Doconce Description

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Mar 10, 2014

1 What Is Doconce?

Doconce is a very simple and minimally tagged markup language that looks like ordinary ASCII text, much like what you would use in an email, but the text can be transformed to numerous other formats, including HTML, Sphinx, LaTeX, PDF, reStructuredText (reST), Markdown, MediaWiki, Google wiki, Creole wiki, blogger.com, wordpress.com, Epytext, and also plain (untagged) text for email. From reST or Markdown you can go to XML, OpenOffice, MS Word, HTML, LaTeX, PDF, DocBook, GNU Texinfo, and more.

Doconce supports a working strategy of never duplicating information. Text is written in a single place and then transformed to a number of different destinations of diverse type: scientific reports, software manuals, books, thesis, software source code, wikis, blog posts, emails, etc. The slogan is: "Write once, include anywhere".

Here are some Doconce features:

- Doconce addresses small and large documents containing *text with much computer source code and LaTeX mathematics*, where the output is desired in different formats such as LaTeX, PDFLaTeX, Sphinx, HTML, MediaWiki, blogger.com, and wordpress.com. A piece of Doconce text can enter (e.g.) a classical science book, an ebook, a web document, and a blog post.
- Doconce targets in particular large book projects where many different pieces of text and software can be assembled and published in different formats for different devices.
- Doconce enables authors who write for many times of media (blog posts, wikis, LaTeX manuscripts, Sphinx, HTML) to use a common source language such that lots of different pieces can easily be brought together later to form a coherent (big) document.

- Doconce has good support for copying computer code directly from the source code files via regular expressions for the start and end lines.
- Doconce first runs two preprocessors (Preprocess and Mako), which allow programming constructs (includes, if-tests, function calls) as part of the text. This feature makes it easy to write *one text* with different flavors: long vs short text, Python vs Matlab code examples, experimental vs mature content.
- Doconce can be converted to plain *untagged* text, often desirable for email and computer code documentation.
- Doconce markup does include tags, so the format is more tagged than Markdown, but less than reST, and very much less than LaTeX and HTML.
- Compared to the related tools Sphinx and Markdown, Doconce allows more types of equations (especially systems of equations with references), has more flexible inclusion of source code, integrates preprocessors, has special support for exercises, and produces cleaner LaTeX and HTML output.

History. Doconce was developed in 2006 at a time when most popular markup languages used quite some tagging. Later, almost untagged markup languages like Markdown and the Pandoc translator became popular. Doconce is not a replacement of Pandoc, which is a considerably more sophisticated project. Moreover, Doconce was developed mainly to fulfill the needs for a flexible source code base for books with much mathematics and computer code.

Disclaimer. Doconce is a simple tool, largely based on interpreting and handling text through regular expressions. The possibility for tweaking the layout is obviously limited since the text can go to all sorts of sophisticated markup languages. Moreover, because of limitations of regular expressions, some formatting of Doconce syntax may face problems when transformed to HTML, LaTeX, Sphinx, and similar formats.

2 Installation of Doconce and its Dependencies

Below, we explain the manual installation of all software that may be needed when working with Doconce documents. The impatient way to install what is needed is to run the `install_doconce.sh` (or `install_doconce.py`) script.

2.1 Doconce

Doconce itself is pure Python code hosted at https://github.com/hplgit/doconce. Its installation from the Git source follows the standard procedure:

```
# Doconce
git clone git@github.com:hplgit/doconce.git
cd doconce
sudo python setup.py install
cd ..
```

Since Doconce is frequently updated, it is recommended to use the above procedure and whenever a problem occurs, make sure to update to the most recent version:

```
cd doconce
git pull origin master
sudo python setup.py install
```

2.2 Dependencies

Producing HTML documents, plain text, pandoc-extended Markdown, and wikis can be done without installing any other software. However, if you want other formats as output (LaTeX, Sphinx, reStructuredText) and assisting utilities such as preprocesors, spellcheck, file differences, bibliographies, and so on, the software below must be installed.

To install all needed packages on a GNU/Linux Debian system, such as Ubuntu, you can jump to the script in Section 2.3.

Preprocessors. If you make use of the Preprocess preprocessor, this program must be installed:

```
svn checkout http://preprocess.googlecode.com/svn/trunk/ preprocess
cd preprocess
cd doconce
sudo python setup.py install
cd ..
```

A much more advanced alternative to Preprocess is Mako. Its installation is most conveniently done by `pip`,

```
pip install Mako
```

This command requires `pip` to be installed. On Debian Linux systems, such as Ubuntu, the installation is simply done by

```
sudo apt-get install python-pip
```

Alternatively, one can install from the `pip` source code.

Mako can also be installed directly from source: download the tarball, pack it out, go to the directory and run the usual `sudo python setup.py install`.

Image file handling. Different output formats require different formats of image files. For example, PostScript or Encapuslated PostScript is required for `latex` output, while HTML needs JPEG, GIF, or PNG formats. Doconce calls up programs from the ImageMagick suite for converting image files to a proper format if needed. The ImageMagick suite can be installed on all major platforms. On Debian Linux (including Ubuntu) systems one can simply write

```
sudo apt-get install imagemagick
```

The convenience program `doconce combine_images`, for combining several images into one, will use `montage` and `convert` from ImageMagick and the `pdftk`, `pdfnup`, and `pdfcrop` programs from the `texlive-extra-utils` Debian package. The latter gets installed by

```
sudo apt-get install texlive-extra-utils
```

Automatic image conversion from EPS to PDF calls up `epstopdf`, which can be installed by

```
sudo apt-get install texlive-font-utils
```

Spellcheck. The utility `doconce spellcheck` applies the `ispell` program for spellcheck. On Debian (including Ubuntu) it is installed by

```
sudo apt-get install ispell
```

Bibliography. The Python package Publish is needed if you use a bibliography in your document. On the website, click on *Clone*, copy the command and run it:

```
hg clone https://bitbucket.org/logg/publish
```

Thereafter go to the `publish` directory and run the `setup.py` script for installing Publish:

```
cd publish
sudo python setup.py
```

Ptex2tex for LaTeX Output. To make LaTeX documents with very flexible choice of typesetting of verbatim code blocks you need ptex2tex, which is installed by

```
svn checkout http://ptex2tex.googlecode.com/svn/trunk/ ptex2tex
cd ptex2tex
sudo python setup.py install
```

It may happen that you need additional style files, you can run a script, `cp2texmf.sh`:

```
cd latex
sh cp2texmf.sh  # copy stylefiles to ~/texmf directory
cd ../..
```

This script copies some special stylefiles that that `ptex2tex` potentially makes use of. Some more standard stylefiles are also needed. These are installed by

```
sudo apt-get install texlive
```

on Debian Linux (including Ubuntu) systems. TeXShop on Mac comes with the necessary stylefiles (if not, they can be found by googling and installed manually in the `~/texmf/tex/latex/misc` directory).

Note that the `doconce ptex2tex` command, which needs no installation beyond Doconce itself, can be used as a simpler alternative to the `ptex2tex` program.

The *minted* LaTeX style is offered by `ptex2tex` and `doconce ptext2tex` and popular among many users. This style requires the package Pygments to be installed. On Debian Linux,

```
sudo apt-get install python-pygments
```

Alternatively, the package can be installed manually:

```
hg clone ssh://hg@bitbucket.org/birkenfeld/pygments-main pygments
cd pygments
sudo python setup.py install
```

One can also do the simple

```
pip install sphinx
```

which also installs pygments.

If you use the minted style together with `ptex2tex`, you have to enable it by the `-DMINTED` command-line argument to `ptex2tex`. This is not necessary if you run the alternative `doconce ptex2tex` program.

All use of the minted style requires the `-shell-escape` command-line argument when running LaTeX, i.e., `latex -shell-escape` or `pdflatex -shell-escape`.

Inline comments apply the `todonotes` LaTeX package if the `ptex2tex` or `doconce ptex2tex` command is run with `-DTODONOTES`. The `todonotes` package requires several other packages: `xcolor`, `ifthen`, `xkeyval`, `tikz`, `calc`, `graphicx`, and `setspace`. The relevant Debian packages for installing all this are listed below.

LaTeX packages. Many LaTeX packages are potentially needed (depending on various preprocessor variables given to `ptex2tex` or `doconce ptex2tex`. The standard packages always included are `relsize`, `epsfig`, `makeidx`, `setspace`, `color`, `amsmath`, `amsfonts`, `xcolor`, `bm`, `microtype`, `titlesec`, and `hyperref`. The `ptex2tex` package (from ptex2tex) is also included, but removed again if `doconce ptex2tex` is run instead of the `ptex2tex` program, meaning that if you do not use `ptex2tex`, you do not need `ptex2tex.sty`. Optional packages that might be included are `minted`, `fontspec`, `xunicode`, `inputenc`, `helvet`, `mathpazo`, `wrapfig`, `calc`, `ifthen`, `xkeyval`, `tikz`, `graphicx`, `setspace`, `shadow`, `disable`, `todonotes`, `lineno`, `xr`, `framed`, `mdframe`, `movie15`, `a4paper`, and `a6paper`.

Relevant Debian packages that gives you all of these LaTeX packages are

```
texlive
texlive-extra-utils
texlive-latex-extra
texlive-font-utils
```

On old Ubuntu 12.04 one has to do `sudo add-apt-repository ppa:texlive-backports/ppa` and `sudo apt-get update` first, or alternatively install these as well:

```
texlive-math-extra
texlive-bibtex-extra
texlive-xetex
texlive-humanities
texlive-pictures
```

Alternatively, one may pull in `texlive-full` to get all available style files.

If you want to use the *anslistings* code environment with `ptex2tex` (`.ptex2tex.cfg` styles `Python_ANS`, `Python_ANSt`, `Cpp_ANS`, etc.) or `doconce ptex2tex` (`envir=ans` or `envir=ans:nt`), you need the `anslistings.sty` file. It can be obtained from the ptex2tex source. It should get installed by the `cp2texmf.sh` script executed above.

reStructuredText (reST) Output. The `rst` output from Doconce allows further transformation to LaTeX, HTML, XML, OpenOffice, and so on, through the docutils package. The installation of the most recent version can be done by

```
svn checkout \
  http://docutils.svn.sourceforge.net/svnroot/docutils/trunk/docutils
cd docutils
sudo python setup.py install
cd ..
```

The command

```
pip install sphinx
```

installs Docutils along with Sphinx and Pygments.

To use the OpenOffice suite you will typically on Debian systems install

```
sudo apt-get install unovonv libreoffice libreoffice-dmaths
```

There is a possibility to create PDF files from reST documents using ReportLab instead of LaTeX. The enabling software is rst2pdf. Either download the tarball or clone the svn repository, go to the `rst2pdf` directory and run the usual `sudo python setup.py install`.

Sphinx Output. Output to `sphinx` requires of course the Sphinx software, installed by

```
hg clone https://bitbucket.org/birkenfeld/sphinx
cd sphinx
sudo python setup.py install
cd ..
```

An alternative is

```
pip install sphinx
```

Doconce comes with many Sphinx themes that are not part of the standard Sphinx source distribution. Some of these themes require additional Python/Sphinx modules to be installed:

- cloud and redcloud: https://bitbucket.org/ecollins/cloud_sptheme
- bootstrap: https://github.com/ryan-roemer/sphinx-bootstrap-theme
- solarized: https://bitbucket.org/miiton/sphinxjp.themes.solarized
- impressjs: https://github.com/shkumagai/sphinxjp.themes.impressjs
- sagecellserver: https://github.com/kriskda/sphinx-sagecell

These must be downloaded or cloned, and `setup.py` must be run as shown above.

Markdown and Pandoc Output. The Doconce format `pandoc` outputs the document in the Pandoc extended Markdown format, which via the `pandoc` program can be translated to a range of other formats. Installation of Pandoc, written in Haskell, is most easily done by

```
sudo apt-get install pandoc
```

on Debian (Ubuntu) systems.

Epydoc Output. When the output format is `epydoc` one needs that program too, installed by

```
svn co https://epydoc.svn.sourceforge.net/svnroot/epydoc/trunk/epydoc epydoc
cd epydoc
sudo make install
cd ..
```

Remark. Several of the packages above installed from source code are also available in Debian-based system through the `apt-get install` command. However, we recommend installation directly from the version control system repository as there might be important updates and bug fixes. For `svn` directories, go to the directory, run `svn update`, and then `sudo python setup.py install`. For Mercurial (`hg`) directories, go to the directory, run `hg pull; hg update`, and then `sudo python setup.py install`.

Analyzing file differences. The `doconce diff file1 file prog` command for illustrating differences between two files `file1` and `file2` using the program `prog` requires `prog` to be installed. By default, `prog` is `difflib` which comes with Python and is always present if you have Doconce installed. Another choice, `diff`, should be available on all Unix/Linux systems. Other choices, their URL, and their `sudo apt-get install` command on Debian (Ubuntu) systems appear in the table below.

Program	URL	Debian/Ubuntu install
`pdiff`	a2ps wdiff	`sudo apt-get install a2ps wdiff texlive-latex-extra texli`
`latexdiff`	latexdiff	`sudo apt-get install latexdiff`
`kdiff3`	kdiff3	`sudo apt-get install kdiff3`
`diffuse`	diffuse	`sudo apt-get install diffuse`
`xxdiff`	xxdiff	`sudo apt-get install xxdiff`
`meld`	meld	`sudo apt-get install meld`
`tkdiff.tcl`	tkdiff	not in Debian

2.3 Quick Debian/Ubuntu Install

On Debian (including Ubuntu) systems, it is straightforward to install the long series of Doconce dependencies:

```
# Version control systems
sudo apt-get install -y mercurial git subversion

# Python
sudo apt-get install -y idle ipython python-pip python-pdftools texinfo

# LaTeX
sudo apt-get install -y texlive texlive-extra-utils texlive-latex-extra texlive-math-extra texli
# or sudo apt-get install -y texlive-full  # get everything
sudo apt-get install -y latexdiff auctex

# Image and movie tools
sudo apt-get install -y imagemagick netpbm mjpegtools pdftk giftrans gv libav-tools libavcodec-e

# Misc
sudo apt-get install -y ispell pandoc libreoffice unoconv libreoffice-dmaths curl a2ps wdiff mel

# More Python software
sudo pip install sphinx  # install pygments and docutils too
sudo pip install mako
sudo pip install -e svn+http://preprocess.googlecode.com/svn/trunk#egg=preprocess
sudo pip install -e hg+https://bitbucket.org/logg/publish#egg=publish

sudo pip install -e hg+https://bitbucket.org/ecollins/cloud_sptheme#egg=cloud_sptheme
sudo pip install -e git+https://github.com/ryan-roemer/sphinx-bootstrap-theme#egg=sphinx-bootstr
sudo pip install -e hg+https://bitbucket.org/miiton/sphinxjp.themes.solarized#egg=sphinxjp.theme
sudo pip install -e git+https://github.com/shkumagai/sphinxjp.themes.impressjs#egg=sphinxjp.them
sudo pip install -e git+https://github.com/kriskda/sphinx-sagecell#egg=sphinx-sagecell
sudo pip install -e svn+https://epydoc.svn.sourceforge.net/svnroot/epydoc/trunk/epydoc#egg=epydo

# Doconce itself
rm -rf srclib   # put downloaded software in srclib
mkdir srclib
```

```
cd srclib
git clone git@github.com:hplgit/doconce.git
cd doconce
sudo python setup.py install -y
cd ../..

# Ptex2tex
cd srclib
svn checkout http://ptex2tex.googlecode.com/svn/trunk/ ptex2tex
cd ptex2tex
sudo python setup.py install -y
cd latex
sh cp2texmf.sh  # copy stylefiles to ~/texmf directory
cd ../../..
```

2.4 Demos

The current text is generated from a Doconce format stored in the directory

```
doc/manual/manual.do.txt
```

file in the Doconce source code tree. Here you can run a `make.sh` script to generate a lot of different formats: HTML, LaTeX, plain text, etc., stored in the subdirectory `demo`.

Another demo is found in

```
doc/tutorial/tutorial.do.txt
```

In the `tutorial` directory there is also a `make.sh` file producing a lot of formats in the subdirectory `demo`.

3 From Doconce to Other Formats

Transformation of a Doconce document `mydoc.do.txt` to various other formats apply the script `doconce format`:

```
Terminal> doconce format format mydoc.do.txt
```

or just

```
Terminal> doconce format format mydoc
```

3.1 Generating a makefile

Producing HTML, Sphinx, and in particular LaTeX documents from Doconce sources requires a few commands. Often you want to produce several different formats. The relevant commands should then be placed in a script that acts as a "makefile".

The `doconce makefile` can be used to automatically generate such a makefile, more precisely a Python script `make.py`, which carries out the commands explained below. If our Doconce source is in `main_myproj.do.txt`, we run

```
doconce makefile main_myproj html pdflatex sphinx
```

to produce the necessary output for generating HTML, PDFLaTeX, and Sphinx. Usually, you need to edit `make.py` to really fit your needs. Some examples lines are inserted as comments to show various options that can be added to the basic commands. A handy feature of the generated `make.py` script is that it inserts checks for successful runs of the many `doconce` commands, and if something goes wrong, the script aborts.

3.2 Preprocessing

The `preprocess` and `mako` programs are used to preprocess the file. The Doconce program detects whether `preprocess` and/or `mako` statements are present and runs the corresponding programs, first `preprocess` and then `mako`.

Options to `preprocess` and/or `mako` can be added after the filename with the syntax `-DMYVAR` or `MYVAR=True` for defining a variable `MYVAR`, or `-DMYVAR=val` or `MYVAR=val` for giving `MYVAR` the value `val`. For example,

```
Terminal> doconce format sphinx mydoc -Dextra_sections -DVAR1=5
Terminal> doconce format sphinx mydoc extra_sections=True VAR1=5
```

The variable `FORMAT` is always defined as the current format when running `preprocess` or `mako`. That is, in the above examples, `FORMAT` is defined as `sphinx`. Inside the Doconce document one can then perform format specific actions through tests like `#if FORMAT == "sphinx"` (for `preprocess`) or `% if FORMAT == "sphinx":` (for `mako`).

