A hybrid LaTeX collection.

Leo Brewin

School of Mathematics, Monash University

Leo.Brewin@monash.edu

August 2019

Version 0.2

https://github.com/leo-brewin/hybrid-latex

Abstract

This collection of LaTeX packages and Python scripts provides simple tools to allow active code, including Python, Mathematica, Maple, Matlab and Cadabra, to be embedded in a LaTeX document. It does this by defining a number of simple macros and Python scripts that build a bridge between the output generated by the active code and the LaTeX code. Each language is defined independently of the other languages. The scripts and macros for one language are almost identical to those for the other languages in the collection differing only in the names of the macros and some minor language specific tweaks. This makes it a simple matter to extend the collection to other languages.

Contents

1 Introduction

Programs like Mathematica, Maple, Python and so on do a wonderful job of symbolic, numeric and graphical computations. On the other hand, LaTeX produces exceptional typeset documents but is less adept at symbolic and other forms of computation. It is a case of horses for courses. A common approach is to use one or more codes (in Python, Maple etc.) to do the computations and a separate LaTeX code to collect and document the results. It seems reasonable then to attempt to consolidate these separate codes into a single hybrid LaTeX document. That is the main purpose of this collection – to provide tools that support active code within a LaTeX document.

This collection currently supports Mathematica, Maple, Matlab, Python and Cadabra. The majority of the remainder of this document will deal specifically with Python as the guest language within the LaTeX source. The changes required to use the other guest languages are minor and are spelt out in detail in a later section.

2 The pyLaTeX package

There are other packages that do the same job, such as `PythonTeX` and `SageTeX`. In each case the journey from source to pdf is similar; the Python code is mined from the LaTeX source, passed over to Python and the output feed back into the LaTeX source. The packages differ in both the tools used to mine the Python code and the manner in which the output is made visible to the LaTeX source.

In both `PythonTeX` and `SageTeX`, the Python code is mined using a first pass of LaTeX whereas `pyLaTeX` uses a Python script to do the same job. There are two main reasons for adopting Python as the code miner. First, it is much easier to extend `pyLaTeX` to incorporate other foreign languages (such as Mathematica, Matlab, Cadabra etc.). Doing so in `PythonTeX` and `SageTeX` requires careful tweaking of the underlying LaTeX class files – such adventures requires a solid understanding of the LaTeX programming language and is not for the faint of heart. The second reason is that LaTeX is not the ideal tool to mine Python (or any other code) from a LaTeX document. There are much better text processing tools (such as Python) that can do the same job while being much easier (for the developer) to read, debug and extend.

There are also some important differences in the way the end user accesses the Python output for use in the LaTeX source. In `PythonTeX` and `SageTeX` the Python output is saved to a collection of files (roughly one file per Python code block) while `pyLaTeX` creates a single file. In all cases the contents of these files are made available to the LaTeX source by way of special LaTeX macros. The `PythonTeX` and `SageTeX` output files are tightly coupled to the original LaTeX source. This means that their contents can not be easily shared with other LaTeX sources (without manual intervention). In contrast, the `pyLaTeX` output is a single file that contains a collection of LaTeX macros, one for each piece of (tagged) Python output, that can be read by any other LaTeX source. Thus a user could split a larger LaTeX source into a collection of smaller sources (e.g., chapters of a book) with the Python output included as needed. An example of this sharing of Python output can be found in the examples directory. The file `summary.tex` does no symbolic calculations of its own but instead pulls in selected parts of the Python output from other examples in that directory.

Another important difference between `pyLaTeX` and `PythonTeX` and `SageTeX` is that `pyLaTeX` uses tags to nominate which pieces of the Python output should be made available to the LaTeX source. Both `PythonTeX` and `SageTeX` allow all Python output to accessed in the LaTeX source.

The problem with that method is that it is not possible to record the history of a single Python expression. For example, if the Python code computes the expression `foo` five times then only the fifth instance will be seen by LaTeX.

3 Requirements

3.1 Latex

The `pyLaTeX` package draws on a number of latex packages all of which should be available in any recent LaTeX distribution. The examples described in this document were run using the `TeXLive-2018` distribution for macOS.

The `pyLaTeX` package provides both a normal style file, `pylatex.sty` and a class file `pylatex.cls`. The preamble to the style file contains the following declarations.

```
\usepackage{comment}
\usepackage{listings}
\usepackage{keyval}
\usepackage{etoolbox}
\usepackage{xcolor}
\usepackage{pymacros}
```

The `pymacros` package is a key part of the `pyLaTeX` collection as it defines the core macros for `pyLaTeX`.

