This document describes the work flow and how we can utilize many nice DocOnce features when writing chapters for a future, potential book project.

1 Use of variables

Mako is the preprocessor that is always run prior to translating DocOnce documents into a specific format. It means that your DocOnce source is actually a computer program where you can use variables and functions.

Writing chapters that can both live their individual lives and be part of a book faces some challenges for which we have some nice solutions in the coming sections.

The easiest way to utilize Mako is to introduce variables in the text. For example, one can introduce a variable `COPYRIGHT` for the type of copyright desired for authors. Most Mako variables in this text are upper case, but any legal variable name in Python is also a legal name in Mako. In the DocOnce source file we can replace the variable by its content by writing `${COPYRIGHT}`:

```
AUTHOR: H. P. Langtangen ${COPYRIGHT} at Simula & UiO
```

The content of the variable can either be set at the command line as part of the `doconce format` command,

Terminal

```
Terminal> doconce format html mydoc COPYRIGHT='{copyright|CC BY}'
```

or hardcoded in the DocOnce file (as a standard Python variable) inside the `<%...%>` directives (before the first use of the variable):

```
<%
COPYRIGHT = '{copyright|This work is released under a BSD license.}'
%>
```

By having the copyright as a variable, we can use this variable for all authors to ensure consistency of copyrights, and we can easily compile different versions of the documents with different copyrights by just changing `COPYRIGHT=` on the command line.

Mako variables can be used in loops and if tests. DocOnce always defines a variable `FORMAT` holding the chosen output format. This variable is often used for emitting different text depending on the format, e.g.,

```
See
% if FORMAT in ('latex', 'pdflatex'):
Section ref{mysec}
% elif FORMAT == 'html':
ref{mysec}
% elif FORMAT == 'sphinx':
ref{mysec}
% else:
```

```
the previous section
% endif
for more information.
```

1.1 How to speak about "this chapter"

In a book you will often need the phrase "this chapter", but his is inappropriate if the chapter is a stand-alone document. Then you would rather say "this document". Similarly, "this book" must read "this document" in a stand-alone chapter. We have resolved this issue by introducing Mako variables `CHAPTER`, `BOOK`, and `APPENDIX` such that you write

```
In this ${BOOK}, the convention is to use boldface for vectors.
```

For this to work, you need to define `CHAPTER`, `BOOK`, and `APPENDIX` as variables on the command line as part of the `doconce format` command:

Terminal

```
Terminal> doconce format pdflatex ch2 --latex_code_style=pyg \
          CHAPTER=document BOOK=document APPENDIX=document
```

When the book is compiled, you do

Terminal

```
Terminal> doconce format pdflatex ch2 --latex_code_style=pyg \
          CHAPTER=chapter BOOK=chapter APPENDIX=appendix
```

The `make*.sh` files found in `doc/src/chapter/` and `doc/src/book` make proper definitions of `CHAPTER`, `BOOK`, and `APPENDIX`.

2 How to make several variants of the text

Sometimes you want to write some text slightly differently if the chapter is a stand-alone document compared to the case when it is part of a book. Mako if tests are ideal for this. Suppose you introduce a Mako variable `ALONE` that is true/defined if the chapter is a stand-alone document and false/undefined if part of a book. Then you can simply write

```
In this
% if ALONE:
rather small
% else:
large
% endif
${BOOK}
```

Running `doconce format` with the option `-DALONE` will turn `ALONE` to true and the output is typically

```
In this rather small document
```

while for a book we skip `-DALONE` as argument to `doconce format`, which makes `ALONE` undefined, and we get the output

```
In this large book
```

Mako variables can be defined/undefined (boolean variables) or be standard strings:

```
% if SOME_STRING_VARIABLE in ('value1', 'value2'):
some running text
% endif

...

% if not SOME_BOOLEAN_VARIABLE:
some other running text
% else:
yet more different text
% endif
```

With Mako variables, you can easily comment out large portions of text by testing on some variable you do not intend to define:

```
% if EXTRA:
This is
text that
will never
appear in the
output.
% endif
```

Also, it is straightforward to write more than one version of a chapter. For example, you may want to produce a version of a chapter that is tailored to a specific course, while you for general publishing on the Internet want a more general version, and maybe a third version when the chapter is included in a book for the international market. All this is easily done by if tests on appropriately defined Mako variables

```
% if COURSE == 'IT1713':
# Specific text for a course IT1713
...
% elif COURSE == 'IT1713b':
# Specific text for a the special IT1713b variant of the course
...
% elif COURSE == 'general':
# General text when the chapter is a stand-alone document
...
% elif COURSE == 'book1':
# Text when course is a part of a particular book
...
```

```
% elif COURSE == 'book2':
# Text when course is a part of another book
...
% endif
```

3 Mako's Python functions

The if tests above are fine to handle larger portions of text. What if you need to have four versions of just one word or very short text? A Mako function, defined as a standard Python function, is then more appropriate.

