Doconce Description

Author: Hans Petter Langtangen

Date: Jan 26, 2013

What Is Doconce?

Doconce is two things:

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, Epytext, and also plain text (where non-obvious formatting/tags are removed for clear reading in, e.g., emails). 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, blogs, emails, etc. The slogan is: "Write once, include anywhere".

Here are some Doconce features:

- Doconce has strong support for *text with computer source code and LaTeX mathematics* in the formats LaTeX, pdfLaTeX, Sphinx, HTML, and MediaWiki. One piece of text can enter (e.g.) a classical science book, an ebook, a web document, and a blog.
- Doconce targets scientific papers and reports, thesis, large book projects, software manuals, memos, as well as blogs and wikis with math and code content.
- 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.
- Doconce can be converted to plain *untagged* text, often desirable for computer code and email.
- Doconce has good support for copying in parts of computer code directly from the source code files via regular expressions for the start and end lines.
- Doconce is almost self-explanatory and fast to write. The tools can be used for handy generation of more verbose and complicated markup languages, such as LaTeX, Sphinx, HTML, MediaWiki, and Google wiki.
- Doconce first runs two preprocessors (Preprocess and Mako), which allows 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.
- Compared to related tools, foremost 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 Pandoc 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.

Installation of Doconce and its Dependencies

Doconce

Doconce itself is pure Python code hosted at http://code.google.com/p/doconce. Its installation from the Mercurial (`hg`) source follows the standard procedure:

```
# Doconce
hg clone https://doconce.googlecode.com/hg/ doconce
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
hg pull
hg update
sudo python setup.py install
```

Debian GNU/Linux users can also run:

```
sudo apt-get install doconce
```

This installs the latest release and not the most updated and bugfixed version. On Ubuntu one needs to run:

```
sudo add-apt-repository ppa:scitools/ppa
sudo apt-get update
sudo apt-get install doconce
```

Dependencies

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
```

Spellcheck

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

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

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-latex-recommended texlive-latex-extra
```

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` is 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
```

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`.

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/doc
cd docutils
sudo python setup.py install
cd ..
```

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`.

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 ..
```

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 epydo
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`.

Demos

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

```
docs/manual/manual.do.txt
```

file in the Doconce source code tree. We have made a demo web page where you can compare the Doconce source with the output in many different formats: HTML, LaTeX, plain text, etc.

The file `make.sh` in the same directory as the `manual.do.txt` file (the current text) shows how to run `doconce format` on the Doconce file to obtain documents in various formats.

Another demo is found in:

```
docs/tutorial/tutorial.do.txt
```

In the `tutorial` directory there is also a `make.sh` file producing a lot of formats, with a corresponding web demo of the results.

From Doconce to Other Formats

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

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

or just:

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

Preprocessing

The `preprocess` and `mako` programs are used to preprocess the file, and options to `preprocess` and/or `mako` can be added after the filename. For example:

```
Terminal> doconce format latex mydoc -Dextra_sections -DVAR1=5     # preprc
Terminal> doconce format latex yourdoc extra_sections=True VAR1=5  # mako
```

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

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.

HTML

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.

The HTML style can be defined either in the header of the HTML file or in an external CSS file. The latter is enabled by the command-line argument `--css=filename`. There is a default style with blue headings, and a style with the solarized color palette, specified by the `--html-solarized` command line argument. If there is no file with name `filename` in the `--css=filename` specification, the blue or solarized

styles are written to `filename` and linked from the HTML document. You can provide your own style sheet either by replacing the content inside the `style` tags or by specifying a CSS file through the `--css=filename` option.

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.

The HTML file can be embedded in a template if the Doconce document does not have a title (because then there will be no header and footer in the HTML file). 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, i.e., the Doconce document translated to HTML. The title becomes the first heading in the Doconce document, and the date is extracted from the `DATE:` line, if present. With the template feature one can easily embed the text in the look and feel of a website. The template can be extracted from the source code of a page at the site; just insert `%(title)s` and `%(date)s` at appropriate places and replace the main bod of text by `%(main)s`. Here is an example:

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

Blogs

Doconce can be used for writing blogs provided the blog site accepts raw HTML code. Google's Blogger service (`blogname.blogspot.com`) is particularly well suited since it also allows extensive LaTeX mathematics via MathJax. Write the blog text as a Doconce document without any title, author, and date. Then generate HTML as described above. Copy the text and paste it into the text area in the blog, making sure the input format is HTML. On Google's Blogger service you can use Doconce to generate blogs with LaTeX mathematics and pretty (pygmentized) blocks of computer code. See a blog example for details on blogging.

Warning

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

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. There is a doconce example on blogging with mathematics and code on WordPress.

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.

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 `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 or `align*` environments are well understood.

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.

LaTeX

Making a LaTeX file `mydoc.tex` from `mydoc.do.txt` is done in two steps: .. Note: putting code blocks inside a list is not successful in many

Step 1. Filter the doconce text to a pre-LaTeX form `mydoc.p.tex` for the `ptex2tex` program (or `doconce ptex2tex`):

