Ptex2tex: Flexible Handling of Computer Code in LaTeX Documents

Ilmar Wilbers Hans Petter Langtangen

Simula Research Laboratory
Email: {ilmarw,hpl}@simula.no

May 19, 2011

1 Introduction

1.1 What is Ptex2tex?

Ptex2tex is a tool that allows you to replace LaTeX environment declarations with simple keywords. In a way, Ptex2tex allows you to create LaTeX packages without any sophisticated knowledge on how to write LaTeX packages. The idea behind Ptex2tex is code generation: instead of hiding complicated LaTeX constructions in complex LaTeX packages, one simply generates the necessary LaTeX commands on the fly, from a compact begin-end environment indication in the LaTeX source. This implies that you have to preprocess your LaTeX source to make an ordinary LaTeX file that can be compiled in the usual way.

The main application of Ptex2tex is for inserting verbatim-style computer code in LaTeX documents. The main application of Ptex2tex is for inserting verbatim-style computer code in LaTeX documents. Code can be copied directly from the source files of the software (complete files or just snippets), and output from programs can be created and copied into the documentation as a part of running Ptex2tex. This guarantees that your LaTeX document contains the most recent version of the program code and its output!

With the default Ptex2tex configuration style, you can switch between 30+ styles for computer code within seconds and just recompile your LaTeX files. Even in a several-hundred pages book it takes seconds to consistently change various styles for computer code, terminal sessions, output from programs, etc. This means that you never have to worry about choosing a proper style for computer/verbatim code in your LaTeX document. Just use Ptex2tex and leave the decision to the future. It takes seconds to change your mind anyway.

Let us look at an example. Say that you are writing a book or an article, and there is a certain style of box you tend to use frequently. The annoying thing is that this box has a lot of LaTeX syntax connected with it to make it look like you want. Here is an example involving a blue box:

```
import sys; print 'script name is', sys.argv[0]
```

The LaTeX source code for this box needs quite some statements:

```
\providecommand{\shadedquoteBluesnippet}{}
\renewenvironment{shadedquoteBluesnippet}[1][]{
\definecolor{shadecolor}{rgb}{0.87843, 0.95686, 1.0}
\definecolor{shadetitle}{rgb}{0.5, 0.95686, 1}
\bgroup\rmfamily
\fboxsep=0mm\relax
\begin{shaded}
{{\hfill\tiny\textsf{\textcolor{shadetitle}{Snippet\ \ }}}}
\list{}{\parsep=-2mm\parskip=0mm\topsep=0pt\leftmargin=2mm
\rightmargin=2\leftmargin\leftmargin=4pt\relax}
\item\relax}
{\endlist{\textcolor{shadecolor}{\ }}\end{shaded}\egroup}
\begin{shadedquoteBluesnippet}
\fontsize{9pt}{9pt}
\begin{Verbatim}
import sys; print 'script name is', sys.argv[0]
\end{Verbatim}
\end{shadedquoteBluesnippet}
\noindent
```

With Ptex2tex, the box is crated by just enclosing the text (inside the box) in a certain *environment*, say `mybox`:

```
\bmybox
import sys; print 'script name is', sys.argv[0]
\emybox
```

The LaTeX code corresponding to the `mybox` environment, i.e., how the box should look like, can be defined in a configuration file. It is then easy to work with boxes, typically contining computer code, in a large document and with Ptex2tex just use some environment names rather than the full, complicated LaTeX surroundings. That is, the Ptex2tex approach allows much less code, and makes it easier to concentrate on the contents of the document we are writing and not the LaTeX code necessary to create such a box. Suppose you want the box to have a completely different look:

```
import sys; print 'script name is', sys.argv[0]
```

which in pure LaTeX looks like:

```
\providecommand{\shadedskip}{}
\definecolor{shadecolor}{rgb}{0.87843, 0.95686, 1.0}
\renewenvironment{shadedskip}{
\def\FrameCommand{\colorbox{shadecolor}}\FrameRule0.6pt
\MakeFramed {\FrameRestore}\vskip3mm}{\vskip0mm\endMakeFramed}
\providecommand{\shadedquoteBlue}{}
\renewenvironment{shadedquoteBlue}[1][]{
\bgroup\rmfamily
\fboxsep=0mm\relax
\begin{shadedskip}
\list{}{\parsep=-2mm\parskip=0mm\topsep=0pt\leftmargin=2mm
\rightmargin=2\leftmargin\leftmargin=4pt\relax}
\item\relax}
{\endlist\end{shadedskip}\egroup}\begin{shadedquoteBlue}
\fontsize{9pt}{9pt}
\begin{Verbatim}
import sys; print 'script name is', sys.argv[0]
\end{Verbatim}
\end{shadedquoteBlue}
```

Using plain LaTeX, you would have to replace the LaTeX code, which in this case is located both before and after the code you want to display. In addition, you may have several hundred boxes throughout your document. Changing all the LaTeX code for these boxes would require a lot of work, even if you use search and replace in your editor. Using Ptex2tex you could simply swap `mybox` with `yourbox`, or change the configuration file accordingly if the change is to be made for all boxes using this style. In this way, you do not need to make any changes to the document itself at all!

At this point you may think that definition of new LaTeX environments, stylefiles and packages solves the problem outlined above. This is not always true, as we elaborate on in the "Motivation" section below. Ptex2tex, which is built on LaTeX code generation, apperas to be a simpler and more powerful technology.

