DOCUMENTATION FOR MATRIX2LATEX

ØYSTEIN BJØRNDAL

Takes a python matrix or nested list and converts to a LaTeX table or matrix. Author: ob@cakebox.net, inspired by the work of koehler@in.tum.de who has written a similar package for matlab `http://www.mathworks.com/matlabcentral/fileexchange/4894-matrix2latex`

1. TODO

- Provide installation.
- Remove dependency on numpy (might be more portable to other systems) and will allow for more flexible input (for instance non-rectangular).
- Complete test suite.
- Improve error handling by fixing errors instead of crashing.
- Clean the preamble for this documentation.
- Code is given in all kinds of fonts in this document. Fix!

2. Installation

The following packages and definitions are recommended in the latex preamble

```
% scientific notation, 1\e{9} will print as 1x10^9
\providecommand{\e}[1]{\ensuremath{\times 10^{#1}}}
\usepackage{amsmath} % needed for pmatrix
\usepackage{booktabs} % Fancy tables
...
\begin{document}
...
```

To use the code make sure all python files are in your `PYTHONPATH`, note that numpy is required though this may change in the near future (and even sooner by request).

3. Arguments

3.1. **matrix.** A numpy matrix or a nested list

3.2. **Filename.** File to place output, extension .tex is added automatically. File can be included in a LaTeX document by `\input{filename}`. If filename is None or not a string, output will be returned in a string

3.3. ***environments.** Use `matrix2latex(m, None, "align*", "pmatrix", ...)` for matrix. This will give

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Use `matrix2latex(m, "test", "table", "center", "tabular"...)` for table. Table is default so given no arguments: table, center and tabular will be used. The above command is then equivalent to
`matrix2latex(m, "test", ...)`

3.3.1. *Example.*

```
from matrix2latex import matrix2latex
m = [[1, 2, 3], [1, 4, 9]] # python nested list
t = matrix2latex(m)
print t
```

```
\begin{table}[ht]
  \begin{center}
    \begin{tabular}{cc}
      \toprule
      $1$ & $1$\\
      $2$ & $4$\\
      $3$ & $9$\\
      \bottomrule
    \end{tabular}
  \end{center}
\end{table}
```

1	1
2	4
3	9

3.4. ****keywords.**

3.4.1. *rowLabels.* A row at the top used to label the columns. Must be a list of strings.

Using the same example from above we can add row labels

```
rl = ['$x$', '$x^2$']
t = matrix2latex(m, rowLabels=rl)
```

x	x^2
1	1
2	4
3	9

3.4.2. *columnLabels.* A column used to label the rows. Must be a list of strings

3.4.3. *transpose.* Flips the table around in case you messed up. Equivalent to matrix2latex(m.H, ...) if m is a numpy matrix. Note the use of columnLabels in the example.

```
cl = ['$x$', '$x^2$']
t = matrix2latex(m, columnLabels=cl, transpose=True)
```

x	1	2	3
x^2	1	4	9

3.4.4. *caption.* Use to define a caption for your table. Inserts `\caption` after `\end{tabular}`. Always use informative captions!

```
t = matrix2latex(m, rowLabels=rl,
                 caption='Nice table!')
```

TABLE 1. Nice table!

x	x^2
1	1
2	4
3	9

3.4.5. *label.* Used to insert `\label{tab:...}` after `\end{tabular}` Default is filename without extension.

We can use `label='niceTable'` but if we save it to file the default label is the filename, so:

```
matrix2latex(m, 'niceTable', rowLabels=rl,
                 caption='Nice table!')
```

can be referenced by `\ref{tab:niceTable}`. Table 2 was included in latex by `\input{niceTable}`.

TABLE 2. Nice table!

x	x^2
1	1
2	4
3	9

3.4.6. *format.* Printf syntax format, e.g. \$%.2f\$. Default is \$%g\$. This format is then used for all the elements in the table.

```
m = [[1, 2, 3], [1, 1/2, 1/3]]
rl = ['$x$', '$1/x$']
t = matrix2latex(m, rowLabels=rl,
                 format='%.2f')
```

x	$1/x$
1.00	1.00
2.00	0.50
3.00	0.33

3.4.7. *formatColumn.* A list of printf-syntax formats, e.g. [\$%.2f\$, \$%g\$] Must be of same length as the number of columns. Format i is then used for column i. This is useful if some of your data should be printed with more significant figures than other parts

```
t = matrix2latex(m, rowLabels=rl,
                 formatColumn=['%g', '%.2f'])
```

x	$1/x$
1	1.00
2	0.50
3	0.33

3.4.8. *alignment.* Used as an option when tabular is given as enviroment. `\begin{tabular}{alignment}` A latex alignment like c, l or r. Can be given either as one per column e.g. "ccc". Or if only a single character is given e.g. "c", it will produce the correct amount depending on the number of columns. Default is "c", if you use columnLabels it will always use "r" as the alignment for that column.

```
m = [[1, 1/2, 1/3], [1, 1/2, 1/3]]
t = matrix2latex(m, alignment='rc')
```