3.1 Basics of Mako functions

Here is a definition of a suitable Mako function, which must be defined inside `<%` and `%>` tags, using standard Python code:

```
<%
def chversion(text_IT1413, text_IT1713b, text_general,
              text_book1, text_book2):
    if COURSE == 'IT1713':
        return text_IT1413
    elif COURSE == 'IT1713b':
        text_IT1413b
    elif COURSE == 'general':
        return text_general
    elif COURSE == 'book1':
        return text_book1
    elif COURSE == 'book2':
        return text_book2
    else:
        return 'XXX WRONG value of COURSE: %s' % COURSE
%>
```

In the running text you can call `chversion` with five arguments, corresponding to the desired text in the five cases, and when `doconce format` is run, the value of `COURSE` determines which of the five cases that is used. Here is an example on DocOnce text with a function call to `chversion`:

```
It is extremely important to define the term *cure* accurately.
Here we mean ${chversion('handle', 'handle',
'resolve', 'treat', 'resolve')}.
```

You can easily use long multi-line strings as arguments, e.g.,

```
... ${chversion("""
Here comes
a multi-line
string""",
'short string',
'another short string',
"""4th
multi-line
string""",
'5th string')}
...
```

There are two types of Mako functions.

One type resembles Python functions, as demonstrated above. The other type employs a slightly different syntax and is exemplified in the file `doc/src/chapters/index_files.do.txt`. We refer to the Mako syntax documentation for more information.

3.2 How to automatically generate a DocOnce file with repetitive structure

To illustrate how Python and Mako can be used to efficiently generate repetitive structures with a minimum of manual work, we consider the following case. Suppose you have a DocOnce document made up of a number of sections, where the DocOnce source of each section resides in a subdirectory with name `issueX`, where `X` is an integer counter. You want to create a "master" DocOnce file that includes all the sections, e.g..

```
======= Issue 1 =======

# INCLUDE "issue1/issue.do.txt"

======= Issue 2 =======

# INCLUDE "issue2/issue.do.txt"

======= Issue 3 =======

# INCLUDE "issue3/issue.do.txt"
```

Maybe issues come and go, and so do the subdirectories, implying that one should automate the making of the above content of the master document.

Generating a set of sections via Mako is easy:

```
<%
sections = range(1, 8)
%>

% for i in sections:
======= Issue ${i} =======
% endfor
```

Unfortunately, we cannot write

```
% for i in sections:
======= Issue ${i} =======

# INCLUDE "issue${i}/issue.do.txt"
% endfor
```

because the `#include` statement is run by Preprocess *prior* to Mako's interpretation of the file. Instead, we can generate (parts of) the master file in a separate

Python script. This makes it also easier to check which subdirectories we have and set up the contents of sections based on the file structure:

```
import os, glob
outfile = open('master_section.do.txt', 'w')
subdirs = glob.glob('issue*')
# Run through all issue* subdirectory names in sorted sequence
for subdir in sorted(subdirs):
    if os.path.isdir(subdir):               # directory?
        if os.path.isfile('issue.do.txt'):  # file?
            # Extract number X from "issueX" name:
            no = subdir[5:]
            outfile.write("""
======= Issue %s =======

# INCLUDE "%s/issue.do.txt"
""" % (no, subdir))
outfile.close()
```

The master file can now just do an include of `master_sections.do.txt`. If the make script for compiling DocOnce to various formats first runs the script above, the `master_sections.do.txt` contents are up-to-date with the current file structure, and the contents automatically propagate to the master document.

There is one potential problem in the above example: the `issue.do.txt` files may include figures with local paths. For example, `issue5/issue.do.txt` contains

```
FIGURE: [fig/myfig, width=500 frac=0.8] My figure. label{my:fig}
```

When compiling the master document, no `fig/myfig.png` is found because the correct path, relative to the master document's directory, is `issue5/fig/myfig.png`. The same problem arises if there are source code inclusion statements like `@@@CODE src/myprog.f`. The master document would then need `@@@CODE issue5/src/myprog.f`. The best way out of these problems is

1. Let figure and source code directories have a unique name, say `fig5` and `src5` in this example.
2. Create links from the master document's directory to all the `fig*` and `src*` subdirectories.

Point 2 can be automated:

```
subdirs = glob.glob('issue*')
for subdir in sorted(subdirs):
    if os.path.isdir(subdir):
        no = subdir[5:]
        figdir = 'fig' + no
        srcdir = 'src' + no
        if not os.path.islink(figdir):
           path = os.path.join(subdir, figdir)
               os.symlink(figdir, path)
        if not os.path.islink(srcdir):
           path = os.path.join(subdir, srcdir)
               os.symlink(figdir, path)
```

This little case study shows the power of using scripts to assist the writing process.