The result of a Doconce file `mydoc.do.txt` after running `preprocess` is available in a file `tmp_preprocess__mydoc.do.txt`. Similarly, the result of running `mako` is available in `tmp_mako__mydoc.do.txt`.

3.3 Removal of inline comments

The command-line arguments `--no_preprocess` and `--no_mako` turn off running `preprocess` and `mako`, respectively.

Inline comments in the text are removed from the output by

```
Terminal> doconce format latex mydoc --skip_inline_comments
```

One can also remove all such comments from the original Doconce file by running:

```
Terminal> doconce remove_inline_comments mydoc
```

This action is convenient when a Doconce document reaches its final form and comments by different authors should be removed.

3.4 Notes

Doconce does not have a tag for longer notes, because implementation of a "notes feature" is so easy using the `preprocess` or `mako` programs. Just introduce some variable, say `NOTES`, that you define through `-DNOTES` (or not) when running `doconce format ...`. Inside the document you place your notes between `# #ifdef NOTES` and `# #endif` preprocess tags. Alternatively you use `% if NOTES:` and `% endif` that `mako` will recognize. In the same way you may encapsulate unfinished material, extra material to be removed for readers but still nice to archive as part of the document for future revisions.

3.5 Demo of different formats

A simple scientific report is available in a lot of different formats. How to create the different formats is explained in more depth in the coming sections.

3.6 HTML

Basics. Making an HTML version of a Doconce file `mydoc.do.txt` is performed by

```
Terminal> doconce format html mydoc
```

The resulting file `mydoc.html` can be loaded into any web browser for viewing.

Typesetting of Code. If the Pygments package (including the `pygmentize` program) is installed, code blocks are typeset with aid of this package. The command-line argument `--no_pygments_html` turns off the use of Pygments and makes code blocks appear with plain (`pre`) HTML tags. The option `--pygments_html_linenos` turns on line numbers in Pygments-formatted code blocks. A specific Pygments style is set by `--pygments_html_style=style`, where `style` can be `default`, `emacs`, `perldoc`, and other valid names for Pygments styles.

Handling of Movies. MP4, WebM, and Ogg movies are typeset with the HTML5 `video` tag and the HTML code tries to load as many versions among MP4, WebM, and Ogg as exist (and the files are loaded in the mentioned order). If just the specified file is to be loaded, use the `--no_mp4_webm_ogg_alternatives` command-line option. Other movie formats, e.g., `.flv`, `.mpeg` and `.avi`, are embedded via the older `embed` tag.

HTML Styles. The HTML style can be defined either in the header of the HTML file, using a named built-in style; in an external CSS file; or in a template file.

An external CSS file `filename` used by setting the command-line argument `--css=filename`. There available built-in styles are specified as `--html_style=name`, where `name` can be

- `solarized`: the famous solarized style (yellowish),
- `blueish`: a simple style with blue headings (default),
- `blueish2`: a variant of *bluish*,
- `bloodish`: as `bluish`, but dark read as color.

Using `--css=filename` where `filename` is a non-existing file makes Doconce write the built-in style to that file. Otherwise the HTML links to the CSS stylesheet in `filename`. Several stylesheets can be specified: `--ccs=file1.css,file2.css,file3.css`.

HTML templates. Templates are HTML files with "slots" `%(main)s` for the main body of text, `%(title)s` for the title, and `%(date)s` for the date. Doconce comes with a few templates. The usage of templates is described in a "separate document": "". That document describes how you your Doconce-generated HTML file can have any specified layout.

The HTML file can be embedded in a template with your own tailored design, see a tutorial on this topic. The template file must contain valid HTML code and can have three "slots": `%(title)s` for a title, `%(date)s` for a date, and `%(main)s` for the main body of text. The latter is the Doconce document translated to HTML. The title becomes the first heading in the Doconce document, or the title (but a title is not recommended when using templates). The date is extracted from the `DATE:` line. With the template feature one can easily embed the text in the look and feel of a website. Doconce comes with two templates in `bundled/html_styles`. Just copy the directory containing the template and the CSS and JavaScript files to your document directory, edit the template as needed (also check that paths to the `css` and `js` subdirectories are correct - according to how you store the template files), and run

```
Terminal> doconce format html mydoc --html_template=mytemplate.html
```

The template in `style_vagrant` also needs an extra option `--html_style=vagrant`. With this style, one has nice navigation buttons that are used if the document contains `!split` commands for splitting it into many pages.

The HTML File Collection. There are usually a range of files needed for an HTML document arising from a Doconce source. The needed files are listed in `.basename_html_file_collection`, where `basename` is the filestem of the Doconce file (i.e., the Doconce source is in `basename.do.txt`).

Filenames. An HTML version of a Doconce document is often made in different styles, calling for a need to rename the HTML output file. This is conveniently done by the `--html_output=basename` option, where `basename` is the filestem of the associated HTML files. The `.basename_html_file_collection` file lists all the needed files for the HTML document. Here is an example on making three versions of the HTML document: `mydoc_bloodish.html`, `mydoc_solarized`, and `mydoc_vagrant`.

```
Terminal> doconce format html mydoc --html_style=bloodish \
          --html_output=mydoc_bloodish
Terminal> doconce split_html mydoc_bloodish.html
Terminal> doconce format html mydoc --html_style=solarized \
          --html_output=mydoc_solarized \
          --pygments_html_style=perldoc --html_admon=apricot
Terminal> doconce format html mydoc --html_style=vagrant \
          --html_output=mydoc_vagrant --pygments_html_style=default \
          --html_template=templates/my_adapted_vagrant_template.html
Terminal> doconce split_html mydoc_vagrant.html
```

3.7 Blog Posts

Doconce can be used for writing blog posts provided the blog site accepts raw HTML code. Google's Blogger service (`blogger.com` or `blogname.blogspot.com`) is particularly well suited since it also allows extensive LaTeX mathematics via MathJax.

1. Write the text of the blog post as a Doconce document without any title, author, and date.
2. Generate HTML as described above.
3. Copy the text and paste it into the text area in the blog post (just delete the HTML code that initially pops up in the text area). Make sure the input format is HTML.

See a simple blog example and a scientific report for demonstrations of blog posts at `blogspot.no`.

Warning.

In the readers' comments after the blog post one cannot paste raw HTML code with MathJax scripts so there is no support for mathematics in the discussion forum.

Notice.

Figure files must be uploaded to some web site and the local filenames name must be replaced by the relevant URL. This is usually done by using the `--figure_prefix=http://project.github.io/...` option to give some URL as prefix to all figure names (a similar `--movie_prefix=` option exists as well).

Changing figure names in a blog post can also be done "manually" by some editing code in the script that compiles the Doconce document to HTML format:

```
cp mydoc.do.txt mydoc2.do.txt
url="https//raw.github.com/someuser/someuser.github.com"
dir="master/project/dir1/dir2"
for figname in fig1 fig2 fig3; do
  doconce replace "[$figname," "[$site/$dir/$figname.png," \
          mydoc2.do.txt
done
doconce format html mydoc2
# Paste mydoc2.html into a new blog post page
```

Blog posts at Google can also be published automatically through email. A Python program can send the contents of the HTML file to the blog site's email address using the packages `smtplib` and `email`.

WordPress (`wordpress.com`) allows raw HTML code in blogs, but has very limited LaTeX support, basically only formulas. The `--wordpress` option to `doconce` modifies the HTML code such that all equations are typeset in a way that is acceptable to WordPress. Look at a simple doconce example and a scientific report to see blog posts with mathematics and code on WordPress.

Speaking of WordPress, the related project `http://pressbooks.com` can take raw HTML code (from Doconce, for instance) and produce very nice-looking books. There is no support for mathematics in the text, though.

3.8 Pandoc and Markdown

Output in Pandoc's extended Markdown format results from

```
Terminal> doconce format pandoc mydoc
```

The name of the output file is `mydoc.mkd`. From this format one can go to numerous other formats:

```
Terminal> pandoc -R -t mediawiki -o mydoc.mwk --toc mydoc.mkd
```

Pandoc supports `latex`, `html`, `odt` (OpenOffice), `docx` (Microsoft Word), `rtf`, `texinfo`, to mention some. The `-R` option makes Pandoc pass raw HTML or LaTeX to the output format instead of ignoring it, while the `--toc` option generates a table of contents. See the Pandoc documentation for the many features of the `pandoc` program. The HTML output from `pandoc` needs adjustments to provide full support for MathJax LaTeX mathematics, and for this purpose one should use `doconce md2html`:

```
Terminal> doconce format pandoc mydoc
Terminal> doconce m2html mydoc
```

The result `mydoc.html` can be viewed in a browser.

Pandoc is useful to go from LaTeX mathematics to, e.g., HTML or MS Word. There are two ways (experiment to find the best one for your document): `doconce format pandoc` and then translating using `doconce md2latex` (which runs `pandoc`), or `doconce format latex`, and then going from LaTeX to the desired format using `pandoc`. Here is an example on the latter strategy:

```
Terminal> doconce format latex mydoc
Terminal> doconce ptex2tex mydoc
Terminal> doconce replace '\Verb!' '\verb!' mydoc.tex
Terminal> pandoc -f latex -t docx -o mydoc.docx mydoc.tex
```

When we go through `pandoc`, only single equations, `align`, or `align*` environments are well understood for output to HTML.

Note that Doconce applies the `Verb` macro from the `fancyvrb` package while `pandoc` only supports the standard `verb` construction for inline verbatim text. Moreover, quite some additional `doconce replace` and `doconce subst` edits might be needed on the `.mkd` or `.tex` files to successfully have mathematics that is well translated to MS Word. Also when going to reStructuredText using Pandoc, it can be advantageous to go via LaTeX.

Here is an example where we take a Doconce snippet (without title, author, and date), maybe with some unnumbered equations, and quickly generate HTML with mathematics displayed my MathJax:

```
Terminal> doconce format pandoc mydoc
Terminal> pandoc -t html -o mydoc.html -s --mathjax mydoc.mkd
```

The `-s` option adds a proper header and footer to the `mydoc.html` file. This recipe is a quick way of makeing HTML notes with (some) mathematics.

GitHub-flavored Markdown. Adding the command-line option `github-md` turns on the GutHub-flavored Markdown dialect, which is used for the issue tracker on GitHub. A special feature is the support of task lists: unnumbered lists with `[x]` (task done) or `[ ]` (task not done). (Tables get typeset directly as HTML and the syntax for code highlighting is different from Pandoc extended Markdown.) Below is a typical response in a GitHub issue tracker where one first quotes the issue and then provides an answer:

```
!bquote
===== Problems with a function =====

There is a problem with the `f(x)` function

!bc pycod
def f(x):
    return 1 + x
!ec
This function should be quadratic.
!equote

OK, this is fixed:

!bc pycod
def f(x, a=1, b=1, c=1):
    return a*x**2 + b*x + c
!ec

===== Updated task list =====

   * [x] Offer an `f(x)` function
   * [ ] Extension to cubic functions
   * [x] Allowing general coefficient in the quadratic function
```

```
=== Remaining functionality ===

|-------------------------------------------------------------|
| function | purpose                          | state |
|----l-----------l--------------------------------l-----------|
|  `g(x)`  | Compute the Gaussian function. | Formula ready. |
|  `h(x)`  | Heaviside function.            | Formula ready. |
|  `I(x)`  | Indicator function.            | Nothing done yet. |
|-------------------------------------------------------------|
```

Say this text is stored in a file `mycomments.do.txt`. Running

```
Terminal> doconce format pandoc mycomments --github_md
```

produces the Markdown file `mycomments.md`, which can be pasted into the Write field of the GitHub issue tracker. Turning on Preview shows the typesetting of the quote, compute code, inline verbatim, headings, the task list, and the table.

3.9 LaTeX

Notice.

XeLaTeX and PDFLaTeX are used very much in the same way as standard LaTeX. The minor differences are described in separate sections of the documentation of the Doconce to LaTeX translation.

Making a LaTeX file `mydoc.tex` from `mydoc.do.txt` is done in two steps: 1) compile the Doconce source to the `ptex2tex` format, and 2) compile the `ptex2tex` format to standard LaTeX. The `ptex2tex` format can be viewed as an extended LaTeX. For Doconce users, the `ptex2tex` format essentially means that the file consists of

1. `if-else` statements for the `preprocess` processor such that LaTeX constructions can be activated or deactivated, and
2. all code environments can be typeset according to a `.ptex2tex.cfg` configuration file.

Point 2 is only of interest if you aim to use a special computer code formatting that requires you to use a configuration file and the `ptex2tex` program.

The reason for generating `ptex2tex` and not standard LaTeX directly from Doconce was that the `ptex2tex` format shows a range of possible LaTeX constructions for controlling the layout. It can be instructive for LaTeX users to look at this code before choosing specific parts for some desired layout. Experts may also want to edit this code (which should be automated by a script such that the edits can be repeated when the Doconce source is modified, see Step 2b below). (Direct control of the LaTeX layout in the `doconce format` program would not spit out alternative LaTeX constructs as is now done through the `ptex2tex` step.)

Going from `ptex2tex` format to standard LaTeX format is enabled by either the `ptex2tex` program or Doconce's (simplified) version of it: `doconce ptex2tex`. Details are given below.

Information on typesetting of inline verbatim.

The `ptex2tex` and the `doconce ptex2tex` programs take inline verbatim code, typeset with backticks in Doconce, and translate this to

```
\Verb!text!
```

Thereafter, if `text` does not contain illegal characters for the `\texttt` command, the latter is used instead since then LaTeX can insert linebreaks in the inline verbatim text and hence avoid overfull hboxes.

Step 1. Filter the doconce text to the `ptex2tex` "pre-LaTeX form" `mydoc.p.tex`:

```
Terminal> doconce format latex mydoc
```

LaTeX-specific commands ("newcommands") in math formulas and similar can be placed in files `newcommands.tex`, `newcommands_keep.tex`, or `newcommands_replace.tex` (see Section 4.20). If these files are present, they are included in the LaTeX document so that your commands are defined.

An option `--device=paper` makes some adjustments for documents aimed at being printed. For example, links to web resources are associated with a footnote listing the complete web address (URL). (Very long URLs in footnotes can be shortened using services such as `http://goo.gl/`, `http://tinyurl.com/`, and `https://bitly.com/`.) The default, `--device=screen`, creates a PDF file for reading on a screen where links are just clickable.

There are many additional options (run `doconce format --help` and look for options starting with `--latex` to get a more verbose description):

- `--latex_font=helvetica,palatino`
- `--latex_papersize=a4,a6`
- `--latex_bibstyle=plain`
- `--latex_title_layout=titlepage,std,beamer,doconce_heading`
- `--latex_style=std,Springer_lncse,Springer_llncs,Springer_T2,...`
- `--latex_list_of_exercises=loe,toc,none`
- `--latex_fancy_header`
- `--latex_section_headings=std,blue,strongblue,gray,gray-wide`
- `--latex_colored_table_rows=blue,gray,no`

- `--latex_todonotes`
- `--latex_double_spacing`
- `--latex_preamble=filename`
- `--latex_admon=mdfbox,graybox2,grayicon,yellowicon,paragraph,colors1,colors2`
- `--latex_admon_color=0.34,0.02,0.8`
- `--latex_admon_envir_map=2`
- `--latex_exercise_numbering=absolute,chapter`

Step 2. Run `ptex2tex` (if you have installed the Python ptex2tex package) to make a standard LaTeX file,

```
Terminal> ptex2tex mydoc
```

In case you do not have `ptex2tex`, you may run the (simplified) version that comes with Doconce:

```
Terminal> doconce ptex2tex mydoc
```

The `.p.tex` file contains a lot of preprocessor variables (like C macros) that can be used to steer certain properties of the LaTeX document. For example, to turn on the Helvetica font instead of the standard Computer Modern font, run

```
Terminal> ptex2tex -DHELVETICA mydoc
Terminal> doconce ptex2tex mydoc -DHELVETICA  # alternative
```

Preprocessor variables to be defined or undefined are

- `XELATEX` for processing by `xelatex`
- `MOVIE` for specifying how movies are handled: the value `media9` implies the `media9` package and the `\includemedia` command for `.flv` and `.mp4` movies, or `movie15` and the `\includemovie` command for `.mpeg` and `.avi` movies; `multimedia` for Beamer-style `\movie` command, or `href` for the plain `\href{run:file}` command. The latter is default.
- `MOVIE_CONTROLS` adds buttons for starting/stopping movies if the `media9` package is used.
- `PREAMBLE` to turn the LaTeX preamble on or off (i.e., complete document or document to be included elsewhere - and note that the preamble is only included if the document has a title, author, and date)
- `MINTED` for inclusion of the minted package for typesetting of code with the Pygments tool (which requires `latex` or `pdflatex` to be run with the `-shell-escape` option)

If you are not satisfied with the generated Doconce preamble, you can provide your own preamble by adding the command-line option `--latex_preamble=myfile`. In case `myfile` contains a documentclass definition, Doconce assumes that the file contains the *complete* preamble you want (not that all the packages listed in the default preamble are required and must be present in `myfile`). Otherwise, `myfile` is assumed to contain *additional* LaTeX code to be added to the Doconce default preamble.

The `ptex2tex` tool makes it possible to easily switch between many different fancy formattings of computer code in LaTeX documents. After any `!bc` command in the Doconce source you can insert verbatim block styles as defined in your `.ptex2tex.cfg` file, e.g., `!bc sys` for a terminal session, where `sys` is set to a certain environment in `.ptex2tex.cfg` (e.g., `CodeTerminal`). There are about 40 styles to choose from, and you can easily add new ones.

