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{sympytex}{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 Sympy again.

```
    {\framebox[2cm]{\rule[-1cm]{0cm}{2cm}\textbf{??}}%
     \PackageWarning{sympytex}{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 `.sympy` file block", an internal-use abbreviation that sets things up when we start writing a chunk of Sympy code to the `.sympy` file. It begins with some TEX magic that fixes spacing, then puts the start of a try/except block in the .sympy file—this not only allows the user to indent code without Sympy/Python complaining about indentation, but lets us tell the user where things went wrong. The last bit is some magic from the verbatim package manual that makes LATEX respect line breaks.

```
\newcommand{\ST@beginsfbl}{%
  \@bsphack%
  \ST@wsf{sympytex.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{ sympytex.goboom(\the\inputlineno)}%
\ST@wsf{sympytex.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 .sympy file or not.

sympyblock This environment does both: it typesets your code and puts it into the .sympy file for execution by Sympy.

```
\newenvironment{sympyblock}{\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 .sympy file.

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

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

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

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

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

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

```
\newenvironment{sympysilent}{\ST@beginsfbl%
\def\verbatim@processline{\ST@wsf{ \the\verbatim@line}}%
\verbatim@start}%
{\ST@endsfbl\@esphack}
```

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

```
\newenvironment{sympyverbatim}{%
\def\verbatim@processline{\hspace{\sympytexindent}\the\verbatim@line\par}%
\verbatim}%
{\endverbatim}
```

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

Now we deal with some end-of-file cleanup.

We tell the Sympy 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{sympytex.endofdoc()}%
\@ifundefined{ST@rerun}{}%
```

Otherwise, we issue a warning to tell the user to run Sympy on the `.sympy` 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 Sympy 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 *Sympy*. 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 Sympy if it's necessary.

```
{\PackageWarningNoLine{sympytex}{There were undefined Sympy formulas
and/or plots}%
\PackageWarningNoLine{sympytex}{Run python on \jobname.sympy, and then run
LaTeX on \jobname.tex again}}}
```

5.2 The Python module

The style file writes things to the `.sympy` file and reads them from the `.sout` file. The Python module provides functions that help produce the `.sout` file from the `.sympy` 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 `sympytex.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 Sympy modules; that way, this error message gets printed whether you run the script with Sympy or with Python.

```
import sys
```

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

This file will be used by Sympy scripts generated from a LaTeX document
using the sympytex package. Keep it somewhere where Sympy 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 Sympy, so using global variables isn't a bad idea. Plus I think when we import this module, they will all stay inside the `sympytex` namespace anyway.

```
import sympy
from sympy.plotting.plot import plot, Plot
import os
import os.path
import hashlib
import traceback
import subprocess
import shutil
initplot_done = False
dirname       = None
filename      = ""
```

`ttexprint` This function gets around the insertion of begin/end math symbols that sympy puts into its latex output

```
from string import strip
def ttexprint(exp):
  return strip(sympy.latex(exp, mode='inline'),'$')
```

`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 sympy files; we are automatically generating a file with sympy, 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] + '.sympy.\n'
  _file_.write(s)
  progress('Processing Sympy 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 = 'sympy-plots-for-' + f + '.tex'
    if os.path.isdir(dirname):
      shutil.rmtree(dirname)
    os.mkdir(dirname)
    initplot_done = True
```

inline This function works with `\sympy` from the style file to put Sympy 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 Sympy on `s`.

We print out the line number so if something goes wrong, the user can more easily track down the offending `\sympy` 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{@sympylabel' + str(counter) + '}{{' + \
               ttexprint(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.

inlineplain This function works with `\sympy` from the style file to put Sympy output into your LaTeX file. This does not format the output!

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

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

```
def inlineplain(counter, s):
  progress('Inline Plain formula %s' % counter)
  _file_.write('\\newlabel{@sympylabel' + str(counter) + '}{{' + \
               str(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 `\sympyplot` 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 Sympy plotting functions) has nice support for keyword arguments. The `#3` argument to `\sympyplot` 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))
    if (isinstance(p, Plot)):
      p.save(plotfilename)
    else:
      p.savefig(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 `sympytex` 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 `.sympy` file, should result in a reasonable error message if something strange happens.

goboom When a chunk of Sympy 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 `.sympy` 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 Sympy code on line %s of %s.tex! Traceback\
 follows.' % (line, filename))
  traceback.print_exc()
  print('\n**** Running Sympy on %s.sympy 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 `.sympy` 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 Sympy to update the `.sout` file. But there is a problem: we write line numbers to the `.sympy` 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 Sympy even though none of the actual calls to Sympy have changed.

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

The answer is to only find the MD5 sum of *part* of the `.sympy` 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 '^ sympytex.goboom' filename.sympy | md5sum
```

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

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

Now, we do issue warnings to run Sympy on the `.sympy` file and an external program might look for those to detect the need to rerun Sympy, but those warnings do not quite capture all situations. (If you've already produced the `.sout` file and change a `\sympy` 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('Sympy processing complete. Run LaTeX on %s.tex again.' %\
           filename)
```

6 Credits and acknowledgements

This is competely based around the excellent sagetex package for embedding sage into LaTeX. All credit to Dan Drake and others.

7 Copying and licenses

Change History

v1.0

General: Initial version 1

v1.3

\sympyplot: Iron out warnings, cool Ti*k*Z flowchart 10

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.