```
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 the section Macros (Newcommands)). If these files are present, they are included in the LaTeX document so that your commands are defined.

An option `--latex-printed` 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).

Step 2. Run `ptex2tex` (if you have it) to make a standard LaTeX file:

```
Terminal> ptex2tex mydoc
```

In case you do not have `ptex2tex`, you may run a (very) simplified version:

```
Terminal> doconce ptex2tex mydoc
```

Note that Doconce generates a `.p.tex` file with some preprocessor 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
```

The title, authors, and date are by default typeset in a non-standard way to enable a nicer treatment of multiple authors having institutions in common. However, the standard LaTeX "maketitle" heading is also available through `-DLATEX_HEADING=traditional`. A separate titlepage can be generate by `-DLATEX_HEADING=titlepage`.

Preprocessor variables to be defined or undefined are

- `BOOK` for the "book" documentclass rather than the standard "article" class (necessary if you apply chapter headings)
- `PALATINO` for the Palatino font
- `HELVETIA` for the Helvetica font
- `A4PAPER` for A4 paper size
- `A6PAPER` for A6 paper size (suitable for reading on small devices)
- `MOVIE15` for using the movie15 LaTeX package to display movies
- `PREAMBLE` to turn the LaTeX preamble on or off (i.e., complete document or document to be included elsewhere)
- `MINTED` for inclusion of the minted package (which requires `latex` or `pdflatex` to be run with the `-shell-escape` option)

The `ptex2tex` tool makes it possible to easily switch between many different fancy formattings of computer or verbatim 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++}`. If no environments like `sys`, `fpro`, or the common `envir` are defined on the command line, the plain `\begin{verbatim}` and `\end{verbatim}` 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. Here are two examples:

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

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
```

If one wishes to run `ptex2tex` and 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 included by running `ptex2tex` with the `-DMINTED` option:

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

In this case, `latex` 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
```

When running `doconce ptex2tex mydoc envir=minted` (or other minted specifications with `doconce ptex2tex`), the minted package is automatically included so there is no need for the `-DMINTED` option.

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
```

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
```

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

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 file1 file2 file3 ...
```

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 files `file1`, `file2`, etc. that together represent the whole document. Note that none of the individual Doconce files `file1`, `file2`, etc. should include the rest as their union makes up the whole document. 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'`).

With a single-file document in `mydoc.do.txt` one often just runs:

```
Terminal> doconce sphinx_dir mydoc
```

and then an appropriate Sphinx directory `sphinx-rootdir` is made with relevant files.

The `doconce sphinx_dir` command generates a script `automake_sphinx.py` for compiling the Sphinx document into an HTML document. One can either run `automake_sphinx.py` or perform the steps in the script manually, possibly with necessary modifications. You should at least read the script prior to executing it to have some idea of what is done.

The `doconce sphinx_dir` script copies directories named `figs` or `figures` over to the Sphinx directory so that figures are accessible in the Sphinx compilation. If figures or movies are located in other directories, `automake_sphinx.py` must be edited accordingly. Files, to which there are local links (not `http:` or `file:` URLs), must be placed in the `_static` subdirectory of the Sphinx directory. The utility `doconce sphinxfix_localURLs` is run to check for local links in the Doconce file: for each such link, say `dir1/dir2/myfile.txt` it replaces the link by `_static/myfile.txt` and copies `dir1/dir2/myfile.txt` to a local `_static` directory (in the same directory as the script is run). However, we recommend instead that the writer of the document places files in `_static` or lets a script do it automatically. The user must copy all `_static/*` files to the `_static` subdirectory of the Sphinx directory. It may be wise to always put files, to which there are local links in the Doconce document, in a `_static` or `_static-name` directory and use these local links. Then links do not need to be modified when creating a Sphinx version of the document.