Also the `doconce ptex2tex` command supports preprocessor directives for processing the `.p.tex` file. The command allows specifications of code environments as well. Here is an example:

```
Terminal> doconce ptex2tex mydoc -DLATEX_HEADING=traditional \
          -DPALATINO -DA6PAPER \
          "sys=\begin{quote}\begin{verbatim}@\end{verbatim}\end{quote}" \
          fpro=minted fcod=minted shcod=Verbatim envir=ans:nt
```

Note that `@` must be used to separate the begin and end LaTeX commands, unless only the environment name is given (such as `minted` above, which implies `\begin{minted}{fortran}` and `\end{minted}` as begin and end for blocks inside `!bc fpro` and `!ec`). Specifying `envir=ans:nt` means that all other environments are typeset with the `anslistings.sty` package, e.g., `!bc cppcod` will then result in `\begin{c++}`. A predefined shortcut as in `shcod=Verbatim-0.85` results in denser vertical spacing (baselinestretch 0.85 in LaTeX terminology), and `shcod=Verbatim-indent` implies indentation of the verbatim text. Alternatively, one can provide all desired parameters `\begin{Verbatim}` instruction using the syntax illustrated for the `sys` environments above.

If no environments like `sys`, `fpro`, or the common `envir` are defined on the command line, the plain `\begin{Verbatim}` and `\end{Verbatim}` instructions are used.

Step 2b (optional). Edit the `mydoc.tex` file to your needs. For example, you may want to substitute `section` by `section*` to avoid numbering of sections, you may want to insert linebreaks (and perhaps space) in the title, etc. This can be automatically edited with the aid of the `doconce replace` and `doconce subst` commands. The former works with substituting text directly, while the latter performs substitutions using regular expressions. You will use `doconce replace` to edit `section{` to `section*{`:

```
Terminal> doconce replace 'section{' 'section*{' mydoc.tex
```

For fixing the line break of a title, you may pick a word in the title, say "Using", and insert a break after than word. With `doconce subst` this is easy employing regular expressions with a group before "Using" and a group after:

```
Terminal> doconce subst 'title\{(.+)Using (.+)\}' \
          'title{\g<1> \\\\ [1.5mm] Using \g<2>' mydoc.tex
```

A lot of tailored fixes to the LaTeX document can be done by an appropriate set of text replacements and regular expression substitutions. You are anyway encourged to make a script for generating PDF from the LaTeX file so the `doconce subst` or `doconce replace` commands can be put inside the script.

Step 3. Compile `mydoc.tex` and create the PDF file:

```
Terminal> latex mydoc
Terminal> latex mydoc
Terminal> makeindex mydoc   # if index
Terminal> bibitem mydoc     # if bibliography
Terminal> latex mydoc
Terminal> dvipdf mydoc
```

See the next two sections for compilation with XeLaTeX or PDFLaTeX.

If one wishes to use the minted LaTeX package for typesetting code blocks (`Minted_Python`, `Minted_Cpp`, etc., in `ptex2tex` specified through the `*pro` and `*cod` variables in `.ptex2tex.cfg` or `$HOME/.ptex2tex.cfg`), the minted LaTeX package is needed. This package is automatically included by `doconce ptex2tex` if the minted style is used, while you have to include the `-DMINTED` preprocessor option when running the `ptex2tex` program:

```
Terminal> ptex2tex -DMINTED mydoc
```

If the minted style is used, `latex` (or `pdflatex` or `xelatex`) *must* be run with the `-shell-escape` option:

```
Terminal> latex -shell-escape mydoc
Terminal> latex -shell-escape mydoc
Terminal> makeindex mydoc   # if index
Terminal> bibitem mydoc     # if bibliography
Terminal> latex -shell-escape mydoc
Terminal> dvipdf mydoc
```

3.10 PDFLaTeX

Running `pdflatex` instead of `latex` follows almost the same steps, but the start is

```
Terminal> doconce format latex mydoc
```

Then `ptex2tex` is run as explained above, and finally

```
Terminal> pdflatex -shell-escape mydoc
Terminal> makeindex mydoc   # if index
Terminal> bibitem mydoc     # if bibliography
Terminal> pdflatex -shell-escape mydoc
```

3.11 XeLaTeX

XeLaTeX is an alternative to PDFLaTeX and is run in almost the same way, except for the `-DXELATEX` flag to ptex2tex:

```
Terminal> doconce format pdflatex mydoc
Terminal> doconce ptex2tex mydoc -DXELATEX
Terminal> ptex2tex -DXELATEX mydoc  # alternative
Terminal> xelatex mydoc
```

3.12 Plain ASCII Text

We can go from Doconce "back to" plain untagged text suitable for viewing in terminal windows, inclusion in email text, or for insertion in computer source code:

```
Terminal> doconce format plain mydoc.do.txt  # results in mydoc.txt
```

3.13 reStructuredText

Going from Doconce to reStructuredText gives a lot of possibilities to go to other formats. First we filter the Doconce text to a reStructuredText file `mydoc.rst`:

```
Terminal> doconce format rst mydoc.do.txt
```

We may now produce various other formats:

```
Terminal> rst2html.py  mydoc.rst > mydoc.html # html
Terminal> rst2latex.py mydoc.rst > mydoc.tex  # latex
Terminal> rst2xml.py   mydoc.rst > mydoc.xml  # XML
Terminal> rst2odt.py   mydoc.rst > mydoc.odt  # OpenOffice
```

The OpenOffice file `mydoc.odt` can be loaded into OpenOffice and saved in, among other things, the RTF format or the Microsoft Word format. However, it is more convenient to use the program `unovonv` to convert between the many formats OpenOffice supports *on the command line*. Run

```
Terminal> unoconv --show
```

to see all the formats that are supported. For example, the following commands take `mydoc.odt` to Microsoft Office Open XML format, classic MS Word format, and PDF:

```
Terminal> unoconv -f ooxml mydoc.odt
Terminal> unoconv -f doc mydoc.odt
Terminal> unoconv -f pdf mydoc.odt
```

Remark about Mathematical Typesetting. At the time of this writing, there is no easy way to go from Doconce and LaTeX mathematics to reST and further to OpenOffice and the "MS Word world". Mathematics is only fully supported by `latex` as output and to a wide extent also supported by the `sphinx` output format. Some links for going from LaTeX to Word are listed below.

- http://ubuntuforums.org/showthread.php?t=1033441
- http://tug.org/utilities/texconv/textopc.html
- http://nileshbansal.blogspot.com/2007/12/latex-to-openofficeword.html

3.14 Sphinx

Sphinx documents demand quite some steps in their creation. We have automated most of the steps through the `doconce sphinx_dir` command:

```
Terminal> doconce sphinx_dir author="authors' names" \
          title="some title" version=1.0 dirname=sphinxdir \
          theme=mytheme mydoc
```

The keywords `author`, `title`, and `version` are used in the headings of the Sphinx document. By default, `version` is 1.0 and the script will try to deduce authors and title from the doconce file `mydoc.do.txt` The default value of `dirname` is `sphinx-rootdir`. The `theme` keyword is used to set the theme for design of HTML output from Sphinx (the default theme is `'default'`).

One often just runs the simple command

```
Terminal> doconce sphinx_dir mydoc
```

which creates the Sphinx directory `sphinx-rootdir` with relevant files.

The `doconce sphinx_dir` command generates a script `automake_sphinx.py` for compiling the Sphinx document into an HTML document. Run

```
Terminal> python automake_sphinx.py
```

As the output also tells, you can see the Sphinx HTML version of the document by running

```
Terminal> google-chrome sphinx-rootdir/_build/html/index.html
```

or loading the `index.html` file manually into your favorite web browser.

If you cycle through editing the Doconce file and watching the HTML output, you should observe that `automake_sphinx.py` does not recompile the Doconce file if the Sphinx `.rst` version already exists. In each edit-and-watch cycle do

```
Terminal> rm mydoc.rst; python automake_sphinx.py
```

Tip.

If you are new to Sphinx and end up producing quite some Sphinx documents, you are encouraged to read the Sphinx documentation and study the `automake_sphinx.py` file. Maybe you want to do things differently.

The following paragraphs describes the many possibilities for steering the Sphinx output.

Links. The `automake_sphinx.py` script copies directories named `fig*` over to the Sphinx directory so that figures are accessible in the Sphinx compilation. It also examines `MOVIE:` and `FIGURE:` commands in the Doconce file to find other image files and copies these too. I strongly recommend to put files to which there are local links (not `http:` or `file:` URLs) in a directory named `_static`. The `automake_sphinx.py` copies `_static*` to the Sphinx directory, which guarantees that the links to the local files will work in the Sphinx document.

There is a utility `doconce sphinxfix_localURLs` for checking links to local files and moving the files to `_static` and changing the links accordingly. For example, a link to `dir1/dir2/myfile.txt` is changed to `_static/myfile.txt` and `myfile.txt` is copied to `_static`. However, I recommend instead that you manually copy files to `_static` when you want to link to them, or let your script which compiles the Doconce document do it automatically.

Themes. Doconce comes with a rich collection of HTML themes for Sphinx documents, much larger than what is found in the standard Sphinx distribution. Additional themes include `agni`, `basicstrap`, `bootstrap`, `cloud`, `fenics`, `fenics_minimal`, `flask`, `haiku`, `impressjs`, `jal`, `pylons`, `redcloud`, `scipy_lectures`, `slim-agogo`, and `vlinux-theme`.

All the themes are packed out in the Sphinx directory, and the `doconce sphinx_dir` insert lots of extra code in the `conf.py` file to enable easy specification and customization of themes. For example, modules are loaded for the additional themes that come with Doconce, code is inserted to allow customization of the look and feel of themes, etc. The `conf.py` file is a good starting point for fine-tuning your favorite team, and your own `conf.py` file can later be supplied and used when running `doconce sphinx_dir`: simply add the command-line option `conf.py=conf.py`.

A script `make-themes.sh` can make HTML documents with one or more themes. For example, to realize the themes `fenics`, `pyramid`, and `pylon` one writes

```
Terminal> ./make-themes.sh fenics pyramid pylon
```

The resulting directories with HTML documents are `_build/html_fenics` and `_build/html_pyramid`, respectively. Without arguments, `make-themes.sh` makes all available themes (!). With `make-themes.sh` it is easy to check out various themes to find the one that is most attractive for your document.

You may supply your own theme and avoid copying all the themes that come with Doconce into the Sphinx directory. Just specify `theme_dir=path` on the command line, where `path` is the relative path to the directory containing the Sphinx theme. You must also specify a configure file by `conf.py=path`, where `path` is the relative path to your `conf.py` file.

Example. Say you like the `scipy_lectures` theme, but you want a table of contents to appear *to the right*, much in the same style as in the `default` theme

(where the table of contents is to the left). You can then run `doconce sphinx_dir`, invoke a text editor with the `conf.py` file, find the line `html_theme == 'scipy_lectures'`, edit the following `nosidebar` to `false` and `rightsidebar` to `true`. Alternatively, you may write a little script using `doconce replace` to replace a portion of text in `conf.py` by a new one:

```
doconce replace "elif html_theme == 'scipy_lectures':
    html_theme_options = {
        'nosidebar': 'true',
        'rightsidebar': 'false',
        'sidebarbgcolor': '#f2f2f2',
        'sidebartextcolor': '#20435c',
        'sidebarlinkcolor': '#20435c',
        'footerbgcolor': '#000000',
        'relbarbgcolor': '#000000',
    }" "elif html_theme == 'scipy_lectures':
    html_theme_options = {
        'nosidebar': 'false',
        'rightsidebar': 'true',
        'sidebarbgcolor': '#f2f2f2',
        'sidebartextcolor': '#20435c',
        'sidebarlinkcolor': '#20435c',
        'footerbgcolor': '#000000',
        'relbarbgcolor': '#000000',
    }" conf.py
```

Obviously, we could also have changed colors in the edit above. The final alternative is to save the edited `conf.py` file somewhere and reuse it the next time `doconce sphinx_dir` is run

```
doconce sphinx_dir theme=scipy_lectures \
                   conf.py=../some/path/conf.py mydoc
```

The manual Sphinx procedure. If it is not desirable to use the autogenerated scripts explained above, here is the complete manual procedure of generating a Sphinx document from a file `mydoc.do.txt`.

Step 1. Translate Doconce into the Sphinx format:

```
Terminal> doconce format sphinx mydoc
```

Step 2. Create a Sphinx root directory either manually or by using the interactive `sphinx-quickstart` program. Here is a scripted version of the steps with the latter:

```
mkdir sphinx-rootdir
sphinx-quickstart <<EOF
sphinx-rootdir
n

_
Name of My Sphinx Document
Author
version
version
.rst
```

```
index
n
y
n
n
n
n
y
n
n
y
y
y
EOF
```

The autogenerated `conf.py` file may need some edits if you want to specific layout (Sphinx themes) of HTML pages. The `doconce sphinx_dir` generator makes an extended `conv.py` file where, among other things, several useful Sphinx extensions are included.

Step 3. Copy the `mydoc.rst` file to the Sphinx root directory:

```
Terminal> cp mydoc.rst sphinx-rootdir
```

If you have figures in your document, the relative paths to those will be invalid when you work with `mydoc.rst` in the `sphinx-rootdir` directory. Either edit `mydoc.rst` so that figure file paths are correct, or simply copy your figure directories to `sphinx-rootdir`. Links to local files in `mydoc.rst` must be modified to links to files in the `_static` directory, see comment above.

Step 4. Edit the generated `index.rst` file so that `mydoc.rst` is included, i.e., add `mydoc` to the `toctree` section so that it becomes

```
.. toctree::
   :maxdepth: 2

   mydoc
```

(The spaces before `mydoc` are important!)

Step 5. Generate, for instance, an HTML version of the Sphinx source:

```
make clean   # remove old versions
make html
```

Sphinx can generate a range of different formats: standalone HTML, HTML in separate directories with `index.html` files, a large single HTML file, JSON files, various help files (the qthelp, HTML, and Devhelp projects), epub, LaTeX, PDF (via LaTeX), pure text, man pages, and Texinfo files.

Step 6. View the result:

```
Terminal> firefox _build/html/index.html
```

Note that verbatim code blocks can be typeset in a variety of ways depending the argument that follows `!bc`: `cod` gives Python (`code-block:: python` in Sphinx syntax) and `cppcod` gives C++, but all such arguments can be customized both for Sphinx and LaTeX output.

3.15 Wiki Formats

There are many different wiki formats, but Doconce only supports three: Googlecode wiki, MediaWiki, and Creole Wiki. These formats are called `gwiki`, `mwiki`, and `cwiki`, respectively. Transformation from Doconce to these formats is done by

```
Terminal> doconce format gwiki mydoc.do.txt
Terminal> doconce format mwiki mydoc.do.txt
Terminal> doconce format cwiki mydoc.do.txt
```

The produced MediaWiki can be tested in the sandbox of wikibooks.org. The format works well with Wikipedia, Wikibooks, and ShoutWiki, but not always well elsewhere (see this example).

Large MediaWiki documents can be made with the Book creator. From the MediaWiki format one can go to other formats with aid of mwlib. This means that one can easily use Doconce to write Wikibooks and publish these in PDF and MediaWiki format, while at the same time, the book can also be published as a standard LaTeX book, a Sphinx web document, or a collection of HTML files.

The Googlecode wiki document, `mydoc.gwiki`, is most conveniently stored in a directory which is a clone of the wiki part of the Googlecode project. This is far easier than copying and pasting the entire text into the wiki editor in a web browser.

When the Doconce file contains figures, each figure filename must in the `.gwiki` file be replaced by a URL where the figure is available. There are instructions in the file for doing this. Usually, one performs this substitution automatically (see next section).

3.16 Google Docs

Google Docs are normally made online in the interactive editor. However, you may upload a Doconce document to Google Docs. This requires transforming the Doconce document to one of the accepted formats for Google Docs:

- OpenOffice: `doconce format rst` and then run `rst2odt` (or `rst2odt.py`). Upload the `.odt` file, click *Open...* in Google Drive and choose *Google Docs* as viewer.

- MS Word: `doconce format pandoc` and then run `pandoc` to produce a `.docx` file that can be uploaded to Google Drive and opened in Google Docs.
- RTF: `doconce format pandoc` and then run `pandoc` to produce a `.rtf` file that can be uploaded to Google Drive and opened. Another possibility is to run `doconce format latex` and then latex2rtf (the support of mathematics has gotten worse).
- Plain text: `doconce format plain`. Upload the `.txt` file to Google Drive and open in Google Docs.
- HTML: `doconce format html`. Upload the `.html` file and open in Google Docs. Complicated HTML files can be misinterpreted by Google Docs.

This is not yet much tested. It remains to see how code becomes in Google Docs. Support for mathematics is probably impossible until Google Docs can import LaTeX files, but LaTeX mathematics can be embedded in Google Docs and the googledoc2latex script can convert a Google document to LaTeX.

3.17 Tweaking the Doconce Output

Occasionally, one would like to tweak the output in a certain format from Doconce. One example is figure filenames when transforming Doconce to reStructuredText. Since Doconce does not know if the `.rst` file is going to be filtered to LaTeX or HTML, it cannot know if `.eps` or `.png` is the most appropriate image filename. The solution is to use a text substitution command or code with, e.g., sed, perl, python, or scitools subst, to automatically edit the output file from Doconce. It is then wise to run Doconce and the editing commands from a script to automate all steps in going from Doconce to the final format(s). The `make.sh` files in `docs/manual` and `docs/tutorial` constitute comprehensive examples on how such scripts can be made.

4 The Doconce Markup Language

The Doconce format introduces four constructs to markup text: lists, special lines, inline tags, and environments.