Another major advantage of Ptex2tex is that it provides means to include text from files, *or from parts of files*, as well as output from programs ran at the command line at compile time. Say that you are writing a book about programming or a manual for a programming tool. You frequently include computer program code in your document, and also the output of running programs. While writing the document, you might make changes to programs and the output of them. Instead of cutting and pasting both

the actual program and the output every time it is changed, using Ptex2tex, the only thing you need to do is to run `ptex2tex` on the document. Let's look at an example. We include the following in the `.p.tex` document:

```
@@@CODE division.py def@if
The output from running this function with a=2 and b=0 is:
@@@CMD python division.py #0
```

Now, the file `division.py` itself as well as the output from running it will be included in our document, and the result looks as follows:

```
def division(a, b):
    try:
        return a/float(b)
    except ZeroDivisionError:
        print 'cannot divide by zero'
        return None
```

The output from running this function with a=2 and b=0 is:

Terminal

```
cannot divide by zero
None
```

This makes Ptex2tex a very convenient tool, particularly when writing large LaTeX documents about computer programming where one needs to include a lot of programs and code snippets.

1.2 Motivation

Why do we not simply define LaTeX environments and use these directly in our files? We consider Python more powerful and convenient than LaTeX for generating environments. Also, including text from file using search expressions as well as including results from programs ran at the command line are very challenging, if not impossible, tasks in LaTeX. Imagine that you want to include a certain paragraph or code block from a file. Using plain LaTeX this means copying the necessary text from the source file, and pasting it into the LaTeX file. If you make changes to the file you copied from, you have to remember to copy the text you need a second time. Even though there exist ways of including whole files in LaTeX (`verbatiminput`), it is very difficult to include a part of the file. Also, you might want the text to be included to be typeset in a certain way, something that cannot be achieved by a simple `verbatiminput`. The point is that even though these things *could* be done in LaTeX using classes, for instance, writing them in Python is much easier. As Ptex2tex uses Python configuration files to define commands, one does not even have to know Python in order to extend and tailor Ptex2tex.

1.3 Structure of this tutorial

This tutorial is divided in four sections. Following the introduction is section about running the `ptex2tex` program and generating native LaTeX code, followed by a section on including files and output from program execution. Finally, we will explain how the Ptex2tex environments work and how we can configure them.

2 The prepocessing step

When using Ptex2tex, your LaTeX source file must have the extension `.p.tex`. All edits to your text must be done in this file. Running the program `ptex2tex` on the `.p.tex` file produces an ordinary LaTeX file, which can be compiled to a `.dvi` and `.pdf` file in the usual way. However, if you apply any of the `minted` code environments in your `.p.tex` file (`Minted_Python` for instance), you must run `latex` with the option `-shell-escape` option. Most editors will recognize a `.p.tex` file as a LaTeX or TeX file and invoke relevant styles.

Let us start by looking at what happens when we run the command

Terminal

```
ptex2tex test.p.tex
```

on the command line. If the extension of the file is not `.p.tex`, the program will exit with an error message. If no extension is given, that is, we run `ptex2tex test`, it is assumed that we are looking for the file `test.p.tex`.

The first step of Ptex2tex consists of running Trent Mick's `preprocessor`. This is a Python package that allows us to use preprocessor statements in files, just like the preprocessor statements known from the `C` and `C++` languages. These statements take the form of normal comments in the source file, but when running `preprocessor`, these are treated in a special way. For instance, we can include whole files, or include certain blocks of test or code in our file only when a special requirement is met, otherwise this text or code is ignored. The statements need to be on a separate line. The statements that are implemented in the current version (which is 1.1.0 as of January 2009), are:

- `% #define VAR [VALUE]`
- `% #undef VAR`
- `% #ifdef VAR`
- `% #ifndef VAR`
- `% #if EXPRESSION`

- `% #elif EXPRESSION`
- `% #else`
- `% #endif`
- `% #error ERROR_STRING`
- `% #include "FILE"`

The source code for the present document, `doc.p.tex`, includes a few of these statements as an example. (For now, it is not possible to include arguments to `preprocess` within Ptex2tex, meaning that only the input file and output file are specified, with an additional argument to force the preprocessing by using the `-f` flag for `preprocess`). For further documentation, please visit the website for this package[1]. The input file type to `preprocess` is `.p.tex` and the output is `.tmp1`. If `preprocess` is not available on the system, this step is skipped. Preprocessor statements in the file will then be treated as comments, and the file is copied directly to `.tmp1`.

3 Including files and output from program execution

The second step is to examine and execute the Ptex2tex keywords for including files and for including the output from executed programs. For both these two stages, the input file type will be `.tmp1` and the output file type is `.tmp2`.

3.1 `@@@CODE`

A line *starting* with this statement indicates that a file containing a program (that is, computer code) is to be included in the document. If there is only one argument after the statement (the file name), the whole file is included. If a second argument is included, it is split with respect to the character `'@'`. If only an expression in front of this character is given, only the part of the file starting with that argument and ending with the end of the file is included. If an expression after the character is given as well, the part of the file starting with the first expression and ending at the beginning of the second expression is included. This means that the included part ends *before* the text in the second expression, hence that text is *not* included. When searching the file, the first occurrence of the start expression is used. Ptex2tex then starts scanning for the stop expression from the beginning of the location of the start expression, which will denote the end of the region

[1] `http://trentm.com/projects/preprocess`

to be copied into the text. White-spaces in front of and at the end of the expressions before and after `'@'` are ignored.

The searching in the file for start and stop expressions is done using the Python function `find` on strings. Hence, start/stop expressions are compared directly to the text in the file. In the `ctex2tex` script, which Ptex2tex is based on, regular expressions were used.