The class file first includes the style file then executes the following statements

```
\usepackage[papersize={297mm,210mm},
            hmargin=2cm,tmargin=1.0cm,bmargin=1.5cm]{geometry}
\usepackage{amsmath}
\usepackage{amssymb}
\usepackage{hyperref}
\usepackage{breqn}
```

The extra packages included in the class file are not essential but have proven useful in many cases and thus are included as a convenient default. Note that the options to the geometry package sets the output to A4 landscape format. This can be changed, either by editing the class file or by including a line like the following somewhere in the preamble of the users document.

```
\geometry{margin=2.0cm,paperheight=25cm}
```

3.2 Python

There are no particular requirements on the Python environment in order to use the `pyLaTeX` package though the examples listed below do require `sympy` version 1.1.1 or later. The simple measure is that if your embedded Python code can be run in your normal Python environment then that same code should work equally well using `pyLaTeX`. All of the examples described in this document were run under macOS using the default Apple Python environment (Python 2.7) as well as under the Anaconda 5.2 (Python3.6) environment.

4 Setup

There is no setup tool as you only have a few files to install. The `Python` directory contains a number of subdirectories each of which contains files that should be copied as follows.

`shell/*`	Copy all files to anywhere that is visible on your path.
`python/*`	Copy all files to anywhere that is visible on your path.
`latex/*`	Copy all files to anywhere that is visible to LaTeX.

For example, the shell and Python scripts could be copied to `$HOME/local/sh/` while the LaTeX files could be copied to `$HOME/tex/inputs/`. Provided those directories are on the appropriate paths (i.e., PATH and TEXINPUTS), that should be all that needs doing.

Note that each of the Python scripts uses the following shebang (i.e., the first line in the file)

```
#!/usr/local/bin/python3
```

You may need to change this to match the path to your version of `python3`.

5 Compiling

To compile the file `foo.tex` use

```
pylatex.sh -i foo
```

The `pylatex.sh` script will call Pyhton and LaTeX as required to produce the pdf output. Along the way a number of intermediate files will be created (most of which will be deleted by default). Using the above command will produce not only the pdf but also `foo.pytex`. This file contains a list of macros, one for each `pyLaTeX` tag in `foo.tex`, that are executed when the `pyLaTeX` macros such as `\py{bah}` are called (in this case `bah` is a tag associated with part of the Python code in the `foo.tex` source). This same file can also be included in other LaTeX files using `\input{foo.pytex}`. This makes the Python output from `foo.tex` available to other LaTeX files.

Executing `pylatex.sh` with the `-h` flag will produce the following usage information.

```
usage : pylatex.sh -i file [-P<path to python>]
                           [-I<path to pymacros.sty>] [-s] [-k] [-x] [-W] [-h]
options :  -i file : source file (with or without .tex extension)
           -I file : full path to pymacros.sty file
           -P path : full path to the Python binary
           -s : silent, don't open the pdf file
           -k : keep all temporary files
           -x : don't call latex
           -W : warn if errors found in the output for some tags
           -h : this help message
example : pylatex.sh -i file -P/usr/local/bin/python
```

The intermediate files produced by the `-k` option are (excluding the standard LaTeX files)

`foo.py:`	A copy of all the Python code stripped bare of (most) tags.
`foo_.py:`	Similar to `foo.py` but with additional Python code to catch Python output for the `pyLaTeX` tags.
`foo.pyidx:`	An index of all `pyLaTeX` tags.
`foo.pytex:`	A set of macros, one for each tag, expanding to the Python output.
`foo.pytxt:`	The raw Python output from `foo_.py`.

The `-k` option is useful in cases where there is a bug in the users Python code. This option will force `pylatex.sh` to retain all intermediate files, in particular the mined Python code `foo.py`. This file can then be debugged and in turn the original LaTeX source can be corrected.

The `-I` option can be used in cases where the output from `foo.tex` is intended for use in other LaTeX sources that do not use the `pyLaTeX` style file `pylatex.sty`. This could be useful for sharing results with colleagues who have not installed the `pyLaTeX` package or when submitting a paper to a journal. The nett effect of the `-I` option is to prepend the `foo.pytex` file with the LaTeX definitions for all of the `pyLaTeX` macros. Another approach would be to not use the `-I` but share both the `foo.pytex` and `pymacros.sty` file with colleagues or journals.

The only files that must be kept are `foo.tex` (obviously) and `foo.pytex`. This pair of files alone is sufficient for successful execution of LaTeX. The file `foo.pytex` only needs to be recreated (using `pylatex.sh`) when changes have been made to the Python code in `foo.tex`.

It is important to note that the Python code mined from the LaTeX source will be written to a single file. Thus it is important that all parts of this file interact harmoniously. If there is any possibility that one part of the (combined) code may adversely effect other parts then some action must be taken. One solution is to use language features of Python to avoid the clash, such as placing conflicting code in separate functions. Another solution is to move the conflicting code to separate LaTeX documents, run `pylatex.sh` over each document, then include the results in the original LaTeX source using `\input` on the corresponding `.pytex` files.

5.1 Caution

Be very careful – each of the above named temporary files will overwrite any existing files of the same name. An easy road to disaster lies this way. Suppose you have an existing Python file named `foo.py` and suppose you wish to catch its output for later use in some other LaTeX document. You might start by creating a wrapper file, `foo.tex`, such as the following

```
\documentclass[12pt]{pylatex}
\begin{document}
   \begin{python}
      \Input{foo.py}
   \end{python}
\end{document}
```

The `pylatex.sh` will expand the `\Input{foo.py}` command to include the contents of `foo.py`. No damage done – so far. But now `pylatex.sh` will read the newly merged file and will create its own version of `foo.py` without *any* of the `pyLaTeX` tags. Thus any subsequent processing of `foo.tex` will *not* capture any Python output – because all of the Python tags have been deleted.