4.1 Lists

An unordered bullet list makes use of the * as bullet sign and is indented as follows

```
* item 1

* item 2
```

```
  * subitem 1, if there are more
    lines, each line must
    be intended as shown here

  * subitem 2,
    also spans two lines

* item 3
```

This list gets typeset as

- item 1
- item 2
  - subitem 1, if there are more lines, each line must be intended as shown here
  - subitem 2, also spans two lines
- item 3

In an ordered list, each item starts with an `o` (as the first letter in "ordered"):

```
o item 1
o item 2
  * subitem 1
  * subitem 2
o item 3
```

resulting in

1. item 1
2. item 2
   - subitem 1
   - subitem 2
3. item 3

Ordered lists cannot have an ordered sublist, i.e., the ordering applies to the outer list only.

In a description list, each item is recognized by a dash followed by a keyword followed by a colon:

```
- keyword1: explanation of keyword1

- keyword2: explanation
  of keyword2 (remember to indent properly
  if there are multiple
  lines)
```

The result becomes

keyword1: explanation of keyword1

keyword2: explanation of keyword2 (remember to indent properly if there are multiple lines)

4.2 Special lines

The Doconce markup language has a concept called *special lines*. Such lines starts with a markup at the very beginning of the line and are used to mark document title, authors, date, sections, subsections, paragraphs, figures, movies, etc.

4.3 Heading with title and author(s)

Lines starting with `TITLE:`, `AUTHOR:`, and `DATE:` are optional and used to identify a title of the document, the authors, and the date. The title is treated as the rest of the line, so is the date, but the author text consists of the name and associated institution(s) with the syntax

```
name at institution1 and institution2 and institution3
```

The `at` with surrounding spaces is essential for adding information about institution(s) to the author name, and the `and` with surrounding spaces is essential as delimiter between different institutions. An email address can optionally be included, using the syntax

```
name Email: somename@site.net at institution1 and institution2
```

Multiple authors require multiple `AUTHOR:` lines. All information associated with `TITLE:` and `AUTHOR:` keywords must appear on a single line. Here is an example:

```
TITLE: On an Ultimate Markup Language
AUTHOR: H. P. Langtangen at Center for Biomedical Computing, Simula Research Laboratory & Dept. of Informati
AUTHOR: Kaare Dump Email: dump@cyb.space.com at Segfault, Cyberspace Inc.
AUTHOR: A. Dummy Author
DATE: November 9, 2016
```

Note how one can specify a single institution, multiple institutions (with `&` as separator between institutions), and no institution. In some formats (including `rst` and `sphinx`) only the author names appear. Some formats have "intelligence" in listing authors and institutions, e.g., the plain text format:

```
Hans Petter Langtangen [1, 2]
Kaare Dump  (dump@cyb.space.com) [3]
A. Dummy Author

[1] Center for Biomedical Computing, Simula Research Laboratory
[2] Department of Informatics, University of Oslo
[3] Segfault, Cyberspace Inc.
```

Similar typesetting is done for LaTeX and HTML formats.

The current date can be specified as `today`.

4.4 Table of contents

A table of contents can be generated by the line

```
TOC: on
```

This line is usually placed after the `DATE:` line. The value `off` turns off the table of contents.

4.5 Section headings

Section headings are recognized by being surrounded by equal signs (=) or underscores before and after the text of the headline. Different section levels are recognized by the associated number of underscores or equal signs (=):

- 9 = characters for chapters
- 7 for sections
- 5 for subsections
- 3 for subsubsections
- 2 *underscrores* (only! - it looks best) for paragraphs (paragraph heading will be inlined)

Headings can be surrounded by as many blanks as desired.

Doconce also supports abstracts. This is typeset as a paragraph, but *must* be followed by a section heading (everything up to the first section heading is taken as part of the text of the abstract).

Here are some examples:

```
__Abstract.__ The following text just attempts to exemplify
various section headings.

Appendix is supported too: just let the heading start with "Appendix: "
(this affects only ‘latex‘ output, where the appendix formatting
is used - all other formats just leave the heading as it is written).

======= Example on a Section Heading =======

The running text goes here.


===== Example on a Subsection Heading =====

The running text goes here.

=== Example on a Subsubsection Heading ===

The running text goes here.

__A Paragraph.__ The running text goes here.
```

4.6 Figures

Figures are recognized by the special line syntax

```
FIGURE:[filename, height=400 width=600 frac=0.8] caption
```

The filename can be without extension, and Doconce will search for an appropriate file with the right extension. If the extension is wrong, say `.eps` when requesting an HTML format, Doconce tries to find another file, and if not, the given file is converted to a proper format (using ImageMagick's `convert` utility).

The height, width, and frac keywords can be included if desired and may have effect for some formats: the height and width are used for output in the formats `html`, `rst`, `sphinx`, while the frac specification is used for `latex` and `pdflatex` to specify the width of the image as a fraction of the text width.

The figure caption is optional. If omitted, the figure appears "inline" in the text without any figure environment in LaTeX formats or HTML. The caption may contain a label for referencing the figure.

Warning.

Note the comma between the filename and the figure size specifications and that there should be no space around the = sign. This syntax must be strictly followed.

Note also that, like for `TITLE:` and `AUTHOR:` lines, all information related to a figure line *must be written on the same line*. Introducing newlines in a long caption will destroy the formatting (only the part of the caption appearing on the same line as `FIGURE:` will be included in the formatted caption).

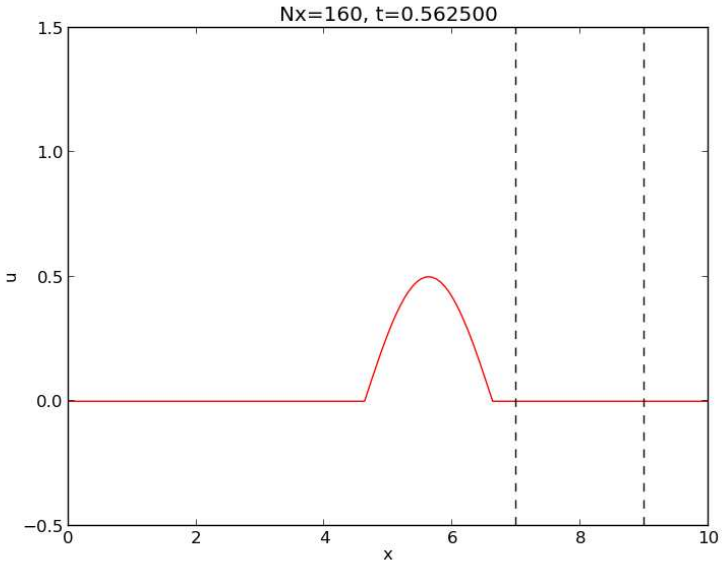

Figure 1: A wave.

Combining several image files into one, in a table fashion, can be done by the `montage` program from the ImageMagick suite:

```
montage -background white -geometry 100% -tile 2x \
        file1.png file2.png ... file4.png result.png
```

The option `-tile XxY` gives `X` figures in the horizontal direction and `Y` in the vertical direction (`tile 2x` means two figures per row and `-tile x2` means two rows).

The `montage` program is only appropriate for bitmap images (PNG, JPEG, GIF, TIFF). Images in the PDF format should be mounted together using `pdftk` (to combine images to one file), `pdfnup` (to align them in tabular format), and `pdfcrop` (to remove surrounding whitespace):

```
Terminal> pdftk file1.pdf file2.pdf ... file4.pdf output tmp.pdf
Terminal> pdfnup --nup 2x2 tmp.pdf        # output in tmp-nup.pdf
Terminal> pdfcrop tmp-nup.pdf result.png  # output in FE1.png
```

4.7 Movies

Movies/videos are inserted using the `MOVIE:` keyword. This feature works well for the `latex`, `html`, `rst`, and `sphinx` formats. Other formats try to generate some HTML file and link to that file for showing the movie.

The Basic Command. As with `FIGURE`, the `MOVIE` command expands just *one line* and is of the form

```
MOVIE: [filename, height=xxx width=yyy] possible caption
```

Note that there must be a blank line after every `MOVIE:` command. The width and height parameters are not required, but leaving them out may lead to movie sizes you do not want.

Here is a movie in the MPEG format:

mov/wave.mpeg

And here is a movie in the Ogg format:

mov/wave.ogg

Movie 2: A movie in Ogg format.

A URL works too as movie address:

http://hplgit.github.io/animate/doc/pub/mov-animate/demo.ogg

Movie 3: Ogg movie in cyberspace.

MP4, WebM, and Ogg Movies in HTML. If a movie is in Ogg, MP4, or WebM format, and the output format is HTML, Doconce will check if the movie file is also in the other formats among Ogg, MP4, and WebM, and include these as well such that the movie has backup formats in case the browser does not support a particular format. Providing a movie in Ogg, MP4, *and* WebM format is therefore the safest way to ensure that the movie can be played in any browser on any device.

Notice.

If you specify a movie in Ogg or WebM format and it also exists in MP4 format, the MP4 format will be loaded first. To avoid having alternative movie formats in HTML, use the `--no_mp4_webm_ogg_alternatives` command-line option when running `doconce format`.

Movie Handling in Various Formats. Movies are easiest shown in the HTML format. The reST and Sphinx formats apply the same raw HTML code as the HTML format and therefore have the same capabilities. The LaTeX format results in a `.p.tex` file that have three methods for showing movies, set by the option `-DMOVIE=...` to `ptex2tex` or `doconce ptex2tex`. The values of `MOVIE` are

1. `media9`: the `media9` package is used for Flash and MP4 movies, `movie15` for MPEG and AVI files, and a simple `\href{run:file}{link}` command for other formats. Only Acrobat Reader supports displaying these type of movies.
2. `multimedia`: the `\movie` command (known from LaTeX Beamer) is used for movies.
3. `href`: the `\href{run:file}{link}` is used for all movies (default).

For all other formats, an HTML file that acts as a movie player is generated and linked from the output document. This movie player has essentially the same code as the HTML format would have, except that the `video` tag is not used, only the `embed` tag. Some wiki types do have support for videos, e.g., Wikipedia can work with Ogg files, but Doconce has not yet implemented robust schemes for anything but LaTeX, HTML, and Sphinx output.

YouTube and Vimeo Movies. Many publish their scientific movies on YouTube or Vimeo, and Doconce recognizes YouTube and Vimeo URLs as movies. When the output from Doconce is an HTML file, the movie will be embedded, otherwise a URL to the YouTube or Vimeo page is inserted. You should equip the `MOVIE:` command with the right width and height of *embedded* YouTube and Vimeo movies. The recipe goes as follows:

1. click on *Share* (and on YouTube then *Embed*)
2. note the height and width of the embedded movie

A typical `MOVIE` command with a YouTube movie is then

```
MOVIE: [http://www.youtube.com/watch?v=sI2uCHH3qIM, width=420 height=315]

MOVIE: [http://vimeo.com/55562330, width=500 height=278] CFD.
```

Animation Based on Filename Generators. It is possible to define a movie from a set of files, usually plot files, which can be shown in sequence to create an animation. If the files are local on the computer, one can specify them by a simple Unix wildcard notation, as in

```
MOVIE: [../experiments/frame_*.png]
```

Output in the HTML, reST, and Sphinx formats will make use of inline JavaScript code to show the frames in sequence. LaTeX employs the `animate` package for the same purpose. Other formats generates a file (`movie_playerX.py`, where `X` is a number) containing the HTML code with JavaScript to show and control the animation. The Doconce document has a link to this movie viewer.

There is an alternative syntax to the Unix wildcard notation:

```
MOVIE: [../experiments/frame_%04d.png:0->320]
```

The filename is specified via printf syntax (typically the same syntax as used to generate the individual frame files). The postfix `:0->320` specifies the lower and upper limit of the counter that is used in the printf specification `%04d`. This latter syntax must be used if the plot files reside on some web server, e.g.,

```
MOVIE: [http://some.where.net/experiments/frame_%04d.png:0->320]
```

Here is an example:

`http://hplgit.github.io/animate/doc/pub/mov-animate/frames/frame_%04d.png:0->320`:
load `movie_player1.html` into a browser

Recommendations.

It is challenging to write robust Doconce code with movies. The recommended formats in HTML are MP4, WebM, and Ogg. One should preferably make all three. These also works in reST and Sphinx.

The filename generation works very well in LaTeX, while true movie formats pose big challenges. On Linux systems, the `media9` does not work well because a proper Flash player for embedding in the PDF file is not always available. The `movie15` package also leads to problems because Acrobat Reader depends on an external player to show the files, and the correct plugins to launch players with support for a given format

are not trivial to install. Even the plain `href{run:file}` command relies on an external player and not all formats will be supported on a given computer.

To have really robust code, use filename generators and not movie files.

```
MOVIE: [../experiments/frame_*.png]
```

One can, of course, write flexible Doconce code and decide at run time if HTML output should have movie files or filename generators. A relevant snippet using Mako is

```
% FORMAT in ("latex", "pdflatex") or HTMLMOVIE == "files":
MOVIE: [../experiments/frame_*.png]

% else:
MOVIE: [../experiments/movie.ogg]

% endif
```

With the `-DHTMLMOVIE=files` flag, animation of individual files will be performed, while any other value than `files` leads to use of the `movie.ogg` in all but LaTeX formats. In HTML one will try to load `movie.mp4` (if it exists) and then `movie.webm` (if it exists) and then finally `movie.ogg`.

There is no way to control the number of frames per second in LaTeX animations based on filename generators such as `myframes*.png`. However, with a little auto editing in a script one can control the frame rates of the various movies. The rate is specified as `2` in lines on the form

```
\begin{animateinline}[controls,loop]{2} % frames: f000.png -> f098.png
```

Setting the rate to 12 in for this particular movie based on the `f%03.png` files, the following `doconce subst` command does the job in a script:

```
doconce subst ',loop]{2}( .+: f000)' ',loop{12}\g<1>' mydoc.do.txt
```

4.8 Copying Computer Code from Source Files

Another type of special lines starts with `@@@CODE` and enables copying of computer code from a file directly into a verbatim environment, see Section 4.18 below.

4.9 Inserting the Output from Operating System Commands

When Doconce is used to document computer program and results from computer code it is important to ensure that the document contains the latest version of the code and the corresponding output. The former is handled by the

`@@@CODE` directive, while the latter has its own directive `@@@OSCMD`. The syntax reads

```
@@@OSCMD cmd
```

where `cmd` is any text that can be run in the operating system. The output is copied into the Doconce source. For example,

```
@@@OSCMD python -c 'print "Hello,\nWorld!"'
```

results in

```
Terminal> python -c 'print "Hello,\nWorld!"'
Hello,
World!
```

There is a command-line option `--os_prompt=` that can be used to set the (terminal) prompt that prefixes the command:

- `--os_prompt=None` results in no prompt, just the command.
- `--os_prompt=nocmd` results in no prompt and no command, just the output.
- `--os_prompt=Terminal>` is the default setting (as in the example above).

4.10 Inline Tagging

Doconce supports tags for *emphasized phrases*, **boldface phrases**, and `verbatim text` (also called type writer text, for inline code), colored words, plus LaTeX/TeX inline mathematics, such as $\nu = \sin(x)$.

Emphasized Words. Emphasized text is typeset inside a pair of asterisk, and there should be no spaces between an asterisk and the emphasized text, as in

```
*emphasized words*
```

Boldface font is recognized by an underscore instead of an asterisk:

```
_several words in boldface_ followed by *ephasized text*.
```

The line above gets typeset as **several words in boldface** followed by *ephasized text*.

Inline Verbatim Text. Verbatim text, typically used for short inline code, is typeset between back-ticks:

```
‘call myroutine(a, b)‘ looks like a Fortran call
while ‘void myfunc(double *a, double *b)‘ must be C.
```

The typesetting result looks like this: `call myroutine(a, b)` looks like a Fortran call while `void myfunc(double *a, double *b)` must be C.

It is recommended to have inline verbatim text on the same line in the Doconce file, because some formats (LaTeX and `ptex2tex`) will have problems with inline verbatim text that is split over two lines.

Notice.

Watch out for mixing back-ticks and asterisk (i.e., verbatim and emphasized code): the Doconce interpreter is not very smart so inline computer code can soon lead to problems in the final format. Go back to the Doconce source and modify it so the format to which you want to go becomes correct (sometimes a trial and error process - sticking to very simple formatting usually avoids such problems).

Links to Web Addresses. Web addresses with links are typeset as

```
some URL like "Search Google": "http://google.com".
```

which appears as some URL like Search Google. The space after colon is optional, but it is important to enclose the link and the URL in double quotes.

To have the URL address itself as link text, put an "URL" or URL before the address enclosed in double quotes:

```
Click on this link: URL: "https://github.com/hplgit/doconce".
```

which gets rendered as Click on this link: https://github.com/hplgit/doconce.

(There is also support for lazy writing of URLs: any http or https web address with a leading space and a trailing space, comma, semi-colon, or question mark (but not period!) becomes a link with the web address as link text.)

Mail Addresses. Links that launches a mail to a specified address is written as ordinary URLs, typically as

```
Send "mail": "mailto:hpl@simula.no"
# Alternative:
to "‘hpl@simula.no‘": "mailto:hpl@simula.no".
```

which appears as Send mail to hpl@simula.no.

Links to Local Files. Links to files ending in `.txt`, `.html`, `.pdf`, `.py`, `.f`, `.f77`, `.f90`, `.f95`, `.sh`, `.csh`, `.ksh`, `.zsh`, `.c`, `.cpp`, `.cxx`, `.pl`, and `.java` follows the same setup:

```
see the "Doconce Manual": "manual.do.txt".
```

which appears as see the Doconce Manual. However, linking to local files like this needs caution:

- In the `html` format the links work well if the files are supplied with the `.html` with the same relative location.
- In the `latex` and `pdflatex` formats, such links in PDF files will unless the `.tex` file has a full URL specified through a `\hyperbaseurl` command and the linked files are located correctly relative to this URL. Otherwise full URL must be used in links.
- In the `sphinx` format, links to local files do not work unless the files reside in a `_static` directory (a warning is issued about this).