The environment keys 'pro' and 'sni' are short for 'program' and 'snippet', respectively, indicating that the former is meant to be used for typesetting a complete program, whereas the latter is meant for a program snippet, for instance a function. Both these Ptex2tex environments are for typesetting code. Environment keys here means keywords that Ptex2tex substitutes with actual LaTeX code later. More information about Ptex2tex environments is given in section 4. If the whole file is included, the Ptex2tex environment key 'pro' is used. Otherwise, the environment key 'sni' is used. An exception here is in the case that a second `'@'` is used after the stop expression. This indicates that the 'pro' environment key is to be used, instead of 'sni'. The reason one might wish to typeset only a part of a file as a full program, is that one can use the search expressions for removing details in the full program, such as headers and test functions, that are irrelevant or confusing in the document, but still are considered important for the actual program.

It is important to differ between the `@@@CODE` statement and the include statement provided with the `preprocessor` package. The latter allows us to include a file by `% #include FILE`, but does not embed the copied text in special LaTeX environments, that is, the whole file is included as is.

We will exemplify this for the file `myprog.py`. Let's look at the output of some different options:
`'@@@CODE myprog.py'` includes the whole file:

```
#!/usr/bin/env python

import math, sys

"""
This is a small script containing a simple function.
"""

def myfunc(x):
    result = None
    if isinstance(x, (list, tuple)):
        result = []
        for i in x:
            result.append(math.fabs(i))
    elif isinstance(x, (int, float)):
        result = math.fabs(x)
    return result

print myfunc(-1.0)
print myfunc((2, -3.))
try:
    result = myfunc([eval(x) for x in (sys.argv[1:])])
```

```
    if result: print result
except:
    pass
```

`'@@@CODE myprog.py def myfunc'` and `'@@@CODE myprog.py def myfunc @'` includes everything from the first instance of the string 'def myfunc' to the end of the file:

```
def myfunc(x):
    result = None
    if isinstance(x, (list, tuple)):
        result = []
        for i in x:
            result.append(math.fabs(i))
    elif isinstance(x, (int, float)):
        result = math.fabs(x)
    return result

print myfunc(-1.0)
print myfunc((2, -3.))
try:
    result = myfunc([eval(x) for x in (sys.argv[1:])])
    if result: print result
except:
    pass
```

`'@@@CODE myprog.py if isinstance@elif'` includes everything from the first instance of the string 'if isinstance' *up to* , but not including, the first instance of 'elif' *after* 'if isinstance'.

```
    if isinstance(x, (list, tuple)):
        result = []
        for i in x:
            result.append(math.fabs(i))
```

3.2 @@@DATA

`@@@DATA` is essentially the same as `@@@CODE`, but different environment keys are used: 'pro' becomes 'dat' and 'sni' becomes 'dsni'. These are short for 'Data' and 'Data snippet', respectively. Whereas `@@@CODE` is meant to be used with program code, `@@@DATA` is meant to be used with data files that are not code, for instance input files to, and output files from, computer programs.

3.3 @@@CMD

A line *starting* with the statement `@@@CMD` indicates that we want to include the output from a shell command with command-line arguments. All text after the statement `@@@CMD` will be executed, except the text after (and including) `'#'`. The character after `'#'` defines how much of the commands

we want to execute should be part of the final text to be included in our document. Any whitespaces after the character are ignored. There are five possible values:

1. `'0'` omits the whole command that is to be executed
2. `'1'` the command is included and the full path stripped
3. `'2'` the command is included and the full path is not stripped
4. `'3'` the command is included except for the name of the program being called (the first word) and the full path is stripped
5. `'4'` the command is included except for the name of the program being called (the first word) and the full path is not stripped.

For options `'1'` and `'3'` a simple regular expression search is used to allow statements like `'python code/myprog.py'` to be printed as `'python myprog.py'` and `'myprog.py'`, respectively, that is, we remove the path to the directory where the program is located, allowing us to run programs in a different directory than where we are running Ptex2tex, without the output reflecting this. For options `'3'` and `'4'` the first word is simply stripped, allowing statements like `'python code/myprog.py'` to be printed as `'myprog.py'` and `'code/myprog.py'`, respectively. The default option is `'3'`, and the terminal session is typeset using the environment key 'sys' (for "system").

Again, let us exemplify this. Running `'python myprog.py -1 2 -5'` in a shell gives us the following output:

Terminal

```
1.0
[2.0, 3.0]
[1.0, 2.0, 5.0]
```

Instead of copying and pasting this into the documents we are creating, meaning we would have to update this every time the code in `myprog.py` is changed, we simply add the following line: `'@@@CMD python myprog.py -1 2 -5'`. The result looks like this:

Terminal

```
myprog.py -1 2 -5
1.0
[2.0, 3.0]
[1.0, 2.0, 5.0]
```

Before looking at the additional options, let us assume that the file `myprog.py` is located in a subfolder `myfolder`, meaning that we would have to run the command `'python myfolder/myprog.py -1 2 -5'`. Adding additional options to this commands gives the following results:

`'@@@CMD python myfolder/myprog.py -1 2 -5 # 0'` omits the whole command that is called:

Terminal

```
1.0
[2.0, 3.0]
[1.0, 2.0, 5.0]
```

`'@@@CMD python myfolder/myprog.py -1 2 -5 # 1'` removes the additional folders:

Terminal

```
python myprog.py -1 2 -5
1.0
[2.0, 3.0]
[1.0, 2.0, 5.0]
```

`'@@@CMD python myfolder/myprog.py -1 2 -5 # 2'` doesn't remove anything:

Terminal

```
python myfolder/myprog.py -1 2 -5
1.0
[2.0, 3.0]
[1.0, 2.0, 5.0]
```

`'@@@CMD python myfolder/myprog.py -1 2 -5 # 3'` removes the additional folders and the first word after `'@@@CMD'` (which is `'python'`):

Terminal

```
myprog.py -1 2 -5
1.0
[2.0, 3.0]
[1.0, 2.0, 5.0]
```

`'@@@CMD python myfolder/myprog.py -1 2 -5 # 4'` removes only the first word after `'@@@CMD'`:

Terminal

```
myfolder/myprog.py -1 2 -5
1.0
[2.0, 3.0]
[1.0, 2.0, 5.0]
```

We give a table with an overview of the different Ptex2tex environments that the `@@@` keywords are mapped to in Table 1.