5.2 The pypreproc.py and pypostproc.py scripts

The job of the `pypreproc.py` script is to harvest the Python code from the LaTeX source. For a source named `foo.tex` the script could be run using

```
pypreproc.py -i foo
```

This will produce `foo.py, foo_.py` and `foo.pyidx`. The file `foo.py` is an almost[1] exact copy of the active Python code while `foo_.py` is a version of `foo.py` containing extra Python commands to catch the nominated Python output. This file is passed to Python to create the `foo.pytxt` file which in turn is passed to `pypostproc.py` using

```
pypostproc.py -i foo
```

which should create the `foo.pytex` file.

The pre and post processor scripts, `pypreproc.py` and `pypreproc.py`, would rarely be used on their own. Though if something fails in the execution of the `pylatex.sh` script it might prove useful to run these scripts on their own. Bug reports are always welcome (working on the time proven axiom that no code is ever bug free).

5.3 The merge-tex.py script

A common practice when writing LaTeX documents is to use a driver file to pull in other smaller LaTeX documents (e.g., a driver for a book could pull in the chapters as separate documents). Each of the chapters could contain some active Python code – so how can such code be made visible to the Python preprocessor `pypreproc.py`? That is the job of the `merge-tex.py` script. It will read each file and copy the contents to a single merged file (named `.merged.tex`). This file is then read by `pypreproc.py`. Not all included files will contain active Python code so `merge-tex.py` only looks for lines like `\Input{foo.tex}`. Other files included using `\input` or `\include` are ignored by `merge-tex.py`. Thus the `.merged.tex` need not be a faithful copy of the original source – but that is not a problem since `.merged.tex` exists solely for `pypreproc.py` to harvest the active code after which it is no longer used.

The `merge-tex.py` script is language agnostic – all it does is read its input and respond to any `\Input` lines. If the input file is `foo.tex` then a merged file `bah.tex` can be created using

```
merge-tex.py -i foo.tex -o bah.tex
```

Note that the `.tex` file extensions *are* required (unlike `pypreproc.py` and `pypostproc.py`).

Note also that `merge-tex.py` will follow nested files up to 10 deep. See Example 10 in the later examples section for a trivial example using `\Input`.

6 Usage

Using `pyLaTeX` entails the addition of user defined tags, in the form of Python comments, within the LaTeX source. These tags serve to connect the Python output to the LaTeX source. A typical example is shown in the following code.

[1] It may retain a few comments.

```
\documentclass[12pt]{pylatex}
\begin{document}
   \begin{python}

      from sympy import *
      x = Symbol('x')
      ans    = exp(-x)                # py (ans.01,ans)
      taylor = ans.series(x, 0, 4)    # py (ans.02,taylor)
      taylor = ans.series(x, 0, 8)    # py (ans.03,taylor)

   \end{python}
   \begin{align*}
      &\py*{ans.01}\\
      &\py*{ans.02}\\
      &\py*{ans.03}
   \end{align*}
\end{document}
```

This will lead to a pdf file that contains not just the Python output but also, for reference, a copy of the Python source (though this can be suppressed, see the section below on customisation).

```
from sympy import *
x = Symbol('x')
ans    = exp(-x)                # py (ans.01,ans)
taylor = ans.series(x, 0, 4)    # py (ans.02,taylor)
taylor = ans.series(x, 0, 8)    # py (ans.03,taylor)
```

$$\texttt{ans.01} := e^{-x}$$

$$\texttt{ans.02} := 1 - x + \frac{x^2}{2} - \frac{x^3}{6} + O\left(x^4\right)$$

$$\texttt{ans.03} := 1 - x + \frac{x^2}{2} - \frac{x^3}{6} + \frac{x^4}{24} - \frac{x^5}{120} + \frac{x^6}{720} - \frac{x^7}{5040} + O\left(x^8\right)$$

The key elements to note are the `pyLaTeX` tags (e.g., `py(ans.01,ans)`) and the Python environment block. The tags are always part of a Python comment and are always of the form `py(foo,bah)` where `bah` is a Python symbol (or any printable entity) and `foo` is the tag (later used in the LaTeX source). The `pyLaTeX` tag can be any string of characters from the set `a-zA-Z0-9._`. In the above example there are three tags, `ans.01`, `ans.02` and `ans.03`. The dot postfix is not essential but it provides a simple way to record a sequence of Python outputs. These tags are used later in the body of the LaTeX code and are accessed in the above example using the `pyLaTeX` macro `\py`. This macro takes just one argument namely the corresponding `pyLaTeX` tag. The macro `\py{foo}` will expand to the Python output for the tag `foo`. Other macros are provided such as `\py*{foo}` which will expand to `foo :=` followed by the Python output. The full set of macros are described in the section LaTeX macros.