As a consequence, we strongly recommend that one copies the relevant files to a `_static` or `_static-name` directory and makes links to files in this directory only (`name` is the nickname of the Doconce document, usually the name of the parent directory or main document). Other links to files should use the full URL. If Doconce is used for HTML output only, then plain links to local files work fine.

If you want a link to a local source code file and have it viewed in the browser rather than being downloaded, we recommend to transform the source code file to HTML format by running `pygmentize`, e.g.,

```
Terminal> pygmentize -l bash -f html -O full,style=emacs \
          -o _static/make.sh.html subdir/make.sh
```

Then you can link to `_static/make.sh.html` instead of `subdir/make.sh`. Here is an example where the reader has the file available as `src/myprog.py` in her software and the document links to `_static/myprog.py`:

```
See the code URL:"src/myprog.py" ("view: "_static/myprog.py.html").
```

Links to files with other extensions are typeset with *the filename as link text*. The syntax consists of the keyword URL, followed by a colon, and then the filename enclosed in double quotes:

```
URL: "manual.html"
```

resulting in the link `manual.html`.

Quotes. Quotations employ either the emphasized font or double quotation marks. In the latter case, one should not use the character `"` but rather the (LaTeX-inspired) construction with double backticks and two single quotes:

```
This is a sentence with ``words to be quoted''.
```

Inline Comments. Doconce also supports inline comments in the text:

```
[name: comment]
```

where `name` is the name of the author of the command, and `comment` is a plain text text. Note that there must be a space after the colon, otherwise the comment is not recognized. **hpl 1**: *Inline comments can span several lines, if desired.*

The name and comment are visible in the output unless `doconce format` is run with a command-line argument `--skip_inline_comments` (see Section 3 for an example). Inline comments are helpful during development of a document since different authors and readers can comment on formulations, missing points, etc. All such comments can easily be removed from the `.do.txt` file (see Section 3).

Inline comments are typeset in a simple way (boldface name and the comment in parenthesis), but in LaTeX very visible color boxes are used (via the `todonotes` package).

Forced Line Breaks. By ending a line with `<linebreak>` the output format has a forced linebreak at this point. This can be used to typeset poems, songs (if not in a verbatim block), or the origin of quotes. Here is an example:

```
!bquote
*Program writing is substantially more demanding than book
writing. Why is it so? I think the main reason is that a larger
attention span is needed when working on a large computer program
than when doing other intellectual tasks.* <linebreak>
Donald Knuth cite[p. 18]{Knuth85}, computer scientist, 1938-.
!equote
```

is rendered as

> *Program writing is substantially more demanding than book writing. Why is it so? I think the main reason is that a larger attention span is needed when working on a large computer program than when doing other intellectual tasks.*
> Donald Knuth [1, p. 18], computer scientist, 1938-.

Inline Mathematics. Inline mathematics is written as in LaTeX, i.e., inside dollar signs. Many formats leave this syntax as it is (including to dollar signs), hence nice math formatting is only obtained in LaTeX, HTML, MediaWiki, and Sphinx (Epytext has some inline math support that is utilized). However, mathematical expressions in LaTeX syntax often contains special formatting commands, which may appear annoying in plain text. Doconce therefore supports an extended inline math syntax where the writer can provide an alternative syntax suited for formats close to plain ASCII:

```
Here is an example on a linear system
${\bf A}{\bf x} = {\bf b}$|$Ax=b$,
where $\bf A$|$A$ is an $n\times n$|$nxn$ matrix, and
$\bf x$|$x$ and $\bf b$|$b$ are vectors of length $n$|$n$.
```

That is, we provide two alternative expressions, both enclosed in dollar signs and separated by a pipe symbol, the expression to the left is used in formats with LaTeX support (`latex`, `pdflatex`, `html`, `sphinx`, `mwiki`), while the expression to the right is used for all other formats. The above text is typeset as "Here is an example on a linear system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an $n \times n$ matrix, and $\mathbf{x}$ and $\mathbf{b}$ are vectors of length n."

4.11 Comments

Comments intended to be (sometimes) visible in the output document and read by readers are known as *inline comments* in Doconce and described in Section 4.10.

Here we address comments in the Doconce source file that are not intended to be visible in the output document. Basic comment lines start with the hash `#`:

```
#
# Here are some comment lines that do not affect any formatting.
# These lines are converted to comments in the output format.
#
```

Such comment lines may have some side effects in the `rst` and `sphinx` formats because following lines are taken as part of the comment if there is not a blank line after the comment.

The Mako preprocessor supports comments that are filtered out *before* Doconce starts translating the document. Such comments are very valuable as they will never interfere with the output format and they are only present in the Doconce source. Mako has two types of comments: lines starting with a double hash `##` and lines enclosed by the `<%doc>` (beginning) and `<%doc/>` (closing) tags.

If you need a lot of comments in the Doconce file, consider using Mako comments instead of the single hash, unless you want the comments to be in the source code of the output document.

To comment out or remove large sections, consider using the Preprocess preprocessor and an if-else block with a variable that is undefined (typically something like a test `# #ifdef EXTRA` in Preprocess).

4.12 Cross-Referencing

References and labels are supported. The syntax is simple:

```
label{section:verbatim}   # defines a label
For more information we refer to Section ref{section:verbatim}.
```

This syntax is close that that of labels and cross-references in LaTeX. When the label is placed after a section or subsection heading, the plain text, Epytext, and StructuredText formats will simply replace the reference by the title of the (sub)section. All labels will become invisible, except those in math environments. In the `rst` and `sphinx` formats, the end effect is the same, but the

"label" and "ref" commands are first translated to the proper reST commands by `doconce format`. In the HTML and (Google Code) wiki formats, labels become anchors and references become links, and with LaTeX "label" and "ref" are just equipped with backslashes so these commands work as usual in LaTeX.

It is, in general, recommended to use labels and references for (sub)sections, equations, and figures only. By the way, here is an example on referencing Figure 1 (the label appears in the figure caption in the source code of this document). Additional references to Sections 4.19 and 4.20 are nice to demonstrate, as well as a reference to equations, say (**??**)-(**??**). A comparison of the output and the source of this document illustrates how labels and references are handled by the format in question.

Hyperlinks to files or web addresses are handled as explained in Section 4.10.

4.13 Generalized Cross-Referencing

Sometimes a series of individual documents may be assembled to one large document, typically a book. In the book one wants to make cross references between chapters and sections, while these become references to external documents when the chapters (or sections) are compiled as stand-alone documents. For example, one can in a Doconce file `file1.do.txt` have text like

```
...as shown in Section ref{sec:eqs}.
```

with `\label{sec:eqs}` defined in another file `file2.do.txt`. If `file1.do.txt` and `file2.do.txt` are combined to a single document, the reference is treated correctly, but if `file1.do.txt` is compiled as a single document, the label `sec:eqs` becomes undefined. Then one would instead write

```
...as shown in the document "Mathematical Equations":
"http://some.net/doc/matheqs.html" cite{math_eqs_2020}.
```

However, LaTeX has functionality for referring to labels in external documents: these work fine when the `xr` package is used and the external documents are listed as `externaldocument{name}`. In that case, we would write the above reference as

```
\externaldocument{file2}
...
...as shown in Section ref{sec:eqs} in cite{math_eqs_2020}.
```

To treat all these cases simultanesouly, Doconce features *generalized references* which allows a reference to have different formulations, depending on whether the label is present in the document or is present in an external document. The syntax of generalized references reads

```
ref[internal][cite][external]
```

If all `ref{label}` references in the text `internal` are to labels defined in the Doconce document, the generalized reference becomes the text `internal`. If

one or more labels are not present in the document and `latex` or `pdflatex` is output format, the generalized reference becomes the text `internal` plus `cite`, while for all other formats the text in `external` is used.

The example above can now be written as the generalized reference

```
...as shown in ref[Section ref{sec:eqs}][ in cite{math_eqs_2020}][
the document "Mathematical Equations":
"http://some.net/doc/matheqs.html" cite{math_eqs_2020}].
```

When `\label{sec:eqs}` is found in the current Doconce document, the generalized reference becomes

```
Section ref{sec:eqs}
```

If not, and `latex` or `pdflatex` is output, the reference becomes

```
Section ref{sec:eqs}] in cite{math_eqs_2020}
```

while in all other cases the reference becomes

```
the document "Mathematical Equations":
"http://some.net/doc/matheqs.html" cite{math_eqs_2020}
```

For the reference to a label in an external document to work in the LaTeX case it is required to list this document in the Doconce file as

```
# Externaldocuments: file2
```

Several external documents can be listed with comma as delimiter:

```
# Externaldocuments: file2, file3, myfile
```

on *a single line*. The `Externaldocuments` comment leads to use of the `xr` package and insertion of `\externaldocument{file2}` in the LaTeX output file. It is a good habit to place the `Externaldocument` comment after the title, author, and date.

Very often a reference to a chapter in a book becomes just a reference to a complete document, not a specific section, if chapters are compiled individually. For `latex` and `pdflatex` output one would then do a reference `Chapter~\ref{ch:eqs}` as just `\cite{eqs_doc_2008}`. That is, one would just use the `cite` text and not `internal` plus `cite`. This is enabled by writing the generalized reference with `refch` instead of `ref`:

```
...as shown in refch[Chapter ref{ch:eqs}][cite{eqs_doc_2008}][
the document "Some Equations": "http://some.net/someeqs/"].
```

The `doconce ref_external` command will read all the labels in the external documents listed in the `Externaldocuments:` comment and use the publish file of the current document to automatically generate substitution commands that translates ordinary references to generalized references. For example, `doconce ref_external file1` will find the reference

```
......as shown in Section ref{sec:eqs}.
```

as a reference to an label `sec:eqs` defined in `file2`, grab the title of `file2.do.txt` and find the bibliographic data in the publish file and make a substitution command

```
doconce subst "Section\s+ref{sec:eqs}" "..." $files
```

where "..." is the generalized reference as shown in the examples above. In other words, with `doconce ref_external` one can automatically generate generalized references between, for example, chapters in a book that exist as stand-alone documents.

Warning.

Generalized references to equations work well in LaTeX, but not in other formats as one cannot resolve the equation number in the external document. It is then better to write different text using the `FORMAT` variable in Mako:

```
% if FORMAT in ("pdflatex", "latex"):
By combining ref[(ref{eqs:g1})-(ref{eqs:g4})][ in cite{some_doc}][
dummy] we can derive the expression ...
% else:
One can from cite{some_doc} derive the expression
% endif
```

The `doconce ref_external` tool generates an `external` text in case of references to equations that says "reference to specific **equations** (label eqs:g1 and eqs:g4) in external document "name": "link" is not recommended". One can then search for this text and make a Mako if-else rewrite as shown above.

Limited support.

The `doconce ref_external` tool cannot correctly handle references to a range of sections like

```
Sections ref{mydoc:sec1}-ref{mydoc:sec2}
```

The automatically generated generalized references should always be manually checked and edited.

A Worked Example. Here is an example on a specific working generalized reference where the LaTeX output also has a hyperlink:

```
As explained in
ref[Section ref{subsec:ex}][in "Langtangen, 2012":
```

```
"http://hplgit.github.io/doconce/test/demo_testdoc.html#subsec:ex"
cite{Doconce:test}][a "section":
"http://hplgit.github.io/doconce/test/demo_testdoc.html#subsec:ex" in
the document "A Document for Testing Doconce":
"http://hplgit.github.io/doconce/test/demo_testdoc.html"
cite{Doconce:test}], Doconce documents may include tables.
```

With `latex` or `pdflatex` as output, this translates to

```
As explained in
Section ref{subsec:ex}, Doconce documents may include tables.
```

if the label `{subsec:ex}` appears in the present Doconce source, and otherwise

```
As explained in
Section ref{subsec:ex} in "Langtangen, 2012":
"http://hplgit.github.io/doconce/test/demo_testdoc.html#subsec:ex"
cite{Doconce:test}, Doconce documents may include tables.
```

The latter Doconce code is translated to the following LaTeX code:

```
As explained in
Section~\ref{subsec:ex} in
\href{{http://hplgit.github.io/doconce/...}}{Langtangen, 2012}
\cite{Doconce:test}, Doconce documents may include tables.
```

In a format different from `latex` and `pdflatex`, the effective Doconce text becomes

```
As explained in
a "section":
"http://hplgit.github.io/doconce/test/demo_testdoc.html#subsec:ex" in
the document "A Document for Testing Doconce":
"http://hplgit.github.io/doconce/test/demo_testdoc.html"
cite{Doconce:test}, Doconce documents may include tables.
```

The rendered text in the current format becomes

> As explained in Section **??**in Langtangen, 2012 [2], Doconce documents may include tables.

A complete "chapter" reference may look like

```
As explained in
refch[Chapter ref{ch:testdoc}]["Langtangen, 2012":
"http://hplgit.github.io/doconce/test/demo_testdoc.html"
cite{Doconce:test}][the document
"A Document for Testing Doconce":
"http://hplgit.github.io/doconce/test/demo_testdoc.html"
cite{Doconce:test}], Doconce documents may include tables.
```

The output now, if `ch:testdoc` is not a label in the document, becomes in the `latex` and `pdflatex` case

```
As explained in
"Langtangen, 2012":
"http://hplgit.github.io/doconce/test/demo_testdoc.html"
cite{Doconce:test}, Doconce documents may include tables.
```

That is, the internal reference `Chapter ...` is omitted since it is not meaningful to refer to an external document as "Chapter". The resulting rendered text in the current format becomes

> As explained in Langtangen, 2012 [2], Doconce documents may include tables.

Note that LaTeX cannot have links to local files, so a complete URL on the form `http://...` must be used.

Tip.

Use `doconce ref_external` to get an overview of the external references in a file. Very often you want to rewrite the text to reduce the amount of external referencing. Remember then to compile your document before running `doconce ref_external` again since the command applies the compiled files to get information (`tmp_preprocess_*` or `tmp_mako_*`) if you use any of the Preprocess or Mako preprocessors.

4.14 Index

An index can be created for the `latex`, `rst`, and `sphinx` formats by the `idx` keyword, following a LaTeX-inspired syntax:

```
idx{some index entry}
idx{main entry!subentry}
idx{‘verbatim_text‘ and more}
```

The exclamation mark divides a main entry and a subentry. Backquotes surround verbatim text, which is correctly transformed in a LaTeX setting to

```
\index{verbatim\_text@\texttt{\rm\smaller verbatim\_text and more}}
```

Everything related to the index simply becomes invisible in plain text, Epytext, StructuredText, HTML, and wiki formats. Note: `idx` commands should be inserted outside paragraphs, not in between the text as this may cause some strange behaviour of reST and Sphinx formatting. As a recommended rule, index items are naturally placed right after section headings, before the text begins, while index items related to a paragraph should be placed above the paragraph one a separate line (and not in between the text or between the paragraph heading and the text body, although this works fine if LaTeX is the output format). For paragraphs with === heading, the index keywords should be placed above the heading.

The keywords in the index are automatically placed in a meta tag in `html` output such that search engines can make use of the them.

4.15 Bibliography/References

Doconce applies the software tool Publish to handle the bibliography in a document. With Publish it is easy to import BibTeX data and maintain a database in a clean, self-explanatory textual format. From the Publish format it is easy to go BibTeX and reST or straightforward Doconce typesetting (and from there to HTML, plain text, wiki formats, and so on).

Installing Publish is straightforward: just checkout the code on `bitbucket.org`, move to the `publish` directory and run `sudo python setup.py install`.

Importing your data to the Publish database. Many scientists have their bibliographic data in the BibTex format. Here we assume that you have two files, `refs1.bib` and `refs2.bib`. These can be imported to a Publish database, residing in the file `papers.pub`, by the commands

```
publish import refs1.bib
publish import refs2.bib
```

During import, Publish may ask you for accepting the name of new institutions or journals. Publish already have a database of journals and institutions/departments, but when you add new, you also get a file `venues.list` (in the current working directory) which will be used for future imports in this directory. Make sure you store `publish.pub` and `venues.list` along with your Doconce document files (e.g., add them to your version control system).

Requirements to input data.

Notice.

Note that Publish only accepts BibTeX files where the keys (author, title, etc.) are in lower case and where the data are enclosed in curly braces. You may need to edit your BibTeX files to meet this demand.

The utility `doconce fix_bibtex4publish file.bib` fixes several known issues with BibTeX files such that Publish has a better chance of accepting the entries. Run this utility first, then run Publish, respond to any requirements that Publish spits out, remove `papers.pub` if it exists, and run the import statements again.

Although references are visible as numbers only in the output, it is recommended to have apply a nice, consistent typesetting of your keys. It is suggested to use the following scheme:

```
Langtangen_2003a          # single author
Langtangen_Pedersen_2002  # two authors
Langtangen_et_al_2002     # three or more authors
```

One can add a, b, c, and so forth if several keys feature the same authors and year.

Adding new references to the database. When you get some new BIBTEX references you simply put them in a file, say `refs3.pub` and run the `publish import refs3.pub` command to update the database. You may also consider editing the `papers.pub` file directly when adding new references.

Exporting the database. Export of everything in the database to BIBTEX is done by

```
publish export mybibtexfile.bib
```

You can easily export subsets of the database, e.g., only papers associated with a particular author (the Publish manual has details on how this is done). Doconce will automatically export the database to BIBTEX if the output format is `latex` or `pdflatex`.

Referring to publications. We use the command

```
cite{key}
```

to refer to a publication with bibliographic key `key`. Here is an example: [6] discussed propagation of large destructive water waves, [5] gave an overview of numerical methods for solving the Navier-Stokes equations, while the use of Backward Kolmogorov equations for analyzing random vibrations was investigated in [4]. The book chapter [7] contains information on C++ software tools for programming multigrid methods. A real retro reference is [3] about a big FORTRAN package. Multiple references are also possible, e.g., see [6, 7].