Doconce comes with a collection of HTML themes for Sphinx documents. These are packed out in the Sphinx directory, the `conf.py` configuration file for Sphinx is edited accordingly, and a script `make-themes.sh` can make HTML documents with one or more themes. For example, to realize the themes `fenics` and `pyramid`, one writes:

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

The resulting directories with HTML documents are `_build/html_fenics` and `_build/html_pyramid`, respectively. Without arguments, `make-themes.sh` makes all available themes (!).

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.

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).

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 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.

The Doconce Markup Language

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

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)

Special Lines (1)

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.

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 Researc
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, 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`.
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.

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 Chapter Heading =========

Some text.
```

```
======= 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.
```

Special Lines (2)

Figures

Figures are recognized by the special line syntax:

```
FIGURE:[filename, height=xxx width=yyy scale=zzz] possible 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 scale keywords (and others) can be included if desired and may have effect for some formats. Note the comma between the sespecifications and that there should be no space around the = sign.

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).

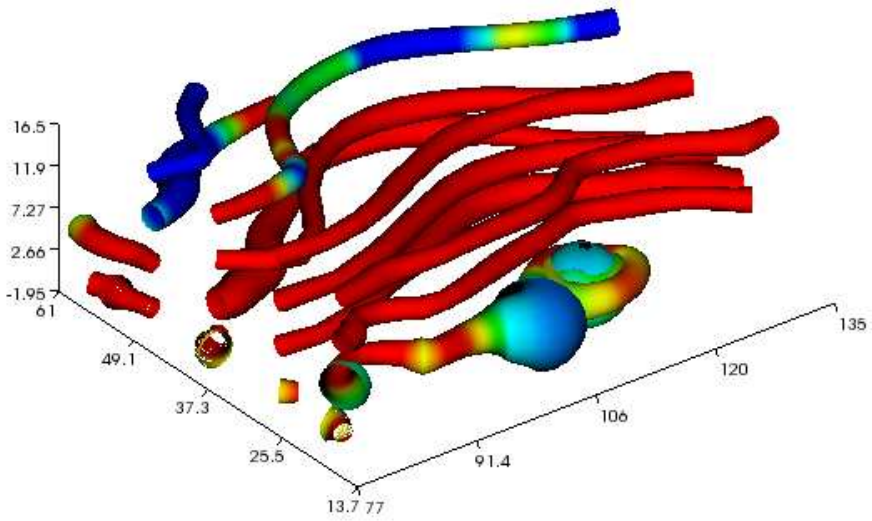

Figure 1: *Streamtube visualization of a fluid flow* (fig:viz)

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 file2 ... file4 result
```

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).

Movies

Here is an example on the `MOVIE:` keyword for embedding movies. 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:

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

The LaTeX format results in a file that can either make use of the movie15 package (requires the PDF to be shown in Acrobat Reader) or just a plain address to the movie. The HTML, reST, and Sphinx formats will play the movie right away by embedding the file in a standard HTML code, provided the output format is HTML. For all other

formats a URL to an HTML file, which can play the code, is inserted in the output document.

When movies are embedded in the PDF file via LaTeX and the `movie15` package wanted, one has to turn on the preprocessor variable `MOVIE15`. There is an associated variable `EXTERNAL_MOVIE_VIEWER` which can be defined to launch an external viewer when displaying the PDF file (in Acrobat Reader):

```
Terminal> ptex2tex -DMOVIE15 -DEXTERNAL_MOVIE_VIEWER mydoc
```

The HTML, reST, and Sphinx formats can also treat filenames of the form `myframes*`. In that case, an HTML file for showing the sequence of frames is generated, and a link to this file is inserted in the output document. That is, a simple "movie viewer" for the frames is made.

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

1. click on *Share* and then *Embed*
2. copy the URL for the embedded movie
3. note the height and width of the embedded movie

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

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

Doconce will be able to embed standard YouTube URLs also, but then the width and height might be inappropriate.

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 the section Blocks of Verbatim Computer Code below.

Inline Tagging

Doconce supports tags for *emphasized phrases*, **boldface phrases**, and `verbatim text` (also called type writer text, for inline code) plus LaTeX/TeX inline mathematics, such as v = 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.

> **Note**
>
> 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:"http://code.google.com/p/doconce".
```

which gets rendered as Click on this link: http://code.google.com/p/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.)