Keyword	Ptex2tex key
@@@CODE without search expressions	pro
@@@CODE with `start` or `start@`	sni
@@@CODE with `start@stop`	sni
@@@CODE with `start@@`	pro
@@@CODE with `start@stop@`	pro
@@@DATA without search expressions	dat
@@@DATA with `start` or `start@`	dsni
@@@DATA with `start@stop`	dsni
@@@DATA with `start@@`	dat
@@@DATA with `start@stop@`	dat
@@@CMD with any parameters	sys

Table 1: Mapping of keywords to environments

4 Ptex2tex environments

This section explains the use of the Ptex2tex environments. Ptex2tex environments consist of text that we want to typeset in a special way (typical computer code), surrounded by keywords for defining the beginning and the end of the text to be typeset. Ptex2tex replaces these keywords with the LaTeX commands that we have configured the Ptex2tex environment in question to be associated with. We will now explain this in more detail.

4.1 How to use the environments

Ptex2tex environments are described by several keywords. In the previous section we looked at the ones starting with `'@@@'`, and explained that these keywords were replaced with different Ptex2tex environment keys. These environment keys can have different names, we already encountered 'pro', 'sni', 'dat', and 'sys'. These are environment keys that are built into Ptex2tex and need to be there, as they are used for the `'@@@'` keywords, but we can define what the LaTeX environment they are translated to should look like, as we will see shortly. In addition, we can add any new environment keys as we please, and some are already defined by default.

Let us have a closer look at the environment key 'pro'. For every environment key, two keywords can be used in the document we are working with. For 'pro' these would be `\bpro` to indicate where we want to start the typesetting for that environment, and the other is `\epro`, indicating the end. In general, for an environment key 'xxx' there will be keywords `\bxxx` and `\exxx` where 'b' and 'e' stand for 'begin' and 'end', respectively. When Ptex2tex is scanning the documents, these two keywords are replaced with the actual LaTeX code that makes up the environment. Let us once again

look at an example. The lines

```
\bpro
//This text is typeset as computer code.
\epro
```

will result in the following box:

```
//This text is typeset as computer code.
```

The advantage is that we now can change the way this box should look like without making any changes in the document itself, as described in the next section. Also, if we want to change the environment to be used, say for instance from 'pro' to 'sys', instead of changing up to several lines of LaTeX code, we simply change the keywords from `\bpro` and `\epro` to `\bsys` and `\esys`. This step results in a `.tex` file that can be compiled with LaTeX. Normally, this is done in the usual way: !bsys Unix/DOS¿ latex myfile.tex !esys However, if the `minted` LaTeX package is included (needed for the `Minted_*` environments in the default `.ptex2tex.cfg` configuration file), one needs to apply the `-shell-escape` option to `latex`: !bsys Unix/DOS¿ latex -shell-espace myfile.tex !esys The Pygments program when `minted.sty` is included, and this program cannot be invoked without the `-shell-escape` option. In the next section we show more of the details behind the environment keys. Note that the `minted` package must be explicitly included in a `usepackage` statement by the writer of the Ptex2tex file.

4.2 The configuration file

As mentioned earlier, the idea of Ptex2tex is to allow the user to write short keywords for parts of the document that are to be typeset in a special way. When running Ptex2tex, these keywords are then replaced with the full LaTeX commands, resulting in plain `.tex` files. For instance may the keyword `\bpy` be replaced with

```
\plin
```

and `\epy` with

```
\elin
```

Some keywords will also output additional LaTeX code for defining LaTeX environments once for every file, more on this later. We will differentiate between Ptex2tex environments, which we are about to explain in detail, and LaTeX environments, which refer to the LaTeX commands that we would have to use instead of the Ptex2tex keywords.

All Ptex2tex environments available are defined in a configuration file. When running Ptex2tex for the first time, or if the file is removed since the last time Ptex2tex was run, this configuration file will be placed in the

user's home directory. The file is named `.ptex2tex.cfg`, indicating that it is hidden. The file is written in the Python ConfigParser language[2] and will be referred to as the global environment file. This file is read every time one runs Ptex2tex. In addition to this file, the `.ptex2tex.cfg` file in the directory where the `.p.tex` file currently being processed is located, is also evaluated, if it exists. This file is referred to as the local environment file. The global environment file is evaluated first, and then the local one. This means that if a specific environment is defined both in the global and local environment files, the environment from the local file will overwrite the one from the global file.

In this way, one can make changes to an environment that will only affect the local `.p.tex` files. At the same time, one can make changes to the global file, but these changes will only matter if the environment it concerns is not overwritten in the local environment files. One can thus add environments to the global file and they will be available when running Ptex2tex from any location.