3.3 How to treat multiple programming languages in the same text

With these ideas, it becomes straightforward to write a book that has its program examples in multiple languages. Introduce `CODE` as the name of the language and use if tests for larger portions of code and text, and Mako functions for shorter inline texts, to handle text that depends on the value of `CODE`. The author has successfully co-written such a book [**?**] for mathematical programming with either Python or Matlab - the version is set when running `doconce format`.

Here is an example of text, in the style of the mention book, where there are small differences depending on the programming language:

```
The following ${CODE} function ‘sampler‘ does the job
(see the file "${src('sampler')}":
"https://github.com/myuser/myproject/src/${src('sampler')}"):

${copyfile('sampler')}

Note that in ${CODE}, arrays start at index ${text2('0', '1')}.
Array slices like ${verb2('vec[2:8]', 'vec(2:7)')}
go from the first index (here ‘2‘) up to
${text2('*but not including* the upper limit (here ‘8‘)',
'(including) the upper limit (here ‘7‘)'}.
% if CODE == 'Python':
Also note that the file ‘sampler.py‘ is a module, meaning
that we can call all the file's functions from other programs,
including ‘sampler_vec‘.
% elif CODE == 'Matlab':
Also note that only the ‘sampler‘ function can be called
from other Matlab programs. If we want the alternative
implementation in function ‘sampler_vec‘ to be reused
by other programs, this function has to reside in a file
‘sampler_vec.py‘.
% endif
```

Here we have made use of a few Mako functions to easily choose between a Python or Matlab relevant text:

- `src` for picking a filename with the right extension (`.py` or `.m`)
- `copyfile` for constructing the right `@@@CODE` line for a Python or Matlab source code file
- `text2` for picking the first (Python) or second (Matlab) argument
- `verb2` for picking the first (Python) or second (Matlab) argument typeset in inline verbatim font

The exact Mako code appears below.

```
<%
def src(filestem, url=None, verb=True):
    """Return filstem plus .m or .py."""
    if CODE == "Python":
        filename = filestem + '.py'
    else:
        filename = filestem + '.m'
    if verb:
        filename = '`%s`' % filename
    if url is not None:
        # Make link to the file at github
        pass
    return filename

def copyfile(filestem, from_=None, to_=None):
    """Return @@@CODE line for copying a Python/Matlab file."""
    r = "@@@CODE "
    if CODE == "Python":
        r += "py-src/" + filestem + '.py'
    else:
        r += "m-src/" + filestem + '.m'
    if from_ is not None:
        r += ' fromto: ' + from_ + '@'
    if to_ is not None:
        r += to_
    return r

def verb2(py_expr, m_expr):
    """Return py_expr or m_expr in verbatim depending on CODE."""
    if CODE == "Python":
        expr = py_expr
    else:
        expr = m_expr
    expr = '`%s`' % expr
    return expr

def text2(py_expr, m_expr):
    """Return py_expr or m_expr depending on CODE."""
    if CODE == "Python":
        expr = py_expr
    else:
        expr = m_expr
    return expr

%>
```

Compiling the document with

Terminal

```
Terminal> doconce format plain mydoc CODE=Python \
          --latex_code_style=pyg
```

results in the output

```
The following Python function \texttt{sampler} does the job
(see the file
\href{{https://github.com/myuser/myproject/src/`sampler.py`}}{\nolinkurl{sampler.py}}):

\begin{minted}[fontsize=\fontsize{9pt}{9pt},linenos=false,
```

```
baselinestretch=1.0,fontfamily=tt,xleftmargin=2mm]{python}
"""Sampler module."""

def sampler(...):
    ...
\end{minted}

Note that in Python, arrays start at index 0.
Array slices like \texttt{vec[2:8]}
go from the first index (here \texttt{2}) up to
\emph{but not including} the upper limit (here \texttt{8}).
Also note that the file \texttt{sampler.py} is a module, meaning
that we can call all the file's functions from other programs,
including \Verb!sampler_vec!.
```

Switching to `CODE=Matlab` gives

```
The following Matlab function \texttt{sampler} does the job
(see the file
\href{{https://github.com/myuser/myproject/src/‘sampler.m‘}}{\nolinkurl{sampler.m}}):

\begin{minted}[fontsize=\fontsize{9pt}{9pt},linenos=false,
baselinestretch=1.0,fontfamily=tt,xleftmargin=2mm]{matlab}
% Sampler code

function samples = sampler(...):
    ...
\end{minted}

Note that in Matlab, arrays start at index 1.
Array slices like \texttt{vec(2:7)}
go from the first index (here \texttt{2}) up to
(including) the upper limit (here \texttt{7}.
Also note that only the \texttt{sampler} function can be called
from other Matlab programs. If we want the alternative
implementation in function \Verb!sampler_vec! to be reused
by other programs, this function has to reside in a file
\Verb!sampler_vec.py!.
```

Another example. The manual contains a useful example on how to use Mako to implement the nomenclature functionality in the LaTeX package `nomencl`.

Index