The `pyLaTeX` tags can be placed anywhere within the Python block. In the above example the tags were placed on the same line as the Python code. This is just for aesthetic effect – they could equally well have been located on following lines. The `pyLaTeX` tags should be unique within the LaTeX file. If the same tag name is used more than once the output associated with that tag will be from the last instance of that tag.

Note that the `pypreproc.py` script will interpret `py(foo)` as a shorthand for `py(foo,foo)`.

6.1 Undefined tags

If any of the `pyLaTeX` macros are called for a tag that was *not* defined then (??) will be printed. Note that this may cause a LaTeX syntax error when the (??) appears in an unexpected context (e.g., in cases where the output of `\py` was expected to expand to one or more \\'s).

6.2 Tagged blocks

There is a second form of tag, known as a tagged block, in which a block of Python code is enclosed in a matching `# pyBeg(foo)`, `# pyEnd(foo)` pair. An example of a tagged block can be seen in the following code. The tag is given the name `Pascal` and can be accessed in the LaTeX code using any of the `pyLaTeX` macros, in this case `\py`. The intention of tagged blocks is that they can be used to capture any Python output generated within the tagged block (except the output that is caught by simple tags such as shown in the previous example).

Here is a short example[2] that demonstrates the use of a single tagged block.

```
\documentclass[12pt]{pylatex}
\begin{document}

   \begin{python}

      def build(n):
          if n==0:
             return [1]
          else:
             P = build(n-1)
             return [1] + [P[i]+P[i+1] for i in range(n-1)] + [1]

      def display(n):
          for i in range(n):
              ans = ""
              for j in range(0,len(build(i))):
                 ans = ans + "\m{" + str(build(i)[j]) + "}"
              print (ans+"\cr")

      # pyBeg (Pascal)
      display(7)
      # pyEnd (Pascal)

   \end{python}

   \def\m#1{\hbox to 1cm{\hfill #1\hfill}}
   \halign{\hbox to \textwidth{\hfill#\hfill}\cr\py{Pascal}}

\end{document}
```

[2]This example is very *inefficient* as it uses recursive calls for *every* element in the table.

The output of the above code (excluding the echoed Python code) is Pascal's triangle (as expected).

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1 \\
1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1
\end{array}
$$

All Python code must be contained within the Python environment block, i.e., within a `\begin{python} ... \end{python}` pair. There is no facility to allow *inline* execution of Python code (though this is allowed in other packages, notably `PythonTeX`). The reason for taking this approach was to force a clear separation between native LaTex code and active Python blocks. This makes it easier to locate the Python code. If an inline calculation is needed then it is a simple matter of including the Python code in a Python block and using a tag to catch and display the result.

Tagged blocks can be nested (though it is not clear what use this might serve) and they can contain simple tags. The simple tags will always take priority over tagged blocks in capturing Python output. Tagged blocks should respect the usual rules of begin-end blocks – they can nested but they must not overlap. The Python pre and post processors (within the `pylatex.sh` script) will report errors when tagged blocks are not properly structured (i.e., overlapping, unmatched `pyBeg/pyEnd` items etc.).

6.3 LaTeX macros

The `pyLaTeX` package provides a number commands for typesetting tagged expressions.

`\py{foo}`	Display the value of `foo` without any additional decorations.
`\py*{foo}`	Similar to `\py{foo}` but adds `foo:=` immediately to the left of `foo`.
`\Py{foo}`	Display the value of `foo` and set the equation tag to be (`foo`). Since this macro creates its own equation tag it should not be used in any maths environment that also creates an equation tag (e.g., `align`, `equation` etc.) It can not be used in any of the `breqn` environments (`breqn` will report a package error).
`\Py*[bah]{foo}`	Use this version of `\Py{foo}` in any of the (starred) `breqn` environments. The optional argument `[bah]` specifies the spacing between the end of `foo` and the right hand edge of the tag. The default is `[\hfill]` which forces the tag to the far right.
`\Dmath[bah]{foo}`	This is a shortcut for the `dmath` environment. The optional and required arguments of `\Dmath[bah]{foo}` are passed onto the `dmath` environment. See page (16) for an example.
`\Dmath*[bah]{foo}`	This is similar to `\Dmath[bah]{foo}` but with the one small change that it applies to the `dmath*` environment.
`\ttTag{foo}`	This is a simple macro that typesets a tag name such as `foo` in a small typewriter font enclosed in matching parenthesis, e.g., (`foo`).

`\PySetup{key=value}`	This command takes one argument in the form of a key=value pair. Its purpose is to set global options for the following Python blocks. The current list of options is limited to one key with three values. These are described in the following section.

6.4 Custom formats

The `pyLaTeX` package allows for some minor customisation of the formatting of the Python code. The macro `\PySetup` can be used to set the preferred options using key-value pairs. There is only one key as described in the following table.

`action=show`	Make the Python code visible and active (i.e., cancel the `hide` and `verbatim` actions).
`action=hide`	Hide the python code but still allow the Python code to be active.
`action=verbatim`	This will make the Python code inactive (i.e., it will not be passed to Python).