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.

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. 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 the section From Doconce to Other Formats 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 the section From Doconce to Other Formats).

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 Ax=b, where A is an nxn matrix, and x and b are vectors of length n."

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 the section Inline Tagging.

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).

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 fig:viz (the label appears in the figure caption in the source code of this document). Additional references to the sections LaTeX Blocks of Mathematical Text and Macros (Newcommands) are nice to demonstrate, as well as a reference to equations, say Equations (myeq1)-(myeq2). 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 the section Inline Tagging.

Generalized Cross-Referencing

Sometimes a series of individual documents may be assembled to one large document. The assembly impacts how references to sections are written: when referring to a section in the same document, a label can be used, while references to sections in other documents are written differently, sometimes involving a link (URL) and a citation. Especially if both the individual documents and the large assembly document are to exist side by side, a flexible way of referencing is needed. For this purpose, Doconce offers *generalized references* which allows a reference to have two different formulations, one for internal references and one for external references. Since LaTeX supports references to labels in external documents via the `xr` package, the generalized references in Doconce has a syntax that may utilize the `xr` feature in LaTeX.

The syntax of generalized references reads:

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

If all `label`_` references in the text ``internal` are references to labels in the present document, the above `ref` command is replaced by the text `internal`. Otherwise, if cite is non-empty and the format is `latex` or `pdflatex` one assumes that the references in `internal` are to external documents declared by a comment line `# Externaldocuments: testdoc, mydoc` (usually after the title, authors, and date). In this case the output text is `internal cite` and the LaTeX package `xr` is used to handle the labels in the external documents. If none of the two situations above applies, the `external` text will be the output.

Here is an example on a specific generalized reference:

```
As explained in
ref[Section ref{subsec:ex}][in "Langtangen, 2012":
"http://code.google.com/p/doconce/wiki/Description"
cite{testdoc:12}][a "section": "testdoc.html#___sec2" in
```

```
the document "A Document for Testing Doconce": "testdoc.html"
cite{testdoc:12}], Doconce documents may include movies.
```

In LaTeX, this becomes:

```
As explained in
Section~\ref{subsec:ex} in
\href{{http://code.google.com/p/doconce/source/browse/test/testdoc.do.txt}}
\cite{testdoc:12}, Doconce documents may include movies.
```

Note that there is a specific numbered reference to an external document, if `subsec:ex` is not a label in the present document, and that we add a citation in the usual way, but also include a link to the document using the name of the other or some other relevant link text. The link can be the same or different from links used in the "external" part of the reference (LaTeX cannot have links to local files, so a complete URL must be used).

Translation to Sphinx or reStructuredText results in:

```
As explained in
a `section <testdoc.html#___sec2>`_ in
the document `A Document for Testing Doconce <testdoc.html>`_
[testdoc:12]_, Doconce documents may include movies.
```

In plain HTML, this becomes:

```
As explained in
a <a href="testdoc.html#___sec2">section</a> in
the document <a href="testdoc.html">A Document for Testing Doconce</a>
<a href="#testdoc:12">[1]</a>, Doconce documents may include movies.
```

The plain text format reads:

```
As explained in
a section (testdoc.html#___sec2) in
the document A Document for Testing Doconce (testdoc.html)
[1], Doconce documents may include movies.
```

And in Pandoc-exteded Markdown we have:

```
As explained in
a [section](testdoc.html#___sec2) in
the document [A Document for Testing Doconce](testdoc.html)
@testdoc:12, Doconce documents may include movies.
```

Index and Bibliography

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).

Literature citations also follow a LaTeX-inspired style:

```
as found in cite{Larsen_1986,Nielsen_Kjeldstrup_1999}.
```

Citation labels can be separated by comma. In LaTeX, this is directly translated to the corresponding `cite` command; in reST and Sphinx the labels can be clicked, while in all the other text formats the labels are consecutively numbered so the above citation will typically look like:

```
as found in [3][14]
```

if `Larsen_1986` has already appeared in the 3rd citation in the document and `Nielsen_Kjeldstrup_1999` is a new (the 14th) citation. The citation labels can be any sequence of characters, except for curly braces and comma.