The configuration file is made up of different sections, one for each environment and one for the mapping of environment keys to the corresponding environment. Each environment can contain the following options:

```
breplace
ereplace
newenv
define
fontsize
bstretch
```

We refer to the `ptex2tex.cfg` file in the `lib/ptex2tex` directory to see a large number of examples of how these environments can be used.

Each section has a heading enclosed by brackets. This heading will also be the name of the environment. The options for this environment should follow on the lines below. For instance, to define the Ptex2tex environment 'Extra' and the option `breplace`, we would write:

```
[Extra]
breplace = \begin{Verbatim}
```

In addition to the sections defining the environments, there is a section `[names]`. In this section we map environment keys to Ptex2tex environments. For instance, if we want the key 'pro' to be associated with the environment 'Bluebox', we write:

```
[names]
pro = Bluebox
```

Hence we can use `\bpro` to indicate the beginning of the 'pro' environment, and `\epro` to indicate the end. We can set multiple keys to point to the same environment:

[2] http://docs.python.org/lib/module-ConfigParser.html

```
[names]
pro = Bluebox
tmp = Bluebox
```

If we try to assign multiple environment to the same key, only the last one is used:

```
[names]
pro = Bluebox
pro = Graybox
```

4.3 Defining new environments

A Ptex2tex environment in the configuration file is defined by a section heading and several options. A minimum requirement of an environment is that the options `breplace` and `ereplace` are defined. Hence, for the environment 'Extra', the following lines should exist and be defined:

```
[Extra]
breplace = \begin{verbatim}
erplace = \end{verbatim}
```

In this case they define a simple `Verbatim` environment. `breplace` and `ereplace` control what LaTeX commands that should be returned at the beginning and at the end of the Ptex2tex environment, respectively. Note that only defining the environment is not enough, the keywords to be used with this environment also need to be defined in the 'names' section:

```
[names]
ext = Extra
```

Now, the keywords `\bext` and `\eext` will be replaced with the text defined in `breplace` and `ereplace`.

We differ between two categories of environments in Ptex2tex: those that make use of the `\newenvironment` keywords in LaTeX, and those who don't, but instead use other LaTeX commands like `\minipage` and different packages like `fancyverb` or `framed` as well as different box environments.

The problem with Ptex2tex environments that use `\newenvironment` is that LaTeX needs this environment to be defined, but not more than once, or it will issue an error. Therefore, when replacing the first occurrence of each environment keyword (for instance 'bpro'), a definition is added as well. This is the keyword `newenv` in the section for an environment. Default, this is an empty string. When running Ptex2tex on multiple files, and then combining these files in a single document, this environment will be defined multiple times, once for every `.p.tex` document. Since we run Ptex2tex on one file at the time, Ptex2tex cannot know if a specific environment is defined earlier.

In order to get around this limitation, we use `\renewenvironment` instead of `\newenvironment`. But since it is not possible to renew an environment unless it already is defined, this by itself does not solve the problem. But if we first add the line

```
\providecommand{\shadedquote}{}
```

we have created a way around this. What it does is to create a command `shadedquote` given that `shadedquote` is not already defined. There is no equivalent
(`\provideenvironment`) for environments, but since `\renewenvironment` simply checks if the name of the environment (`shadedquote`) is defined, not if it is defined as an environment, this works.

When defining new environments as described above, the name of the environment must be unique in the configuration file. For example, "shadedquote" must not be defined in another environment. Ptex2tex is normally able to detect such problems.

It is also possible to use an environment that is defined elsewhere, say in third-party LaTeX package. One can then use the option `define`. If it is set to `False`, it is assumed that the LaTeX environment is defined externally, and Ptex2tex will not define this environment.

There are two types of boxes supported by the standard configuration file `ptex2tex.cfg`. One kind (e.g., 'Blue') allows the text in the box to be split over pages, while the other kind (e.g., 'Blue_sp') does not allow splitting, which often forces LaTeX to move the box to the next space, leaving a lot of white space on the preceding page.

Please note that the names of the environments should not contain numbers, only letters. Naming an environment `\shadedquote2` makes the interpreter think that we are using `\shadedquote` followed by the number 2.

It is possible to use so-called variable interpolation in the configuration file. This means that one can use the name of one option in another option in the section, and the value for that option will then be included in the first option, for example:

```
[SomeBox]
breplace = \begin{Verb}[baselinestretch=%(bstretch)s]
bstretch=0.85
```

What happens here is that the `%(bstretch)s` part is substituted with the value of the option 'bstretch'. The `'%'` indicates that the variable within the following parenthesis is to be interpolated. The 's' after the last parenthesis means that the option to be inserted is a string. See the `ptex2tex.cfg` file in `lib/ptex2tex` for numerous examples.

Often, the option text for 'breplace' or 'ereplace' spans several lines (of LaTeX code). It is then important that the text for the lines following the first use the same indentation. It is possible to comment out some of the lines in the configuration file by using a `'#'` at the beginning of the line. Any other position won't work, as it will be parsed as part of the option instead. If you need to use comments in the LaTeX code, you need to use two `'%'`, as only one will be considered as variable interpolation, and will likely cause an error. In the previous code example we show some of the mentioned points.

For further documentation of Python Config Files, see the Python Library Reference.

All options within a section of the configuration file are parsed within Ptex2tex. You can define your own options beyond the six mentioned, but this will result in a warning.

4.4 Remarks about LaTeX packages

Note that some of the environments present in the environment configuration file require certain LaTeX packages to be available. Specifically, you should use the following lines in the header and make sure the corresponding packages are installed on your system:

```
\usepackage{relsize,fancyvrb,moreverb,epsfig,framed}
\usepackage{color,listings,codehighlight}
%\usepackge{minted}
```

Many of the stylefiles are found in the `latex/styles` directory of the Ptex2tex source. For example, `codehighlight.sty` is only found here and must be copied to the right directory for LaTeX packages.