1	1
0.5	0.5
0.333333	0.333333

But what if I want vertical rules in my table? Well, this package is built on top of booktabs for publication ready tables and the booktabs documentation clearly states “Never, ever use vertical rule”. But as long as you are not publishing your table, you could simply use

```
t = matrix2latex(m, alignment='|r||c|')
```

1	1
0.5	0.5
0.333333	0.333333

3.4.9. *General considerations.* Note that many of these options only has an effect when typesetting a table, if the correct environment is not given the arguments are simply ignored. To give an example of a very useless function call

```
t1 = matrix2latex(m, None, "align*", "pmatrix",
                  alignment='rc',
                  caption='hello world',
                  label='test')
# produces the exact same thing as
t2 = matrix2latex(m, None, "align*", "pmatrix")
assert t1 == t2
```

$$\begin{pmatrix} \texttt{1} & \texttt{1} \\ \texttt{0.5} & \texttt{0.5} \\ \texttt{0.333333} & \texttt{0.333333} \end{pmatrix}$$

The scary thing is that rowLabels actually works when creating a matrix, it just looks a bit wierd.

The options presented by this program represents what I need when creating a table, if you need a more sophisticated table you must either change the python code (feel free to submit a patch) or manually adjust the output afterwards. `http://en.wikibooks.org/wiki/LaTeX/Tables` gives an excellent overview of some advanced table techniques.

The booktabs.pdf documentation is an excellent guide to proper table creation, matrix2latex does not incorporate all the features of this package (yet).

4. Usage examples

The usefulness of a programming interface to create LaTeX tables becomes apparent when the data is dynamically created by python. This can be either because you want flexibility with respect to the tables size or because the table content is somehow created by python.

One day you decide to compare different implementations of the factorial functions, you start by writing the following file as `factorial.py`

```python
# built in factorial
from math import factorial as factorialMath

# recursive
def factorialRecursive(n):
    if n == 0:
        return 1
    elif n == 1:
        return n
    else:
        return n*factorialRecursive(n-1)

# sequential
def factorialSequential(n):
    if n == 0:
        return 1
    res = 1
    for k in xrange(2, n+1):
        res *= k
    return res
```

The first thing to do is to verify that the three implementations actually give the same results, for this we simply loop over the different functions and try for different values of n. The result is shown in table 3.

```python
if __name__ == '__main__':
    from matrix2latex import matrix2latex

    N = range(0, 10)
    table = list()
    for func in (factorialMath,
                 factorialRecursive,
                 factorialSequential):
        row = list()
        for n in N:
            res = func(n)  # call func
            row.append(res)# append result to row
        table.append(row)  # append row to table
```

```
# convert to string for labeling
cl = ["${n}$".format(n=n) for n in N]
# row labels
rl = ['$n$', 'Built-in', 'Recursive', 'Sequential']
caption = '''Vertifying that the different factorial
implementations gives the same results'''
matrix2latex(table, 'facV', caption=caption,
             columnLabels=cl, rowLabels=rl,
             alignment='r')
```

TABLE 3. Vertifying that the different factorial implementations gives the same results

$n$	Built-in	Recursive	Sequential
0	1	1	1
1	1	1	1
2	2	2	2
3	6	6	6
4	24	24	24
5	120	120	120
6	720	720	720
7	5040	5040	5040
8	40320	40320	40320
9	362880	362880	362880

What we really wanted to do was to compare the speed of the different implementations. To do this we use the python package timeit, the rest of the code is mostly unchanged.

```
import timeit
table = list()
for func in ('factorialMath',
             'factorialRecursive',
             'factorialSequential'):
    row = list()
    for n in N:
        statement = 'factorial.{func}({n})'.format(func=func,
                                                   n=n)
        setup = 'import factorial'
        # measure time
        res = timeit.repeat(statement, setup)
        row.append(min(res)) # append result
    table.append(row) # append row to table
```

```
# convert to string for labeling
cl = ["${n}$".format(n=n) for n in N]
rl = ['$n$', 'Built-in [$s$]',
      'Recursive [$s$]', 'Sequential [$s$]']
caption = '''Comparing execution time for
the different factorial implementations'''
matrix2latex(table, 'facT', caption=caption,
             columnLabels=cl, rowLabels=rl,
             format='$%.3f$')
```

TABLE 4. Comparing execution time for the different factorial implementations

$n$	Built-in [$s$]	Recursive [$s$]	Sequential [$s$]
0	0.211	0.423	0.598
1	0.264	0.611	1.654
2	0.326	1.055	1.673
3	0.385	1.585	1.888
4	0.428	1.870	2.133
5	0.491	2.343	2.317
6	0.552	2.748	2.495
7	0.658	3.338	2.716
8	0.733	5.291	2.866
9	0.802	6.493	3.166