Multiple calls to `\PySetup` can be made and apply to all subsequent Python environments or until changed by a later call to `\PySetup`. The default value is `action=show`.

The `pyLaTeX` package uses the `listings` package which allows further formatting changes to be made. For example, to add line numbers and a shaded background to the Python code use

```
\lstset{numbers=left,backgroundcolor=\color{lightgrey}}
```

Many other options are available, see the `listings` documentation for full details.

Note that the settings created in a call to `\lstset` will persit until changed in a later call to `\lstset`. To invoke settings localised to a single instance of a code block pass the settings via the optional argument to `\begin{python}`. Here is a simple example that adds lines numbers just for this single code block.

```
\begin{python}[numbers=left]

    def foo(bah):
        print (bah)
        return "done"

\end{python}
```

6.5 Error reporting

None of the packages in this collection treat errors from the active code with any degree of aplomb – at best they report that an error occurred, at worst they silently ignore the error leaving `pdflatex` to pick up the pieces. This is far from ideal. If in doubt, run the main script (e.g., `pylatex.sh`) using the `-k -x` command line options then inspect the file `foo.pytxt` for any errors. If there are errors then you can debug the extracted code `foo.py` later returning the corrected code back to the LaTeX source `foo.tex`.

6.6 Matplotlib and macOS

There is a well known issue when using Matplotlib on macOS where some Matplotlib codes may fail. The Matplotlib documentation provides a number of suggestions including to use `pythonw` rather than `python`. For such cases use the `-P` flag to set the path to the `pythonw` command when invoking the `pylatex.sh`. This has been tried and tested under the Anaconda Python3 environment for macOS.

7 Examples

The `examples` directory contains a small collection of complete `pyLaTeX` codes that showcase the various features of the `pyLaTeX` package. The Python source contains inline comments that result in long lines of text that do not comfortably fit on an A4 page in portrait mode. Thus most of the examples have been formatted in A4 landscape mode (this is the default setting for the `pyLaTeX` class). Consequently it is not practical to include those examples in this file. The following table provides a very brief summary of each example (with hyperlinks to the corresponding pdf's in `../python/examples/`).

Example 1: This is a collection of basic mathematical computations. It contains examples of simple tags and tagged blocks.

Example 2: This is the first of two examples drawn from the `PythonTeX` website (here). It shows a nice example of using a tagged block.

Example 3: This is the second example taken from the `PythonTeX` website (here). It shows a step-by-step computation of a simple triple integral.

Example 4: This example shows how `Matplotlib` and friends can be used to construct a simple xy-plot.

Example 5: This example shows how problems with the placement of equation tags can be resolved by careful choice of the optional argument to `\Py*`.

Example 6: This is a simple example that uses Python to compute the first few iterations of a Newton-Raphson method.

Example 7: This example contains a single tag-block. It captures basic system data.

Example 8: This example shows how to completely hide the Python code as well a trick for hiding just the tags.

Example 9: This example is a proof of concept example. It shows one approach for passing LaTeX information to the Python code. The code is not elegant but it does the job.

Example 10: This example demonstrates the use of the `\Input{foo.tex}` to include separate LaTeX sources.

Mixed: This example shows how a single LaTeX file can contain both Python and Maple code.

Summary: There is no Python code in this example. Instead, the results from the above examples are included by inputting the corresponding `.pytex` file.

8 Other languages

The pyLaTeX package is one of small number of packages in the hybrid LaTeX collection. Other packages provide tools similar to those of pyLaTeX tailored to other languages including Mathematica, Maple, Matlab and Cadabra. Most of the above discussion concerning Python and pyLaTeX carries over with little change to the other supported languages. The changes are minor and relate to the names of the command line scripts, LaTeX macros and some language specific details. The following table shows a map between the Python based macros and their counterparts in the other languages.

Language naming scheme in the hybrid LaTeX collection								
Python	py	pyBeg	pyEnd	\py	\py*	\Py	\Py*	\PySetup
Matlab	mat	matBeg	matEnd	\mat	\mat*	\Mat	\Mat*	\MatSetup
Mathematica	mma	mmaBeg	mmaEnd	\mma	\mma*	\Mma	\Mma*	\MmaSetup
Maple	mpl	mplBeg	mplEnd	\mpl	\mpl*	\Mpl	\Mpl*	\MplSetup
Cadabra	cdb	cdbBeg	cdbEnd	\cdb	\cdb*	\Cdb	\Cdb*	\CdbSetup

The entries in columns two, three and four denote the tag keys for use in an active code block while the remaining columns denote the LaTeX macros that can be used to display the output from the active code. Each of the setup macros takes exactly the same key-value pairs as used by \PySetup.

The main files for each of the languages follow a predictable naming scheme as shown in the following table.