The `minted` package requires the Python tool `pygments` to be installed and `latex` to be run with the extra option `-shell-escape`. Therefore the `minted` package is optional. It is only demanded for the `Minted_*` environments in the default `.ptex2tex.cfg` configuration file.

The Ptex2tex package comes with a LaTeX style file `ptex2tex.sty` containing the above `usepackage` commands, plus some tweaks of packages. If the `ptex2tex.sty` file is installed correctly so that LaTeX can find it, one can simply use

```
\usepackage{ptex2tex}
```

or

```
\usepackage{ptex2tex,minted}
```

if the `Minted_*` environments are desired for typesetting code.

One can also just copy the `ptex2tex.sty` file to the current working directory. This file, as well as the `.eps` files used for the Ptex2tex environments 'Warnings', 'Rules', and 'Summation', are located in the folder `latex` in the source code for Ptex2tex. The installation script for the package will try to install these to the computers' LaTeX folders, see the README file.

4.5 Additional keywords

In addition to the keywords starting with `@@@` and the keywords associated with beginning and ending Ptex2tex environments, there is another keyword available, namely `\code`. This keyword can be used anywhere in the text, as long as it is contained within a single line. This allows the use of any

character, we can for instance write `\code{@#_%}` without LaTeX giving us any errors. The result is similar to the `\texttt` LaTeX command, but it has nicer typesetting of (for instance) underscores. The `code` command is really just a slightly modified normal inline verbatim construction where the font size can be controlled. Moreover, the escape character in LaTeX, which is `\`, is removed in front of the characters `'#'`, `'%'`, `'@'`, `'$'` and `'_'`. Writing `\code{%}` results in the text for the rest of the line after the character `%` being marked as a comment in most editors, which is confusing. Writing `\code{$}` results in the text for the rest of the document being marked as a math environment. Using `\code{\%}` and `\code{\$}` instead gives us the same result in the final document without this problem appearing in our editor.

4.6 Mapping of keywords to environments

The default mapping between Ptex2tex keywords and environments is included for reference:

```
# computer code in quote environment (gives a left margin):
ccq  = CodeIndented
# computer code with no left margin:
cc   = Code
# computer code with line numbering:
ccl  = CodeLineNo
# program box:
pro  = BlueBar
pypro  = Minted_Python
cpppro = Minted_Cpp
cpro   = Minted_C
fpro   = Minted_Fortran
# computer code box (snippet, not complete program):
cod  = Blue
pycod  = Minted_Python
cppcod = Minted_Cpp
ccod   = Minted_C
fcod   = Minted_Fortran
# computer code box (snippet, not complete program):
sni  = Blue
# data file:
dat  = CodeIndented
# data file snippet:
dsni = CodeIndented
# system commands (in terminal window):
sys  = CodeTerminal
# one-line system command (in terminal window):
slin = Code
# IPython interactive session:
ipy  = Code
# standard interactive python session:
py   = Code
# execution of a Python program ("run python"):
rpy  = CodeTerminal
# one-line program code:
plin = Code
# verbatim environment:
ver  = Verb
# warning box:
warn = Warnings
# tip box:
rule = Tip
# note box:
summ = Note
```

You are free to define completely new keywords (and environments) in the configuration file.

4.7 Demo of the different environments

There is a test script `testconfig.py` that can read a configuration file and make a LaTeX demo of all environments in that file. Just run the script in a directory with a `.ptex2tex.cfg` file. The result is a file `tmp_names` with definition of new environment keys pointing to all environments found in the configuration file. This `tmp_names` can be appended to your `.ptex2tex.cfg`. The script `testconfig.py` also makes a file `tmp_latex` which can be copied into any LaTeX in Ptex2tex format (i.e., a `.p.tex` file) to see a demo of the environments. Below is such a LaTeX demo.
Here is a demo of the environment `Verb`:

```
# Here is some Python code

def height_and_velocity(t, v0):
    """Invoke some advanced math computations."""
    g = 9.81                  # acceleration of gravity
    y = v0*t - 0.5*g*t**2     # vertical position
    v = v0 - g*t              # vertical velocity
    return y, v

class Wrapper:
    def __init__(self, func, alternative_kwarg_names={}):
        self.func = func
        self.help = alternative_kwarg_names

    def __call__(self, *args, **kwargs):
        # Translate possible alternative keyword argument
        # names in kwargs to those accepted by self.func:
        func_kwargs = {}
        for name in kwargs:
            if name in self.help:
                func_kwargs[self.help[name]] = kwargs[name]
            else:
                func_kwargs[name] = kwargs[name]

        return self.func(*args, **func_kwargs)

height_and_velocity = Wrapper(height_and_velocity,
                              {'time': 't',
                               'velocity': 'v0',
```

```
                                 'initial_velocity': 'v0'})

print height_and_velocity(initial_velocity=0.5, time=1)
```

Here is a demo of the environment `CodeRule`:

Code

```
# Here is some short Python code

def height_and_velocity(t, v0):
    """Invoke some advanced math computations."""
    g = 9.81                  # acceleration of gravity
    y = v0*t - 0.5*g*t**2     # vertical position
    v = v0 - g*t              # vertical velocity
    return y, v

print height_and_velocity(initial_velocity=0.5, time=1)
```

Here is a demo of the environment `CodeTerminal`:

Terminal

```
# Here is some short Python code

def height_and_velocity(t, v0):
    """Invoke some advanced math computations."""
    g = 9.81                  # acceleration of gravity
    y = v0*t - 0.5*g*t**2     # vertical position
    v = v0 - g*t              # vertical velocity
    return y, v

print height_and_velocity(initial_velocity=0.5, time=1)
```

Here is a demo of the environment `Code`:

```
# Here is some Python code

def height_and_velocity(t, v0):
    """Invoke some advanced math computations."""
    g = 9.81                  # acceleration of gravity
    y = v0*t - 0.5*g*t**2     # vertical position
    v = v0 - g*t              # vertical velocity
    return y, v

class Wrapper:
    def __init__(self, func, alternative_kwarg_names={}):
        self.func = func
        self.help = alternative_kwarg_names

    def __call__(self, *args, **kwargs):
        # Translate possible alternative keyword argument
        # names in kwargs to those accepted by self.func:
        func_kwargs = {}
        for name in kwargs:
            if name in self.help:
                func_kwargs[self.help[name]] = kwargs[name]
            else:
                func_kwargs[name] = kwargs[name]

        return self.func(*args, **func_kwargs)

height_and_velocity = Wrapper(height_and_velocity,
                              {'time': 't',
                               'velocity': 'v0',
                               'initial_velocity': 'v0'})

print height_and_velocity(initial_velocity=0.5, time=1)
```

Here is a demo of the environment `CodeLineNo`:

```
# Here is some Python code

def height_and_velocity(t, v0):
    """Invoke some advanced math computations."""
    g = 9.81                  # acceleration of gravity
    y = v0*t - 0.5*g*t**2     # vertical position
    v = v0 - g*t              # vertical velocity
    return y, v

class Wrapper:
    def __init__(self, func, alternative_kwarg_names={}):
        self.func = func
        self.help = alternative_kwarg_names

    def __call__(self, *args, **kwargs):
        # Translate possible alternative keyword argument
        # names in kwargs to those accepted by self.func:
        func_kwargs = {}
        for name in kwargs:
            if name in self.help:
                func_kwargs[self.help[name]] = kwargs[name]
            else:
                func_kwargs[name] = kwargs[name]

        return self.func(*args, **func_kwargs)

height_and_velocity = Wrapper(height_and_velocity,
                              {'time': 't',
                               'velocity': 'v0',
                               'initial_velocity': 'v0'})

print height_and_velocity(initial_velocity=0.5, time=1)
```

Here is a demo of the environment `CodeIndented`:

```
    # Here is some Python code

    def height_and_velocity(t, v0):
        """Invoke some advanced math computations."""
        g = 9.81                  # acceleration of gravity
        y = v0*t - 0.5*g*t**2     # vertical position
        v = v0 - g*t              # vertical velocity
        return y, v

    class Wrapper:
        def __init__(self, func, alternative_kwarg_names={}):
            self.func = func
            self.help = alternative_kwarg_names

        def __call__(self, *args, **kwargs):
            # Translate possible alternative keyword argument
            # names in kwargs to those accepted by self.func:
            func_kwargs = {}
            for name in kwargs:
                if name in self.help:
                    func_kwargs[self.help[name]] = kwargs[name]
                else:
                    func_kwargs[name] = kwargs[name]

            return self.func(*args, **func_kwargs)

    height_and_velocity = Wrapper(height_and_velocity,
                                  {'time': 't',
                                   'velocity': 'v0',
                                   'initial_velocity': 'v0'})

    print height_and_velocity(initial_velocity=0.5, time=1)
```

Here is a demo of the environment `CodeIndented10`:

```
    # Here is some Python code
```

```
def height_and_velocity(t, v0):
    """Invoke some advanced math computations."""
    g = 9.81                  # acceleration of gravity
    y = v0*t - 0.5*g*t**2     # vertical position
    v = v0 - g*t              # vertical velocity
    return y, v

class Wrapper:
    def __init__(self, func, alternative_kwarg_names={}):
        self.func = func
        self.help = alternative_kwarg_names

    def __call__(self, *args, **kwargs):
        # Translate possible alternative keyword argument
        # names in kwargs to those accepted by self.func:
        func_kwargs = {}
        for name in kwargs:
            if name in self.help:
                func_kwargs[self.help[name]] = kwargs[name]
            else:
                func_kwargs[name] = kwargs[name]

        return self.func(*args, **func_kwargs)

height_and_velocity = Wrapper(height_and_velocity,
                              {'time': 't',
                               'velocity': 'v0',
                               'initial_velocity': 'v0'})

print height_and_velocity(initial_velocity=0.5, time=1)
```

Here is a demo of the environment `PyHighlight`:

```
# Here is some Python code

def height_and_velocity(t, v0):
    """Invoke some advanced math computations."""
    g = 9.81                  # acceleration of gravity
    y = v0*t - 0.5*g*t**2     # vertical position
    v = v0 - g*t              # vertical velocity
    return y, v

class Wrapper:
    def __init__(self, func,
        alternative_kwarg_names={}):
        self.func = func
        self.help = alternative_kwarg_names

    def __call__(self, *args, **kwargs):
        # Translate possible alternative keyword
            argument
        # names in kwargs to those accepted by
            self.func:
        func_kwargs = {}
        for name in kwargs:
            if name in self.help:
                func_kwargs[self.help[name]] =
                    kwargs[name]
            else:
                func_kwargs[name] = kwargs[name]
```

```
        return self.func(*args, **func_kwargs)

height_and_velocity = Wrapper(height_and_velocity,
                              {'time': 't',
                               'velocity': 'v0',
                               'initial_velocity':
                                   'v0'})

print height_and_velocity(initial_velocity=0.5, time=1)
```