A LATEX-style cite command with additional detailed reference is also possible,

```
cite[details]{key}
```

for example as in [5, Section 2] or [5, Equation (4.2)].

In LATEX, the `cite` command is directly translated to the corresponding LATEX version of the command with a backslash; in reST and Sphinx the citations becomes links, with the citation keys as names; in HTML the citations are numbered from 1, 2, and so forth according to their appearance, and the numbers appear as links; while in other formats the citations are simply the keys inside square brackets and the corresponding references are listed in the order they are cited.

Specifying the Publish database. The specification of the Publish database file in the Doconce document is done one a line containing `BIBFILE: papers.pub` (you may give the database file another name and store it in another directory). The references will be inserted at the place where this command appears. Before the command you will often want to have a headline with "References", "Bibliography", or similar. Here is an example:

```
======= References =======

BIBFILE: papers.pub
```

In LaTeX and PDFLaTeX the `papers.pub` file is exported to BibTeX format and included in the document, while in all other formats, suitable text is produced from the database.

LaTeX bibliography style. The bibliography style is "plain" in LaTeX output. To change this, just edit the `.p.tex` file. For example,

```
doconce format latex mydoc
doconce replace 'bibliographystyle{plain}' 'bibliographystyle{abbrev}' mydoc.p.tex
```

4.16 Tables

A table like

time	velocity	acceleration
0.0	1.4186	-5.01
2.0	1.376512	11.919
4.0	1.1E+1	14.717624

is built up of pipe symbols and dashes:

```
|--------------------------------|
|time  | velocity | acceleration |
|--------------------------------|
| 0.0  | 1.4186   | -5.01        |
| 2.0  | 1.376512 | 11.919       |
| 4.0  | 1.1E+1   | 14.717624    |
|--------------------------------|
```

The pipes and column values do not need to be aligned (but why write the Doconce source in an ugly way?). In the line below the heading, one can insert the characters `c`, `r`, or `l` to specify the alignment of the columns (centered, right, or left, respectively). Similar character can be inserted in the line above the header to algn the headings. Pipes `|` can also be inserted to indicate vertical rules in LaTeX tables (they are ignored for other formats). An example of centered headings (which is default anyway), first column left-adjusted and the others right-adjusted looks like

```
|--c--------c-----------c--------|
|time  | velocity | acceleration |
|--l--------r-----------r--------|
| 0.0  | 1.4186   | -5.01        |
| 2.0  | 1.376512 | 11.919       |
| 4.0  | 1.1E+1   | 14.717624    |
|--------------------------------|
```

Note that not all formats offer alignment of heading or entries in tables (`rst` and `sphinx` are examples). Also note that Doconce tables are very simple: neither entries nor headings can span several columns or rows. When that functionality is needed, one can make use of the preprocessor and if-tests on the format and insert format-specific code for tables.

The command-line option `--tables2csv` (to `doconce format`) makes Doconce dump each table to CSV format in a file `table_X.csv`, where `X` is the table number. This feature makes it easy to load tables into spreadsheet programs for further analysis.

Data in CSV format can be transformed to Doconce table format by the `doconce csv2table` utility:

```
Terminal> doconce csv2table somefile.csv > table.do.txt
```

This is a quick way of writing tables. For example, we can write a text file `tmp.csv` with

```
time, velocity, acceleration
0.0, 1.4186, -5.01
2.0, 1.376512, 11.919
4.0, 1.1E+1, 14.717624
```

Running `doconce csv2table tmp.csv` creates the table

```
|------c--------------c--------------c-------|
| time         | velocity     | acceleration |
|------c--------------c--------------c-------|
| 0.0          | 1.4186       | -5.01        |
| 2.0          | 1.376512     | 11.919       |
| 4.0          | 1.1E+1       | 14.717624    |
|-----------------------------------------------|
```

4.17 Exercises, Problems, Projects, and Examples

Doconce has special support for four types of "exercises", named *exercise*, *problem*, *project*, or *example*. These are all typeset as special kind of sections. Such sections start with a subsection headline, 5 = characters, and last up to the next headline or the end of the file. The headline itself must consists of the word "Exercise", "Problem", "Project", or "Example", followed by a colon and a title of the exercise, problem, or project. The next line(s) may contain a label and specification of the name of result file (if the answer to the exercise is to be handed in) and a solution file. The Doconce code looks like this:

```
===== Project: Determine the Distance to the Moon =====
label{proj:moondist}
file=earth2moon.pdf
solution=eart2moon_sol.do.txt

Here goes the running text of the project....
```

Doconce will recognize the exercise, problem, project, or example *title*, the *label*, the *result file*, the *solution* (if any of these three entities is present), and the *running text*. In addition, one can add subexercise environments, starting with `!bsubex` and ending with `!esubex`, on the beginning of separate lines. Within the main exercise or a subexercise, three other environments are possible: (full) solution, (short) *answer*, and *hints*. The environments have begin-end directives `!bans`, `!eans`, `!bsol`, `!esol`, `!bhint`, `!ehint`, which all must appear on the beginning of a separate line (just as `!bc` and `!ec`).

The solution environment allows inline solution as an alternative to the `solution=...` directive mentioned above, which requires that the solution is in a separate file. Comment lines are inserted so that the beginning and end of answers and solutions can be identified and removed if desired.

A full exercise set-up can be sketched as follows:

```
===== Exercise: Determine the Distance to the Moon =====
label{exer:moondist}
file=earth2moon.pdf

Here goes main body of text describing the exercise...

!bsubex
Subexercises are numbered a), b), etc.

!bans
Short answer to subexercise a).
!eans

!bhint
First hint to subexercise a).
!ehint

!bhint
Second hint to subexercise a).
!ehint
!esubex

!bsubex
Here goes the text for subexercise b).

!bhint
A hint for this subexercise.
!ehint

!bsol
Here goes the solution of this subexercise.
!esol
!esubex

!bremarks
At the very end of the exercise it may be appropriate to summarize
and give some perspectives. The text inside the !bremarks-!eremarks
directives is always typeset at the end of the exercise.
!eremarks

!bsol
Here goes a full solution of the whole exercise.
!esol
!ec
```

```
A recommended rule for using the different "exercise" types goes as follows:

  * Exercises are smaller problems directly related to the present chapter
    (e.g., with references to the text).
  * Problems are sufficiently independent of the chapter's text
    that they make sense on their own, separated from the rest of the docoment.
  * Projects are larger problems that also make sense on their own.
  * Examples are exercises, problems, or projects with full solutions.

The command line options `--without_answers` and `--without_solutions`
turn off output of answers and solutions, respectively, except for
examples.

Sometimes one does not want the heading of an exercise, problem, project,
or example to contain the keyword `Exercise:`, `Problem:`, `Project:`,
or `Example:`. By enclosing the keyword in braces, as in

\begin{Verbatim}[numbers=none,fontsize=\fontsize{9pt}{9pt},baselinestretch=0.95,xleftmargin=0mm]
===== {Problem}: Find a solution to a problem =====
```

the keyword is marked for being left out of the heading, resulting in the heading "Find a solution to a problem".

The various elements of exercises are collected in a special data structure (list of dictionaries) stored in a file `.mydoc.exerinfo`, if `mydoc.do.txt` is the name of the Doconce file. The file contains a list of dictionaries, where keys in the dictionary corresponds to elements in the exercise: filename, solution file, answer, label, list of hints, list of subexercises, closing remarks, and the main body of text. From this data structure it is easy to generate stand-alone documents with exercises, problems, and projects with or without short answers and full solutions.

Tailored formatting of exercises in special output formats can make use of the elements in an exercise. For example, one can image web formats where the hints are displayed one by one when needed and where the result file can be uploaded. One can also think of mechanisms for downloading the solution file if the result file meets certain criteria. Doconce does not yet generate such functionality in any output format, but this is an intended future feature to be impelemented.

For now, exercises, problems, projects, examples are typeset as ordinary Doconce sections (this is the most general approach that will work for many formats). One must therefore refer to an exercise, problem, project, or example by its label, which normally will translate to the section number (in LaTeX, for instance) or a link to the title of the section. The *title* is typeset without any leading "Exercise:", "Problem:", or "Project:" word, so that references like

```
see Problem ref{...}
```

works well in all formats (i.e., no double "Problem Problem" appears).

Remark. Examples are *not* typeset similarly to exercises unless one adds the command-line option `--examples_as_exercises`. That is, without this option, any heading and starting with `Example:` makes Doconce treat the forthcoming text as ordinary text without any interpretation of exercise-style instructions.

With the command-line option `--examples_as_exercises`, one can use the `!bsubex` and `!bsol` commands to indicate a subproblem and a solution. In this way, the typesetting of the example looks like an exercise equipped with a solution.

List of Exercises, Problems, and Projects. Doconce also supports listing all exercises, problems, and projects with corresponding page numbers. By default, no such listing is enabled. When running `doconce ptex2tex` or `ptex2tex`, there is a preprocessor variable `LIST_OF_EXERCISES` that can be set to

- `toc`: include exercises, problems, and projects as part of the table of contents
- `loe`: make a separate list of exercises, problems, and projects, which appears after the table of contents

4.18 Blocks of Verbatim Computer Code

Blocks of computer code, to be typeset verbatim, must appear inside a "begin code" `!bc` keyword and an "end code" `!ec` keyword. Both keywords must be on a single line and *start at the beginning of the line*. Before such a code block there must be a plain sentence (at least if successful transformation to reST and ASCII-type formats is desired). For example, a code block cannot come directly after a section/paragraph heading or a table.

Here is a plain code block:

```
!bc
% Could be a comment line in some file
% And some data
1.003 1.025
2.204 1.730
3.001 1.198
!ec
```

which gets rendered as

```
% Could be a comment line in some file
% And some data
1.003 1.025
2.204 1.730
3.001 1.198
```

There may be an argument after the `!bc` tag to specify a certain environment (for `ptex2tex`, `doconce ptex2tex`, or Sphinx) for typesetting the verbatim code. For instance, `!bc dat` corresponds to the data file environment and `!bc cod` is typically used for a code snippet. There are some predefined environments explained below. If there is no argument specifying the environment, one assumes some plain verbatim typesetting (for `ptex2tex` this means the `ccq` environment, which is defined in the config file `.ptex2tex.cfg`, while for Sphinx it defaults to the `python` environment).

Since the config file for `ptex2tex` and command-line arguments for the alternative `doconce ptex2tex` program can define what some environments map onto with respect to typesetting, a similar possibility is supported for Sphinx as well. The argument after `!bc` is in case of Sphinx output mapped onto a valid Pygments language for typesetting of the verbatim block by Pygments. This mapping takes place in an optional comment to be inserted in the Doconce source file, e.g.,

```
# sphinx code-blocks: pycod=python cod=fortran cppcod=c++ sys=console
```

Here, three arguments are defined: `pycod` for Python code, `cod` also for Python code, `cppcod` for C++ code, and `sys` for terminal sessions. The same arguments would be defined in `.ptex2tex.cfg` for how to typeset the blocks in LaTeX using various verbatim styles (Pygments can also be used in a LaTeX context).

By default, `pro` is used for complete programs in Python, `cod` is for a code snippet in Python, while `xcod` and `xpro` implies computer language specific typesetting where `x` can be `f` for Fortran, `c` for C, `cpp` for C++, `sh` for Unix shells, `pl` for Perl, `m` for Matlab, `cy` for Cython, and `py` for Python. The argument `sys` means by default `console` for Sphinx and `CodeTerminal` (ptex2tex environent) for LaTeX. Other specifications are `dat` for a data file or print out, and `ipy` for interactive Python sessions (the latter does not introduce any environment in `sphinx` output, as interactive sessions are automatically recognized and handled). All these definitions of the arguments after `!bc` can be redefined in the `.ptex2tex.cfg` configuration file for ptex2tex/LaTeX and in the `sphinx code-blocks` comments for Sphinx. Support for other languages is easily added.

The enclosing `!ec` tag of verbatim computer code blocks must be followed by a newline. A common error in list environments is to forget to indent the plain text surrounding the code blocks. In general, we recommend to use paragraph headings instead of list items in combination with code blocks (it usually looks better, and some common errors are naturally avoided).

Here is a verbatim code block with Python code (`pycod` style):

```
!bc pycod
# regular expressions for inline tags:
inline_tag_begin = r'(?P<begin>(^|\s+))'
inline_tag_end = r'(?P<end>[.,?!;:)\s])'
INLINE_TAGS = {
    'emphasize':
    r'%s\*(?P<subst>[^ `][^*`]*)\*%s' % \
    (inline_tag_begin, inline_tag_end),
    'verbatim':
    r'%s`(?P<subst>[^ ][^`]*)`%s' % \
    (inline_tag_begin, inline_tag_end),
    'bold':
    r'%s_(?P<subst>[^ `][^_`]*)_%s' % \
    (inline_tag_begin, inline_tag_end),
}
!ec
```

The typeset result of this block becomes

```
# regular expressions for inline tags:
inline_tag_begin = r'(?P<begin>(^|\s+))'
```

```
inline_tag_end = r'(?P<end>[.,?!;:)\s])'
INLINE_TAGS = {
    'emphasize':
    r'%s\*(?P<subst>[^ `][^*`]*)\*%s' % \
    (inline_tag_begin, inline_tag_end),
    'verbatim':
    r'%s`(?P<subst>[^ ][^`]*)`%s' % \
    (inline_tag_begin, inline_tag_end),
    'bold':
    r'%s_(?P<subst>[^ `][^_`]*)_%s' % \
    (inline_tag_begin, inline_tag_end),
}
```

And here is a C++ code snippet (`cppcod` style):

```
void myfunc(double* x, const double& myarr) {
    for (int i = 1; i < myarr.size(); i++) {
        myarr[i] = myarr[i] - x[i]*myarr[i-1]
    }
}
```

Computer code can be copied directly from a file, if desired. The syntax is then

```
@@@CODE myfile.f
@@@CODE myfile.f fromto: subroutine\s+test@^C\s{5}END1
```

The first line implies that all lines in the file `myfile.f` are copied into a verbatim block, typset in a `!bc Xpro` environment, where `X` is the extension of the filename, here `f` (i.e., the environment becomes `!bc fpro` and will typically lead to some Fortran-style formatting). The second line has a `fromto:` directive, which implies copying code between two lines in the code, typset within a ¡bc Xcod‘ environment (again, `X` is the filename extension, implying the type of file). Note that the `pro` and `cod` arguments are only used for LaTeX and Sphinx output, all other formats will have the code typeset within a plain `!bc` environment.) Two regular expressions, separated by the `@` sign, define the "from" and "to" lines. The "from" line is included in the verbatim block, while the "to" line is not. In the example above, we copy code from the line matching `subroutine test` (with as many blanks as desired between the two words) and the line matching `C END1` (C followed by 5 blanks and then the text END1). The final line with the "to" text is not included in the verbatim block.

One can also specify the code environment explictly rather than relying on the file extension:

```
@@@CODE somefile.py envir=X fromto: def myfunc@def yourfunc
```

Let us copy a whole file (the first line above):

```
C     a comment

      subroutine test()
      integer i
      real*8 r
      r = 0
      do i = 1, i
         r = r + i
```

```
      end do
      return
C     END1

      program testme
      call test()
      return
```

Let us then copy just a piece in the middle as indicated by the `fromto:` directive above:

```
      subroutine test()
      integer i
      real*8 r
      r = 0
      do i = 1, i
         r = r + i
      end do
      return
```

Note that the "to" line is not copied into the Doconce file, but the "from" line is. Sometimes it is convenient to also neglect the "from" line, a feature that is allowed by replacing `fromto:` by `from-to` ("from with minus"). This allows for copying very similar code segments throughout a file, while still distinguishing between them. Copying the second set of parameters from the text

```
# --- Start Example 1 ---
c = -1
A = 2
p0 = 4
simulate_and_plot(c, A, p0)
# --- End Example 1 ---

# --- Start Example 2 ---
c = -1
A = 1
p0 = 0
simulate_and_plot(c, A, p0)
# --- End Example 2 ---
```

is easy with

```
from-to: Start Example 2@End Example 2
```

With only `fromto:` this would be impossible.

Remark for those familiar with `ptex2tex`: The from-to syntax is slightly different from that used in `ptex2tex`. When transforming Doconce to LaTeX, one first transforms the document to a `.p.tex` file to be treated by `ptex2tex`. However, the `@@@CODE` line is interpreted by Doconce and replaced by the mentioned pro or cod environment which are defined in the `ptex2tex` configuration file.

Remark for those familiar with the `listings` package in LaTeX: the `listing` package can copy code from files, but snippets must be specified through exact line numbers. The `@@@CODE` directive above works with regular expressions which are much less sensitive to edits of the source code file than the line numbers. Moreover, copy of code from file works in Dococe across formats (HTML, Sphinx, Markdown, etc.).

4.19 LaTeX Blocks of Mathematical Text

Blocks of mathematical text are like computer code blocks, but the opening tag is `!bt` (begin TeX) and the closing tag is `!et`. It is important that `!bt` and `!et` appear on the beginning of the line and followed by a newline.

```
!bt
\begin{align}
{\partial u\over\partial t} &= \nabla^2 u + f, label{myeq1}\\
{\partial v\over\partial t} &= \nabla\cdot(q(u)\nabla v) + g. label{myeq2}
\end{align}
!et
```

The support of LaTeX mathematics varies among the formats:

- Output in LaTeX (`latex` and `pdflatex` formats) has of course full support of all LaTeX mathematics, of course.
- The `html` format supports single equations and multiple equations via the align environment, also with labels.
- Markdown (`pandoc` format) allows single equations and inline mathematics.
- MediaWiki (`mwiki` format) does not enable labels in equations and hence equations cannot be referred to.

The main conclusion is that for output beyond LaTeX (`latex` and `pdflatex` formats), stick to simple `\[` and `\]` or `equation` and `align` or `align*` environments, and avoid referring to equations in MediaWikis.