Main files in the hybrid LaTeX collection				
Python	pylatex.sh	pylatex.sty	pypreproc.py	pypostproc.py
Matlab	matlatex.sh	matlatex.sty	matpreproc.py	matpostproc.py
Mathematica	mmalatex.sh	mmalatex.sty	mmapreproc.py	mmapostproc.py
Maple	mpllatex.sh	mpllatex.sty	mplpreproc.py	mplpostproc.py
Cadabra	cdblatex.sh	cdblatex.sty	cdbpreproc.py	cdbpostproc.py

Each of the shell scripts such as mmalatex.sh takes exactly the same command line options as given previously for pylatex.sh with just one exception – the path in cdblatex.sh -Ppath specifies the directory that contains the cadabra2 executable.

9 Language specific issues

Here are a few points to note when including some of these languages in a LaTeX document.

9.1 Python

Python has very strict (but simple) indentation rules that must be observed not only within a single Python block but across all Python blocks. Why? Because all of the Python blocks will be collected into a separate single Python code and that code must observe the Python indentation rules.

9.2 Cadabra

The underscore character should not be used as part of a symbol name. Doing so would risk Cadabra interpreting the following characters as tensor subscripts (which is almost certainly not intended and will raise a Cadabra syntax error). However, tag names such as `foo_bah.101` are allowed as they will be processed by Python and LaTeX. The Cadabra language is based on Python (and LaTeX) and thus inherits Python's indentation rules.

Be aware that some Cadabra output may require the `amsmath` and `amssymb` packages (e.g., when evaluating components of a tensor). These packages are part of the `pyLaTeX` package but when using the `-I` option may need to be explicitly included in the LaTeX source.

9.3 Maple

The symbols `rhs` and `lhs` have special meaning in Maple and thus should not be used to capture output in any active Maple block. However, tag names such as `lhs.101` and `rhs.101` are perfectly acceptable.

Be aware that Maple does not always present its output in exactly the same order from one run to the next. Terms maybe be re-ordered in ways that do not change the final meaning of the expressions. This behaviour will be apparent when running the tests in the Maple example directory.

9.4 Matlab

For a LaTeX source named `foo.tex` the active Matlab code is executed (within the `matlatex.sh` script) using a line of the form[3]

```
matlab -r "foo_; quit" > foo.mattxt
```

Matlab will interpret the argument of the `-r` command line option as a series of Matlab instructions. One consequence is that the LaTeX file name should not contain any characters that could be mistaken for an arithmetic operation, in particular the hyphen character. Thus file names like `foo_bah.tex` are allowed but `foo-bah.tex` will cause a syntax error.

Be aware that the Matlab program does take quite some time to start. This can have a significant impact on the execution times – particularly for short Matlab codes. Fortunately the startup delay is encountered only once and thus may not be an issue for longer Matlab codes.

Matlab (since R2016b) allows functions to be included at the end of a Matlab program. Thus care must be taken when writing a `matLaTeX` source that all of the Matlab functions appear in the final Matlab environment block (since all Matlab blocks will be condensed into a single file before being passed to Matlab).

9.5 Versions

See the `version.txt` file in each of the language specific directories (e.g., matlab/examples/) for version information.

[3]The actual command is not so simple, see the `matlatex.sh` script for full details.

10 Splitting long lines using breqn

On some occasions the output from a computation may be too long to fit comfortably on one line. In such cases some form of line splitting is required. A less than ideal approach is to manually edit the output by inserting carefully chosen line breaks. Not only is this approach tedious and inelegant it also requires the line breaks to be reset whenever the document format is changed (e.g., from A4 to letter size) or when some of the computations are changed (e.g., showing twice as many terms in a series expansion).

A better approach is to use the `breqn` package which uses sophisticated algorithms to find good line breaks with minimal input from the user. In many cases `breqn` will produce an acceptable output. But it is not without its own limitations[4] as some of the following examples will demonstrate. The fact is that there are no algorithms that will guarantee prefect line splitting as there will always be some subjective element in what constitutes a good line break. The following series of examples are intended as guide of what to expect when using `breqn` and also to demonstrate some possible remedies.

Here is a simple example based on a truncated Taylor series expansion of $1/(1+x)$ around $x = 0$.

```
from sympy import *
x = Symbol('x')
ans    = 1/(1+x)                  # py (fun,ans)
taylor = ans.series(x, 0,  6)     # py (ans.101,taylor)
taylor = ans.series(x, 0,  9)     # py (ans.102,taylor)
taylor = ans.series(x, 0, 12)     # py (ans.103,taylor)
taylor = ans.series(x, 0, 15)     # py (ans.104,taylor)
taylor = ans.series(x, 0, 30)     # py (ans.105,taylor)
```

For an expansion with only a handful of terms the usual `align` environment produces an acceptable output as demonstrated in the following code and output.

```
\begin{align*}
   \py{fun} &= \py {ans.101}\tag{\ttTag{ans.101}}\\
            &= \py {ans.102}\tag{\ttTag{ans.102}}\\
            &= \py {ans.103}\tag{\ttTag{ans.103}}
\end{align*}
```

$$\begin{aligned}
\frac{1}{x+1} &= 1 - x + x^2 - x^3 + x^4 - x^5 + O\left(x^6\right) && \texttt{(ans.101)}\\
&= 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 + O\left(x^9\right) && \texttt{(ans.102)}\\
&= 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 - x^9 + x^{10} - x^{11} + O\left(x^{12}\right) && \texttt{(ans.103)}
\end{aligned}$$

In this and the following examples the equation tags have been explicitly matched to the corresponding `sympy` expressions. This makes it somewhat easier to follow the progress of a computation. The normal equation numbering (e.g., (1), (2) etc.) can be recovered by deleting the calls to `\tag{...}`. The `\ttTag` macro, defined in the `pyLaTeX` package, does as its name suggests – typesets the tag in the `\tt` font.