Here is a demo of the environment `CppHighlight`:

```
# Here is some Python code

def height_and_velocity(t, v0):
    """Invoke some advanced math computations."""
    g = 9.81                    # acceleration of gravity
    y = v0*t - 0.5*g*t**2       # vertical position
    v = v0 - g*t                # vertical velocity
    return y, v

class Wrapper:
    def __init__(self, func,
        alternative_kwarg_names={}):
        self.func = func
        self.help = alternative_kwarg_names

    def __call__(self, *args, **kwargs):
        # Translate possible alternative keyword
            argument
        # names in kwargs to those accepted by
            self.func:
        func_kwargs = {}
        for name in kwargs:
            if name in self.help:
                func_kwargs[self.help[name]] =
                    kwargs[name]
            else:
                func_kwargs[name] = kwargs[name]

        return self.func(*args, **func_kwargs)

height_and_velocity = Wrapper(height_and_velocity,
                              {'time': 't',
                               'velocity': 'v0',
                               'initial_velocity':
                                   'v0'})

print height_and_velocity(initial_velocity=0.5, time=1)
```

Here is a demo of the environment `PyMatlab`:

```
# Here is some Python code

def height_and_velocity(t, v0):
    """Invoke some advanced math computations."""
    g = 9.81                    # acceleration of gravity
    y = v0*t - 0.5*g*t**2       # vertical position
    v = v0 - g*t                # vertical velocity
    return y, v

class Wrapper:
    def __init__(self, func, alternative_kwarg_names={}):
        self.func = func
        self.help = alternative_kwarg_names

    def __call__(self, *args, **kwargs):
        # Translate possible alternative keyword argument
        # names in kwargs to those accepted by self.func:
        func_kwargs = {}
        for name in kwargs:
            if name in self.help:
                func_kwargs[self.help[name]] =
                   kwargs[name]
            else:
                func_kwargs[name] = kwargs[name]

        return self.func(*args, **func_kwargs)

height_and_velocity = Wrapper(height_and_velocity,
                              {'time': 't',
                               'velocity': 'v0',
                               'initial_velocity': 'v0'})

print height_and_velocity(initial_velocity=0.5, time=1)
```

Here is a demo of the environment PyBash:

```
# Here is some Python code

def height_and_velocity(t, v0):
    """Invoke some advanced math computations."""
    g = 9.81                    # acceleration of gravity
    y = v0*t - 0.5*g*t**2       # vertical position
    v = v0 - g*t                # vertical velocity
    return y, v

class Wrapper:
    def __init__(self, func,
        alternative_kwarg_names={}):
        self.func = func
        self.help = alternative_kwarg_names

    def __call__(self, *args, **kwargs):
```

```
        # Translate possible alternative keyword
            argument
        # names in kwargs to those accepted by
            self.func:
        func_kwargs = {}
        for name in kwargs:
            if name in self.help:
                func_kwargs[self.help[name]] =
                    kwargs[name]
            else:
                func_kwargs[name] = kwargs[name]

        return self.func(*args, **func_kwargs)

height_and_velocity = Wrapper(height_and_velocity,
                              {'time': 't',
                               'velocity': 'v0',
                               'initial_velocity':
                                   'v0'})

print height_and_velocity(initial_velocity=0.5, time=1)
```

Here is a demo of the environment `PySWIG`:

```
# Here is some Python code

def height_and_velocity(t, v0):
    """Invoke some advanced math computations."""
    g = 9.81                    # acceleration of gravity
    y = v0*t - 0.5*g*t**2       # vertical position
    v = v0 - g*t                # vertical velocity
    return y, v

class Wrapper:
    def __init__(self, func,
        alternative_kwarg_names={}):
        self.func = func
        self.help = alternative_kwarg_names

    def __call__(self, *args, **kwargs):
        # Translate possible alternative keyword
            argument
        # names in kwargs to those accepted by
            self.func:
        func_kwargs = {}
        for name in kwargs:
            if name in self.help:
                func_kwargs[self.help[name]] =
                    kwargs[name]
            else:
                func_kwargs[name] = kwargs[name]

        return self.func(*args, **func_kwargs)
```

```
26
height_and_velocity = Wrapper(height_and_velocity,
28                                {'time': 't',
                                  'velocity': 'v0',
30                                'initial_velocity':
                                     'v0'})

32 print height_and_velocity(initial_velocity=0.5, time=1)
```

Here is a demo of the environment `Minted_Python`:

```
# Here is some Python code

def height_and_velocity(t, v0):
    """Invoke some advanced math computations."""
    g = 9.81                  # acceleration of gravity
    y = v0*t - 0.5*g*t**2     # vertical position
    v = v0 - g*t              # vertical velocity
    return y, v

class Wrapper:
    def __init__(self, func, alternative_kwarg_names={}):
        self.func = func
        self.help = alternative_kwarg_names

    def __call__(self, *args, **kwargs):
        # Translate possible alternative keyword argument
        # names in kwargs to those accepted by self.func:
        func_kwargs = {}
        for name in kwargs:
            if name in self.help:
                func_kwargs[self.help[name]] = kwargs[name]
            else:
                func_kwargs[name] = kwargs[name]

        return self.func(*args, **func_kwargs)

height_and_velocity = Wrapper(height_and_velocity,
                              {'time': 't',
                               'velocity': 'v0',
                               'initial_velocity': 'v0'})

print height_and_velocity(initial_velocity=0.5, time=1)
```

5 Support

Please contact `ilmarw@simula.no` for bug reports, feature requests and general help.