Going from Doconce to MS Word is most easily done by outputting in the `latex` format and then using the Pandoc program to translate from LaTeX to MS Word (note that only a subset of LaTeX will be translated correctly).

If the document targets formats with and without support of LaTeX mathematics, one can use the preprocessor to typeset the mathematics in two versions. After `#if FORMAT in ("latex", "pdflatex", "html", "sphinx", "mwiki", "pandoc")` one places LaTeX mathematics, and after `#else` one can write inline mathematics in a way that looks nice in plain text and wiki formats without support for mathematical typesetting. Such branching can be used with mako if-else statements alternatively:

```
% if FORMAT in ("latex", "pdflatex", "html", "sphinx", "mwiki", "pandoc"):
!bt
\[ \sin^2x + \cos^2x = 1,\]
!et
% else:
!bc
              sin^2(x) + cos^2(x) = 1,
!ec
% endif
```

Mathematics for PowerPoint/OpenOffice. If you have LaTeX mathematics written in Doconce, it is fairly easy to generate PNG images of all mathematical formulas and equations for use with PowerPoint or OpenOffice presentations.

1. Make a Sphinx version of the Doconce file.
2. Go to the Sphinx directory and load the `conf.py` file into a browser.
3. Search for "math" and comment out the `'sphinx.ext.mathjax'` (enabled by default) and `'matplotlib.sphinxext.mathmpl'` (disabled by default) lines, and uncomment the `'sphinx.ext.pngmath'` package. This is the package that generates small PNG pictures of the mathematics.
4. Uncomment the line with `pngmath_dvipng_args =` and set the PNG resolution to `-D 200` when the purpose is to generate mathematics pictures for slides.
5. Run `make html`.
6. Look at the HTML source file in the `_build/html` directory: all mathematics are in `img` tags with `src=` pointing to a PNG file and `alt=` pointing to the LaTeX source for the formula in question. This makes it very easy to find the PNG file that corresponding to a particular mathematical expression.

4.20 Macros (Newcommands)

Doconce supports a type of macros via a LaTeX-style *newcommand* construction. The newcommands defined in a file with name `newcommand_replace.tex` are expanded when Doconce is filtered to other formats, except for LaTeX (since LaTeX performs the expansion itself). Newcommands in files with names `newcommands.tex` and `newcommands_keep.tex` are kept unaltered when Doconce text is filtered to other formats, except for the Sphinx format. Since Sphinx understands LaTeX math, but not newcommands if the Sphinx output is HTML, it makes most sense to expand all newcommands. Normally, a user will put all newcommands that appear in math blocks surrounded by `!bt` and `!et` in `newcommands_keep.tex` to keep them unchanged, at least if they contribute to make the raw LaTeX math text easier to read in the formats that cannot render LaTeX. Newcommands used elsewhere throughout the text will usually be placed in `newcommands_replace.tex` and expanded by Doconce. The definitions of newcommands in the `newcommands*.tex` files *must* appear on a single line (multi-line newcommands are too hard to parse with regular expressions).

4.21 Admonitions

Doconce offers strong support for admonition environments, such as warning boxes, notification boxes, question boxes, and summary boxes. The boxes

normally have an icon, a heading, and may also have a background color. A special box, the block, has never any icon and can be used when an icon would be disturbing or misleading.

The following admonition environments are available: `block`, `warning`, `notice`, `question`, and `summary`. The box is defined by begin and end tags such as `!bnotice` and `!enotice`. The title of the box is fully customizable.

Here are a few examples:

```
!bwarning
Here is a warning!
!ewarning

!bnotice Hint
This is a hint.
!enotice

!bblock This is a block.
A block has never any icon. It is mostly used in slides
and give the text block effect in LaTeX beamer.
!eblock

!bnotice Going deeper
This is text meant to provide more details. The box has the
layout of the notice box, but a custom title, here "Going deeper".
!enotice

Finally some summary:

!bsummary
The main message is to utilize the admonition styles for
marking different parts of the text
!esummary
```

The above Doconce code is in the present format rendered as

Warning.

Here is a warning!

Hint.

This is a hint.

This is a block.

A block has never any icon.

Going deeper.

This is text meant to provide more details. The box has the layout of the notice box, but a custom title, here "Going deeper".

Finally some summary:

Summary.

The main message is to utilize the admonition styles for marking different parts of the text

The layout of admonitions depend on the format. In `rst` and `sphinx` one applies the native admonitions, but in `sphinx` the `automake_sphinx.py` script manipulates the HTML file to set a gray background for all admonitions. In `html` one has a command-line argument `--html_admon` that can be set to different styles:

- `--html_admon=gray` for icons with gray background and small font,
- `--html_admon=yellow` and `--html_admon=apricot` are similar, but the icons and colors are different,
- `--html_admon=colors` has quite bright colors as backgrounds for the different admonitions,
- `--html_admon=lyx` gives a white background and small icons,
- `--html_admon=paragraph` results in a simple paragraph with the admon title as heading.

The options `--html_admon_bg_color=...` and `--html_admon_bd_color=...` can be used to override the default background and boundary frame colors of the admon styles (respectively). These options have only effect for the `apricot`, `yellow`, and `gray` styles.

Some recommended combinations for admonitions in HTML are

- `--html_style=solarized`, `--html_admon=apricot`, `--pygments_html_style=perldoc`
- `--html_style=blueish2`, `--html_admon=yellow`, `--no_pygments_html`
- `--html_style=blueish2`, `--html_admon=yellow`, `--pygments_html_style=default`
- `--html_style=bloodish`, `--html_admon=gray`, `--no_pygments_html`
- `--html_style=bloodish`, `--html_admon=gray`, `--pygments_html_style=default`
- `--html_style=vagrant`, `--pygments_html_style=default`, `--html_template=...`

The `vagrant` HTML style has CSS files that override the definition how the admons are typset. The `notice` environment is gray with an icon (defined in `vagrant.css`), while the others are yellow (defined in `twitter_bootstrap.css`). The `--html_admon` color has no effect for the `vagrant` style.

In `latex` and `pdflatex`, the type of admonition style is set by the command-line option `--latex_admon=`. Several values are available:

- `paragraph` is the simplest type of admonition and typeset as plain text with an optional paragraph heading.
- `colors1` (inspired by the NumPy User Guide) applies different colors for the different admons with an embedded icon.
- `colors2` is like `colors1` but the text is wrapped around the icon.
- `mdfbox` is the default and gives rounded (gray) boxes with a potential title and no icon (using the very flexible `mdframed` packaged in LaTeX)
- `graybox2` has square corners, gray background, and is narrower than `mdfbox`. One special feature of `graybox2` is the summary admon, which has a different look with horizontal rules only, and for A4 format, the summary box is half of the text width and wrapped with running text around (if it does not contain verbatim text, in that case the standard `graybox2` style is used). This small summary box is effective in proposals to disperse small paragraphs of key points around.
- `grayicon` has icons and a light gray background.
- `yellowicon` has icons and a light yellow background.

There is also an option `--latex_admon_color=...` that can be used to override the default color. Values are either saturated colors like `gray!10` or an RGB tuple `0.95,0.91,0.97`. The chosen color replaces all default colors for all admon styles except `paragraph`. For example, an oval white box is produced by `--latex_admon=mdfbox` and `--latex_admon_color=white`.

The Box Environment. A plain box without any title or colored background, just a rectangular frame, is also available. The syntax goes like

```
!bbox
!bt
\[ \nabla\cdot\bm{u} = 0  \quad\hbox{(mass balance)}\]
!et
!ebox
```

resulting in

$$\nabla \cdot \boldsymbol{u} = 0 \quad \text{(mass balance)}$$

Admonitions are usually used to typeset something that should be distinct from the running text: a warning, a remark, a notification, a question, a summary, while a box is often used to highlight a key finding in the running text: an equation, a hypothesis, a theorem, a rule, or a conclusion.

4.22 Typesetting of Algorithms

Doconce has no support for typesetting of algorithms, while LaTeX has quite sophisticated support. LaTeX writes need some strategy with dealing with algorithms in Doconce: they should be sophisticated when the output is in LaTeX and they should be readable when the output is in other formats that have no native support for algorithms.

The simplest solution is to use the preprocessor (Preprocess or Mako) to allow different solutions for different formats. There are basically three typesetting styles available: LaTeX algorithm environments, pseudo code in a verbatim block, and a nested list. A preprocessor variable, say `ALG` can be used to select the typesetting. With Mako we can write

```
% if ALG == 'latex':
# Write native LaTeX code for the algorithm
% elif ALG == 'code':
# Write pseudo code in (e.g.) a python-like style
!bc pycode
if feature in element:
...
!ec
% elif ALG == 'list':
# Use lists to express the algorithm
  o if this element has the feature:
    * ...
% endif
```

4.23 Preprocessing Steps

Doconce allows preprocessor commands for, e.g., including files, leaving out text, or inserting special text depending on the format. Two preprocessors are supported: preprocess (http://code.google.com/p/preprocess) and mako (http://www.makotemplates.org/). The former allows include and if-else statements much like the well-known preprocessor in C and C++ (but it does not allow sophisticated macro substitutions). The latter preprocessor is a very powerful template system. With Mako you can automatically generate various type of text and steer the generation through Python code embedded in the Doconce document. An arbitrary set of `name=value` command-line arguments (at the end of the command line) automatically define Mako variables that are substituted in the document.

Doconce will detect if preprocess or Mako commands are used and run the relevant preprocessor prior to translating the Doconce source to a specific format.

The preprocess and mako programs always have the variable FORMAT defined as the desired output format of Doconce (html, latex, plain, rst, sphinx, epydoc, st). It is then easy to test on the value of FORMAT and take different actions for different formats. Below is an example:

```
First some math:

!bt
\begin{align}
x &= 3
label{x:eq1}\\
y &= 5
label{y:eq1}
\end{align}
!et
Let us now reason about this.

# Sphinx cannot refer to labels in align environments

# #if FORMAT in ("latex", "pdflatex", "html")
From (\ref{x:eq})-(\ref{y:eq1}) we get that
# #elif FORMAT == "sphinx"
From
!bt
\[ x = 3 \]
!et
and
!bt
\[ y= 5 \]
!et
it follows that
# #else
From the above equations it follows that
# #endif
$x+y$ is 8.
```

A variable DEVICE is also defined. It equals screen by default, but the command-line argument --device=paper can set DEVICE to paper (or another value). Testing on DEVICE inside the document makes it possible to test if the output is on paper media, a sreen, or a particular device.

Other user-defined variables for the preprocessor can be set at the command line as explained in Section 3.

More advanced use of mako can include Python code that may automate the writing of parts of the document.

4.24 Splitting Documents into Smaller Pieces

Long documents are conveniently split into smaller Doconce files. However, there must be a master document including all the pieces, otherwise references to sections and the index will not work properly. The master document is preferably a file just containing a set of preprocessor include statements of the form `#include "file.do.txt"`. The preprocessor will put together all the pieces so that Doconce sees a long file with the complete text.

For web documents it is often desired to split long pages into shorter ones. This is done by the Doconce command `!split` placed at the beginning of a line. The `!split` commands works with output in `html`, `rst`, `sphinx`, `latex`, and `pdflatex`. The `!split` command are normally placed before section headings. It is very actively used when writing slides with Doconce. The `doconce format` command does not recognize `!split` instructions: one needs to run `doconce split_*` as a postprocess, where the `*` means `html`, `rst`, or `beamer`.

HTML. Splitting an HTML document is done by

```
Terminal> doconce format html mydoc
Terminal> doconce split_html mydoc
```

The `mydoc.html` document created by the first command is replaced by a new HTML file, representing the first part of the document, after the second command. The various files that constitute the parts of the document are listed after the `split_html` command. The files have names `mydoc.html`, `._mydoc000.html` (equal to `mydoc.html`), `._mydoc001.html`, `._mydoc002.html`, and so on. Recall that all the parts are needed if the HTML document is to be moved to another location (you can always check `.mydoc_html_file_collection` for a list of all the files that are needed to display this HTML document).

MathJax cannot refer to equations defined in other HTML files, but the `doconce split_html` fixes this problem. Note, however, that running `doconce split_html` leads to another equation numbering than in the original HTML document. In the latter, we use AMS equation numbering, which means that the standard LaTeX conventions are followed, while in the splitted document only the subset of equations with labels are given numbers.

The HTML documents have very simple navigation buttons for the previous and next document. These are not customizable directly, but one can easily look up the HTML code and use `doconce replace` to edit the links to the images used for navigation. Some more colorful images arise from

```
Terminal> doconce replace 'prev1.png' 'Knob_Left.png' \
          mydoc.html ._mydoc*.html
Terminal> doconce replace 'next1.png' 'Knob_Forward.png' \
          mydoc.html ._mydoc*.html
```

This works because the `Knob*.png` images live in the same place in the Doconce repository as `prev1.png` and `next1.png`. Other images can be used by replacing the whole URL.

With an HTML template one can have much more sophisticated navigation between parts. One example is the template in `bundled/html_styles/style_vagrant/template_vagrant.html` in the Doconce source.

reStructuredText and Sphinx. Here is a typical split of a large Sphinx document `mydoc.rst` into smaller pieces:

```
Terminal> doconce sphinx_dir author="Some Author" \
          title="Short title" theme=fenics dirname=mydir mydoc
Terminal> doconce format sphinx mydoc
Terminal> doconce split_rst mydoc
Terminal> python automake_sphinx.py
```

The `doconce format sphinx mydoc` command is needed to produce `mydoc.rst`, which is the starting point for the `doconce split_rst` command. The various files that constitute the complete Sphinx document are `mydoc.rst`, `._mydoc000.rst`, `._mydoc001.rst`, `._mydoc002.rst`, and so on. The `automake_sphinx.py` script ensures that the Sphinx document is compiled correctly. If all links to local files are in a `_static` directory, the whole Sphinx document exists in a complete version in the compiled directory (usually `sphinx-rootdir/_build/html`) and can easily be moved around.

LaTeX Beamer slides generated from Doconce source also apply `!split` to indicate the start of individual slides. However, the split is performed by the `doconce slides_beamer` command and does not result in individual files like `split_rst` and `split_html` do.

4.25 Missing Features

Doconce does not aim to support sophisticated typesetting, simply because sophisticated typesetting usually depend quite strongly on the particular output format chosen. When a particular feature needed is not supported by Doconce, it is recommended to hardcode that feature for a particular format and use the if-else construction of the preprocessor. For example, if a sophisticated table is desired in LaTeX output, do something like

```
# #if FORMAT in ("latex", "pdflatex")
# insert native LaTeX code for fancy table
# #else
# insert a Doconce-formatted "inline" table
# #endif
```

Similarly, if certain adjustments are needed, like pagebreaks in LaTeX, hardcode that in the Doconce format (and recall that this is really LaTeX dependent - pagebreaks are not relevant HTML formats).

Instead of inserting special code in the Doconce document, one can alternatively script editing of the output from Doconce. That is, we develop a Python or Bash script that runs the translation of a Doconce document to a ready document in another format. Inside this script, we may edit and fine-tune the output from Doconce.

4.26 Header and Footer

Some formats use a header and footer in the document. LaTeX and HTML are two examples of such formats. When the document is to be included in another document (which is often the case with Doconce-based documents),

the header and footer are not wanted, while these are needed (at least in a LaTeX context) if the document is stand-alone. We have introduced the convention that if `TITLE:` is found at the beginning of the line (i.e., the document has a title), the header and footer are included, otherwise not.

4.27 Emacs Doconce Formatter

The file .doconce-mode.el in the Doconce source distribution gives a "Doconce Editing Mode" in Emacs.

Here is how to get the Doconce Editing Mode in Emacs: Download .doconce-mode.el and save it in your home directory, then add these lines to `~/.emacs`:

```
(load-file "~/.doconce-mode.el")
```

Emacs will now recognize files with extension `.do.txt` and enter the Doconce Editing Mode.

The major advantage with the Doconce Editing Mode in Emacs is that many keyboard shortcuts are defined:

Emacs key	Action
Ctrl+c f	figure
Ctrl+c v	movie/video
Ctrl+c h1	heading level 1 (section/h1)
Ctrl+c h2	heading level 2 (subsection/h2)
Ctrl+c h3	heading level 2 (subsection/h3)
Ctrl+c hp	heading for paragraph
Ctrl+c me	math environment: `!bt` equation `!et`
Ctrl+c ma	math environment: `!bt` align `!et`
Ctrl+c ce	code environment: `!bc` code `!ec`
Ctrl+c cf	code from file: `@@@CODE`
Ctrl+c table2	table with 2 columns
Ctrl+c table3	table with 3 columns
Ctrl+c table4	table with 4 columns
Ctrl+c exer	exercise outline
Ctrl+c slide	slide outline
Ctrl+c help	print this table

Typing `Ctrl+c help` prints the above table in Emacs. Try out the different shortcuts and see how handy they are in learning Doconce and saving much typing!

5 Writing Slides

It is a fast procedure to make slides from large amounts of Doconce text, in particular for condensing running material for teaching or just providing the slide set as an overview or study guide. The slides can either be ordinary,

separate slides - or just a document with much briefer text and emphasis on bullet lists, figures, mathematical formulas, admonitions, and little text.

Points:

- Only some pygments style are suited for a particular reveal.js/deck.js theme
- Only some admon styles are appropriate
- Admon styles are erased in reveal
- Use `--keep_pygments_html_bg` to avoid big changes in background color for code

5.1 Overview

Basically, Doconce slides are ordinary Doconce text with `!split` inserted before each slide. Nevertheless, contents of slide differ considerably from ordinary running text. Some guidelines on the elements within each slide are necessary to produce effective slide sets:

- Use a subsection heading as slide heading (5 =).
- Limit the amount of running text (as always).
- Limit the amount of material so it fits within a slide (inspect slides visually to move or delete content - just an extra `!split` and a new heading is enough to make a new slide).
- Use the `slidecell` environment (see below) to create a grid of slide cells (makes it easy to move figures and bullet lists or text around).
- Adjust the size of figures (`width` parameter for HTML, `frac` parameter for LaTeX Beamer) so they become effective on the slide.