The above works well for short expressions but for longer expressions it is much better to use the `breqn` package. The key points to note in the following example are the use of the `dmath` environment and the manner in which the equation tags have been set.

[4] The `breqn` documentation does note that it is a work in progress.

```
\usepackage{breqn}
...
\begin{dmath}[number={\ttTag{ans.105}}]
   \py {fun} = \py {ans.105}
\end{dmath}
```

The corresponding output is

$$\frac{1}{x+1} = 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 - x^9 + x^{10} - x^{11} + x^{12} - x^{13} + x^{14} - x^{15} + x^{16} - x^{17} + x^{18} - x^{19} + x^{20} - x^{21} + x^{22} - x^{23} + x^{24} - x^{25} + x^{26} - x^{27} + x^{28} - x^{29} + O\left(x^{30}\right) \tag{ans.105}$$

The `dmath` environment is one of a number environments provided by the `breqn` package. Its job is to chose sensible line breaks that produce a pleasant output (though what constitutes *pleasant* may be up for debate). Note that all of the `breqn` environments do *not* support the `\tag` macro for setting equation tags. The only way to do so is through the `number={...}` optional argument as shown in the above example. To suppress the equation label use the `dmath*` environment.

> **Note:** This sentence may seem out of place but its purpose is to demonstrate a claim made later is this documentation that, with a suitably modified label macro, the `hyperref` package can be used to jump to the correct page when clicking on a hyperlink for an equation. For example, clicking on this link (ans.105) should take us to page 17. But now, back to the main discussion.

The `breqn` package also provides a `dgroup` environment. This can be used to group a sequence of `dmath` environments to align the output on a targeted symbol (usually the first equals sign). Here is a short example

```
1 \begin{dgroup}
2    \begin{dmath}[number={\ttTag{ans.103}}] \py {fun}  = \py {ans.103} \end{dmath}
3    \begin{dmath}[number={\ttTag{ans.104}}]         {} = \py {ans.104} \end{dmath}
4    \begin{dmath}[number={\ttTag{ans.105}}]         {} = \py {ans.105} \end{dmath}
5 \end{dgroup}
```

and here is the corresponding output

$$\begin{aligned}
\frac{1}{x+1} &= 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 - x^9 + x^{10} - x^{11} + O\left(x^{12}\right) && \text{(ans.103)}\\
&= 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 - x^9 + x^{10} - x^{11} + x^{12} - x^{13} + x^{14} + O\left(x^{15}\right) && \text{(ans.104)}\\
&= 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 - x^9 + x^{10} - x^{11} + x^{12} - x^{13} + x^{14} - x^{15} + x^{16} \\
&\quad - x^{17} + x^{18} - x^{19} + x^{20} - x^{21} + x^{22} - x^{23} + x^{24} - x^{25} + x^{26} - x^{27} + x^{28} - x^{29} + O\left(x^{30}\right) && \text{(ans.105)}
\end{aligned}$$

There are a few points to be made here. First is the less then ideal placement of tags. Second is the somewhat ungainly syntax. Both of these points will be discussed in more detail shortly. Next, notice the `{}` preceding the equal signs in lines 3 and 4. This supplies an empty left hand side for each equation and is essential to ensure proper alignment (on the equals sign) across all three equations. Unlike other maths environments, `breqn` does not allow you to specify which symbol to use as the alignment target – that choice is made deep within the inner workings of `dmath` and friends. It will choose symbols including `=-+:()`. If you wish to tell `dmath` to not use a particular symbol you can wrap that symbol inside `\hiderel` as in the following example.

```
\begin{dgroup*}
   \begin{dmath*} f(x) \hiderel{=} \py {fun} = \py {ans.101} \end{dmath*}
   \begin{dmath*}                          {} = \py {ans.102} \end{dmath*}
\end{dgroup*}
```

$$f(x) = \frac{1}{x+1} = 1 - x + x^2 - x^3 + x^4 - x^5 + O\left(x^6\right) \qquad \texttt{(ans.101)}$$

$$= 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 + O\left(x^9\right) \qquad \texttt{(ans.102)}$$

Consider now the earlier example in which the equation tags were not well placed and where the syntax was described as ungainly. The syntax can be tidied slightly using one of the features of the `breqn` package – it allows the `dmath` and `dmath*` environments to be wrapped in user defined macros such as

```
\newcommand*{\Dmath}[2][]{\begin{dmath}[#1]#2\end{dmath}}
```

This defines `\Dmath` as a macro that takes one required argument and one optional argument (both of which are passed on to `dmath`). It is not a huge saving but even so it does save a few keystrokes. Here is one of the previous examples rewritten using `\Dmath`.

```
\begin{dgroup}
   \Dmath[number={\ttTag{ans.103}}]{ \py {fun}  = \py {ans.103} }
   \Dmath[number={\ttTag{ans.104}}]{         {} = \py {ans.104} }
   \Dmath[number={\ttTag{ans.105}}]{         {} = \py {ans.105} }
\end{dgroup}
```

The `pyLaTeX` package defines `\Dmath` for the `dmath` environment as well as `\Dmath*` as a shortcut for the `dmath*` environment.