Doconce can generate two types of slides: HTML5+CSS3 type of slides to be presented through a web browser, and classical LaTeX Beamer slides.

5.2 HTML5 Slides

Not yet written...

Just a very preliminary sketch of commands:

```
Terminal> doconce format html myslides \
          --pygments_html_style=native --keep_pygments_html_bg
Terminal> doconce slides_html myslides reveal \
          --html_slide_theme=darkgray
```

Potential Problems.

- Some newer Firefox does not show rounded corners of the admonition boxes, e.g., notice and warning (tested on Ubuntu)
- Doconce performs some adjustments of the spacing around equations. More edits (automate with `doconce subst`) might be needed.

5.3 LaTeX Beamer Slides

Not yet written...

Themes. Four themes come with Doconce: `X_Y`, where `X` is `blue` or `red` (the main color of the slides) and `Y` is `plain` for simple layout and `shadow` for shadowed boxes and more visual structure in the slides.

6 Mako Programming

The Mako templating engine is used as preprocessor for Doconce documents, but the Preprocess is run prior to Mako and is recommended for including other files via `# #include "filename"`. Preprocess is also sufficient for if-else tests to steer which parts of the text that are to be compiled. For more demanding tasks, use Mako, which resembles a real programming language.

Warning.

Unfortunately, the combination of Mako and LaTeX mathematics may lead to problems because Mako applies syntax like `${var}` to extract variables or call functions, while LaTeX mathematics sometimes applies the same syntax, e.g., `${\cal O}(\Delta x^2)$` which looks like a Mako function call. This problem can give rise to strange error messages from Mako, usually that a variable is not defined.

6.1 The Basics of Mako

Just a preliminary sketch of some Mako code (next example is better!):

```
# Define variables
<%
mycounter = 1
mydict = {}
%>

# Assume MYVAR is given on the command line as MYVAR=mytext (e.g.)
% if MYVAR is not UNDEFINED:
```

```
The value of MYVAR is ${MYVAR}.
% endif

<%
## Manipulation of variables
mycounter += 1
%>

% if MYVAR in (2,4,6):
MYVAR is even integer in [2,6].
% elif MYVAR > 1000000:
MYVAR is big.
% else:
MYVAR=${MYVAR}, mycounter=${mycounter}.
% endif

# Function
<%
# Define Python function: FORMAT and DEVICE
# are always defined

def link(filename):
    url = "https://github.com/some/path/to/' + filename + '"'
    if DEVICE == 'screen':
        # make link to url
        return '"filename":' + url
    elif DEVICE == 'paper':
        # write URL explicit on paper
        return 'URL:' + url
%>

<%doc>
This
is
a
block
comment in Mako.
<%doc/>
```

One can use Mako to extend the Doconce syntax. Two examples are given next.

6.2 Example: Extending Tables to Handle Figures

Doconce tables cannot contain figures, since figures must appear on a single line with the special syntax `FIGURE: [file, ...]`. What if you want a table of thumbnail figures with hyperlinks?

Solution for LaTeX and HTML. We first restrict the attention to LaTeX and HTML. In those cases we can create an ordinary table and insert a call to a Mako function in cells in the table to return the proper LaTeX or HTML code for displaying a figure.

The Mako call syntax in a cell is decided to be `${tfig('080')}`, which insert the figure `mov/wave_frames/frame_0080.png`. A 3x3 table of figures then look like

```
|-----------------------------------------------|
|              |               |                |
|-------c--------------c----------------c--------|
|${tfig('080')} | ${tfig('085')} | ${tfig('090')} |
|${tfig('095')} | ${tfig('100')} | ${tfig('105')} |
|${tfig('110')} | ${tfig('115')} | ${tfig('120')} |
|-----------------------------------------------|
```

We do not want a heading, just a grid of figures, but a heading is required in Doconce figures, so the solution is to provide empty column names in the heading. This solution gives acceptable results in HTML and LaTeX.

The Mako function can be a plain Python function:

```
<%
def tfig(fileno):
    p = 'mov/wave_frames/frame_0' + fileno + '.png'  # path
    if FORMAT in ("latex", "pdflatex"):
        text = r'\includegraphics[width=2cm]{%s}' % p
    elif FORMAT == "html":
        text = '<a href="%s"><img src="%s" width="300"></a>' % (p, p)
    else:
        text = '"`%s`": "%s"' % (fileno, p)  # plain link
    return text
%>
```

Note that for other formats than LaTeX and HTML we just return a link to the figure.

Notice.

A corresponding solution for Sphinx will not work because inline figures in Sphinx has a syntax with pipe symbols ('|') that interfer with the column separator in tables in Doconce. Returning the HTML code in case of Sphinx will just display that HTML code in the cells, not the rendered HTML code. A solution that includes Sphinx is provided later.

The resulting table is displayed below.

080	085	090
095	100	105
110	115	120

Generating the Entire Table. The specific structure of this table also suggest using Mako to generate the entire table:

```
<%
def generate_table(start, end, step, no_of_columns):
    # Heading
    text = ''
    horizontal_rule = '|----|\n'
```

```
    text += horizontal_rule
    text += '|  '*no_of_columns + '|\n'
    text += horizontal_rule
    fig_counter = 0
    for counter in range(start, end+1, step):
        fig_counter += 1
        text += '| ' + tfig('%03d' % counter) + ' '
        if fig_counter % no_of_columns == 0:
            text += '|\n'
    text += horizontal_rule
    return text
%>

${generate_table(80, 120, 5, 3)}
```

The result is a table like the one above, except that the horizontal rules and the heading are very short lines (but this is legal syntax - it just does not look appealing in the Doconce source).

Generating the Entire Table in a Native Format via Mako. To allow Sphinx and other formats, it is best to generate the entire table. This can be done either by a Mako function or as a post process after Doconce has created the output file. We show the Mako solution here.

Tip.

A pure Python function Mako is easiest to develop in a seperate Python program file because Python has better error messages than Mako. Also, Python has debugging facilities that make the development of the code much more efficient than writing in Mako. When the Python function works, it can be inserted in the Doconce file inside the Mako directives <% and %>.

The Mako function for generating the entire table in LaTeX, HTML, and Sphinx format goes as follows.

```
<%
def generate_native_table(start, end, step, no_of_columns):
    text = ''
    # Heading
    if FORMAT in ("latex", "pdflatex"):
        text += r"""\begin{quote}\begin{tabular}{%s}
\hline
 \\
\hline
""" % ('c'*no_of_columns)
    elif FORMAT in ("sphinx", "rst", "html"):
        text += '<p><table border="1">\n<tr>'
    fig_counter = 0
    latex_columns = []
    for counter in range(start, end+1, step):
        fig_counter += 1
        if FORMAT in ("latex", "pdflatex"):
            latex_columns.append(r'\includegraphics[width=2cm]'\
```

```
                '{mov/wave_frames/frame_%04d.png}\n' % counter)
        elif FORMAT in ("sphinx", "rst", "html"):
            text += '<td align="center"> '\
            '<a href="mov/wave_frames/frame_%04d.png">'\
            '<img src="mov/wave_frames/frame_%04d.png" width="300">'\
            '</a> </td>\n' % (counter, counter)

        if fig_counter % no_of_columns == 0:  # new row
            if FORMAT in ("latex", "pdflatex"):
                text += '   ' + ' & '.join(latex_columns)
                if counter != end:
                    text += r'\\' + '\n'
                latex_columns = []
            elif FORMAT in ("sphinx", "rst", "html"):
                text += '</tr>\n'
                if counter != end:
                    text += '<tr>\n'  # begin new row
    # Footer
    if FORMAT in ("latex", "pdflatex"):
        text += r"""\hline
\end{tabular}\end{quote}
"""
    elif FORMAT in ("sphinx", "rst", "html"):
        text += '</table>\n'
    if FORMAT in ("sphinx", "rst"):
        # Wrap raw HTML code
        lines = text.splitlines()
        text = '\n.. raw:: html\n\n'
        for line in lines:
            text += '    ' + line + '\n'
        text += '\n'
    return text
%>

## Example on call in Doconce source
${generate_native_table(80, 120, 5, 3)}
```

One can easily add support for various other formats, such as `pandoc`, `gwiki`, `mwiki`, etc. The output in format the current format is as above.

Generating the Entire Table in a Native Format as a Post Process. Instead of using Mako as shown above, we can invent our own syntax line for the table with figures and apply a script to the Doconce output file to replace the special line with proper native code. We use the same `generate_native_table` function as above, but this time in a Python script. In the Doconce source we place a line

```
@@@FIGTABLE 80 120 5 3
```

which is meant to generate a table with figures where the numbers correspond to arguments in the `generative_native_table` function.

Suppose we have run

```
Terminal> doconce format sphinx mydoc
```

In the resulting `mydoc.rst` file we have the magic line `@@@FIGTABLE ...`. Such lines can now be processed in our Python script:

```
# read mydoc.rst into the string filestr
pattern = r'@@@FIGTABLE (.+)'
def replacement(m):
    # m is a MatchObject
    args = m.group(1)
    # Convert args to a tuple of words with right type
    args = [eval(arg) for arg in args.split()]
    text = generate_native_table(*args)

filestr = re.sub(pattern, filestr, replacement)
# write filestr back to manual.rst
```

This code makes use of a function for creating the replacement string for the `re.sub` command and demands a knowledge of regular expression syntax and the workings or `re.sub`. A more plain Python version goes like this:

```
# read mydoc.rst into the string filestr
lines = filestr.splitsplines()
for i in range(len(lines)):
    if lines[i].startswith('@@@FIGTABLE'):
        args = lines[i][10:].strip()
        # Convert args to a tuple of words with right type
        args = [eval(arg) for arg in args.split()]
        text = generate_native_table(*args)
        lines[i] = text
filestr = '\n'.join(lines)  # make one string from all separate lines
# write filestr back to manual.rst
```

Recommendation.

This latter way of inserting specialized native text after Doconce has generated the output file is easier than using Mako and usually also safer.

Lesson Learned.

The above examples show different techniques for extending the Doconce capabilities. The ideas are not restricted to Doconce: using Mako as a preprocessing step or inserting "magic" lines that you can later substitute by a script is possible in all types of ASCII-based documents, e.g., LaTeX, Sphinx, and Markdown.

6.3 Example: Defining a Theorem Environment

Doconce supports only basic formatting elements (headings, paragraphs, lists, etc.), while LaTeX users are used to fancy environments for, e.g., theorems. A flexible strategy is to typeset theorems using paragraph headings, which will look satisfactorily in all formats, but add comment lines that can be replaced by LaTeX environments via `doconce replace`. Theorems can be numbered using a variable in Mako. Here is an example on raw Doconce code:

```
<%
theorem_counter = 4
%>

# begin theorem
label{theorem:fundamental1}
<%
theorem_counter += 1
theorem_fundamental1 = theorem_counter
%>

__Theorem ${theorem_counter}.__
Let $a=1$ and $b=2$. Then $c=3$.
# end theorem

# begin proof
__Proof.__
Since $c=a+b$, the result follows from straightforward addition.
$\Diamond$|$END$
# end proof

As we see, the proof of Theorem ${theorem_counter} is a modest
achievement.
```

The `.p.tex` output file now reads

```
% begin theorem
label{theorem:fundamental1}


\paragraph{Theorem 5.}
Let $a=1$ and $b=2$. Then $c=3$.
% end theorem

% begin proof
\paragraph{Proof.}
Since $c=a+b$, the result follows from straightforward addition.
$\Diamond$
% end proof

As we see, the proof of Theorem 5 is a modest
achievement.
```

Note that with Mako variables we can easily create our own counters, and this works in any format. In LaTeX we can use both the generated numbers from Mako variables or we can use the labels.

The next step is to replace the `% begin ...` and `% end ...` lines with the proper LaTeX expressions in the `.p.tex` file. Moreover, we need to remove the paragraphs with *Theorem*. The following Bash script does the job:

```
file=mydoc.p.tex
thpack='\\usepackage{theorem}\n\\newtheorem{theorem}{Theorem}[section]'
doconce subst '% insert custom LaTeX commands\.\.\.' $thpack $file
doconce subst '\\paragraph\{Theorem \d+\.\}' '' $file
doconce replace '% begin theorem' '\begin{theorem}' $file
doconce replace '% end theorem' '\end{theorem}' $file
```

More heavy editing is better done in a Python script that reads the `mydoc.p.tex` file and performs string substitutions and regex substitutions as needed.

The resulting `mydoc.tex` file now becomes

```
\usepackage{theorem}
\newtheorem{theorem}{Theorem}[section]

...

\begin{theorem}
\label{theorem:fundamental1}



Let $a=1$ and $b=2$. Then $c=3$.
\end{theorem}

% begin proof
\paragraph{Proof.}
Since $c=a+b$, the result follows from straightforward addition.
$\Diamond$
% end proof

As we see, the proof of Theorem 5 is a modest
achievement.
```

Even better, HTML output looks nice as well.

Note that Doconce supports fancy environments for verbatim code via the `ptex2tex` program with all its flexibility for choosing environments. Also `doconce ptex2tex` has quite some flexibility for typesetting computer code.

6.4 Tools for Writing Doconce Documents

- Emacs (with the modest Doconce syntax highlighting), Vim, Gedit, etc.
- TeXMaker and Kile can in theory be adapted to handle Doconce compilation and maybe even Doconce constructs.

6.5 Debugging

Given a problem, extract a small portion of text surrounding the problematic area and debug that small piece of text. Doconce does a series of transformations of the text. The effect of each of these transformation steps are dumped to a logfile, named `_doconce_debugging.log`, if the to `doconce format` after the filename is `debug`. The logfile is inteded for the developers of Doconce, but may still give some idea of what is wrong. The section "Basic Parsing Ideas" explains how the Doconce text is transformed into a specific format, and you need to know these steps to make use of the logfile.

7 Basic Parsing Ideas

The (parts of) files with computer code to be directly included in the document are first copied into verbatim blocks.

All verbatim and TeX blocks are removed and stored elsewhere to ensure that no formatting rules are not applied to these blocks.

The text is examined line by line for typesetting of lists, as well as handling of blank lines and comment lines. List parsing needs some awareness of the context. Each line is interpreted by a regular expression

```
(?P<indent> *(?P<listtype>[*o-] )? *)(?P<keyword>[^:]+?:)?(?P<text>.*)\s?
```

That is, a possible indent (which we measure), an optional list item identifier, optional space, optional words ended by colon, and optional text. All lines are of this form. However, some ordinary (non-list) lines may contain a colon, and then the keyword and text group must be added to get the line contents. Otherwise, the text group will be the line.

When lists are typeset, the text is examined for sections, paragraphs, title, author, date, plus all the inline tags for emphasized, boldface, and verbatim text. Plain subsitutions based on regular expressions are used for this purpose.

The final step is to insert the code and TeX blocks again (these should be untouched and are therefore left out of the previous parsing).

It is important to keep the Doconce format and parsing simple. When a new format is needed and this format is not obtained by a simple edit of the definition of existing formats, it might be better to convert the document to reST and then to XML, parse the XML and write out in the new format. When the Doconce format is not sufficient to getting the layout you want, it is suggested to filter the document to another, more complex format, say reST or LaTeX, and work further on the document in this format.

7.1 Typesetting of Function Arguments, Return Values, and Variables

As part of comments (or doc strings) in computer code one often wishes to explain what a function takes of arguments and what the return values are. Similarly, it is desired to document class, instance, and module variables. Such arguments/variables can be typeset as description lists of the form listed below and *placed at the end of the doc string*. Note that `argument`, `keyword argument`, `return`, `instance variable`, `class variable`, and `module variable` are the only legal keywords (descriptions) for the description list in this context. If the output format is Epytext (Epydoc) or Sphinx, such lists of arguments and variables are nicely formatted.

```
    - argument x: x value (float),
      which must be a positive number.
    - keyword argument tolerance: tolerance (float) for stopping
      the iterations.
    - return: the root of the equation (float), if found, otherwise None.
    - instance variable eta: surface elevation (array).
    - class variable items: the total number of MyClass objects (int).
    - module variable debug: True: debug mode is on; False: no debugging
      (bool variable).
```

The result depends on the output format: all formats except Epytext and Sphinx just typeset the list as a list with keywords.

module variable x: x value (float), which must be a positive number.

module variable tolerance: tolerance (float) for stopping the iterations.

8 References

Note.

The references below are just for illustrating and testing citation syntax and not references for explaining parts of the text.

References

[1] D. E. Knuth. Theory and practice. *EATCS Bulletin*, 27:14–21, 1985.

[2] H. P. Langtangen. A document for testing Doconce. http://hplgit.github.io/doconce/test/demo_testdoc.html.

[3] H. P. Langtangen. The FEMDEQS program system. Research report in mechanics, Department of Mathematics, University of Oslo, 1989.

[4] H. P. Langtangen. Numerical solution of first passage problems in random vibrations. *SIAM Journal of Scientific and Statistical Computing*, 15:997–996, 1994.

[5] H. P. Langtangen, K.-A. Mardal, and R. Winther. Numerical methods for incompressible viscous flow. *Advances in Water Resources*, 25:1125–1146, 2002.

[6] H. P. Langtangen and G. Pedersen. Propagation of large destructive waves. *International Journal of Applied Mechanics and Engineering*, 7(1):187–204, 2002.

[7] K.-A. Mardal, G. W. Zumbusch, and H. P. Langtangen. Software tools for multigrid methods. In H. P. Langtangen and A. Tveito, editors, *Advanced Topics in Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*, Lecture Notes in Computational Science and Engineering, pages 97–152. Springer, 2003.

Index