The use of `\Dmath*` does save some keystrokes but it does not fix the problem where the text is obscured by the tag. There seems to be no simple way to tell `breqn` to avoid such problems while retaining `breqn`'s method of placing the tags. The `\Py*` macro avoids this problem by presenting `dmath*` with a concatenated string built from the text (e.g., the output of `\py{foo}`), some flexible white space (usually `\hfill`) and the tag (e.g., `foo`). This string is processed by `dmath*` and line splits are inserted at its discretion. This will guarantee that the all of the text and the tag will be displayed. The user defined whitespace can always be chosen so that the tag does not overlap the text. This is a bare minimum of what could be deemed an acceptable output – non overlap of the text and the tag. The output might not be optimal but at least it is readable. Here is the same example as above but this time using `\Py*` to display the text and the tags.

```
\begin{dgroup*}
   \Dmath*{ \py {fun}  = \Py* {ans.103} }
   \Dmath*{         {} = \Py*[\hskip 2cm] {ans.104} }
   \Dmath*{         {} = \Py*[\hskip 3cm] {ans.105} }
\end{dgroup*}
```

$$\frac{1}{x+1} = 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 - x^9 + x^{10} - x^{11} + O\left(x^{12}\right) \qquad \texttt{(ans.103)}$$

$$= 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 - x^9 + x^{10} - x^{11} + x^{12} - x^{13} + x^{14} + O\left(x^{15}\right) \qquad \texttt{(ans.104)}$$

$$= 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 - x^9 + x^{10} - x^{11} + x^{12} - x^{13} + x^{14} - x^{15} + x^{16} - x^{17} + x^{18} - x^{19} + x^{20} - x^{21} + x^{22} - x^{23} + x^{24} - x^{25} + x^{26} - x^{27} + x^{28} - x^{29} + O\left(x^{30}\right) \qquad \texttt{(ans.105)}$$

The whitespace between the text and the tag is passed through the optional argument of `\Py*` with the default set as `\hfill`. The inner workings of `\Py*` sets the tag using `\llap` and thus the whitespace optional argument measures the space between the right hand edge of the tag and left hand end of the equation text. Note also that this example uses the starred version of `dgroup` and `\Dmath*`. This is to ensure that `breqn` plays no part in forming the tag (as that is handled by `\Py*`).

The `pyLaTeX` package defines both `\Py` and `\Py*`. The `\Py` macro is intended for use only in the starred maths environments (e.g., `align*, equation*` etc.). Like `\Py*`, it also sets the equation tag to match the expression but it does so using standard methods (internally it calls `\tag`). This last point (that `\Py*` uses `\tag`) also means that it can not be used in any of the `breqn` environments (doing so will cause LaTeX to issue a `breqn` package error).

Both `\Py` and `\Py*` support the usual method of labelling and referencing equations using `\label{foo}` and `\ref{foo}`. However, there are good reasons to use a modified version of the `\label` macro (as described below) to improve the accuracy of the label-reference pairs when using hyperlinks (i.e., so that the reference points to the correct page where the label was issued).

10.1 Hyperref, hyperlinks and breqn

The `breqn` package does support the `hyperref` package though as discussed (here) the hyperlinks to an equation may not work as expected (clicking on a link may take you to the wrong page). The suggested fix is to define equation labels via a modified label macro such as the following

```
\newcommand{\pglabel}[1]{\phantomsection\label{#1}}
```

An example of its use might be

```
\begin{dmath}[number={\ttTag{ans.105}}]
   \py{fun} = \py{ans.105} \pglabel{eq:target}
\end{dmath}
```

$$\frac{1}{x+1} = 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 - x^9 + x^{10} - x^{11} + x^{12} - x^{13} + x^{14} - x^{15} + x^{16} - x^{17} + x^{18} - x^{19} + x^{20} - x^{21} + x^{22} - x^{23} + x^{24} - x^{25} + x^{26} - x^{27} + x^{28} - x^{29} + O\left(x^{30}\right) \tag{ans.105}$$

Then clicking on the link defined by `\ref{eq:target}` will bring the reader to this page. An example of such a link was seen earlier in this document (click here to jump back to that page).

11 Licence

12 Version history

v0.2 (2019/08/04)

- Introduced optional arguments for the code block environments. The arguments are passed to `\lstset` and allow settings to be applied to just a single instance of the code block.
- Improved felxibility for placement of tags in Python and Cadabra. Tags can now be placed inside indented code (e.g., inside `if-then-else` blocks).

v0.1 (2018/08/22)

- Initial public release.