The bibliography itself is specified by the special keyword `BIBFILE:`, followed by a BibTeX file with extension `.bib`, a corresponding reST bibliography with extension `.rst`, or simply a Python dictionary written in a file with extension `.py`. The dictionary in the latter file should have the citation labels as keys, with corresponding values as the full reference text for an item in the bibliography. Doconce markup can be used in this text, e.g.:

```
{
'Nielsen_Kjeldstrup_1999': """
K. Nielsen and A. Kjeldstrup. *Some Comments on Markup Languages*.
URL:"http://some.where.net/nielsen/comments", 1999.
""",
'Larsen_1986':
"""
O. B. Larsen. On Markup and Generality.
Personal Press*. 1986.
"""
}
```

In the `latex` and `pdflatex` formats, the `.bib` file will be used in the standard BibTeX way. In the `rst` and `sphinx` formats, the `.rst` file will be copied into the document at the place where the `BIBFILE:` keyword appears, while all other formats will make use of the Python dictionary typeset as an ordered Doconce list inserted at the `BIBFILE:` line in the document.

Only one file with bibliographic references can be used. It is recommended to create all references in BibTeX format. Say the file is `myfile.bib`. Insert `BIBFILE: myfile.bib` at the end of the file (for instance). Then make a LaTeX document and check that the references appear correctly. A next step can be to create the `.rst` file, either by manual editing of `myfile.bbl` or using `doconce bbl2rst myfile.bbl`

to automate (most of) this editing. From the `myfile.rst` file it is easy to create `myfile.py` with the dictionary version of the references.

Conversion of BibTeX databases to reST format can be done by the bibliograph.parsing tool.

Finally, we here test the citation command and bibliography by citing a book [Python:Primer:09], a paper [Osnes:98], and both of them simultaneously [Python:Primer:09] [Osnes:98].

(**somereader**: comments, citations, and references in the latex style is a special feature of doconce :-))

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). 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.

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
```

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:

```
===== {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.

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 in Linux and Sphinx). 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.

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.)

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
```

Here is the result:

```
\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}
```

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.extmath'` 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.

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).

Example. Suppose we have the following commands in `newcommand_replace.tex`:

```
\newcommand{\beqa}{\begin{eqnarray}}
\newcommand{\eeqa}{\end{eqnarray}}
\newcommand{\ep}{\thinspace . }
\newcommand{\uvec}{\vec u}
\newcommand{\Q}{\pmb{Q}}
```

and these in `newcommands_keep.tex`:

```
\newcommand{\x}{\pmb{x}}
\newcommand{\normalvec}{\pmb{n}}
\newcommand{\Ddt}[1]{\frac{D#1}{dt}}
\newcommand{\half}{\frac{1}{2}}
```

The LaTeX block:

```
\beqa
\x\cdot\normalvec &=& 0, label{my:eq1}\\
\Ddt{\uvec} &=& \Q \ep   label{my:eq2}
\eeqa
```

will then be rendered to:

```
\begin{eqnarray}
\x\cdot\normalvec &=& 0, label{my:eq1}\\
\Ddt{{\vec u}} &=& \pmb{Q} {\thinspace . }   label{my:eq2}
\end{eqnarray}
```

in the current format.

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.
```

Other user-defined variables for the preprocessor can be set at the command line as explained in the section From Doconce to Other Formats.

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

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 reST and Sphinx documents it is a point to have separate `.rst` files and an index file listing the various `.rst` that build up the document. To generate the various `.rst` files one should not run Doconce on the individual `.do.txt` files, because then references and index entries are not treated correctly. Instead, run Doconce on the master file and invoke the script `doconce split_rst` to split the long, complete `.rst` into pieces. This process requires that each `#include "file.do.txt` line in the master file is preceded by a "marker line" having the syntax `#>>>>>> part: file >>>>>>`, where `file` is the filename without extension. The number of greater than signs is not important, but it has to be a comment line and it has to contain the keyword `part:`.

Here is an example. Say the name of the master file is `master.do.txt`. The following Bash script does the job: We run:

```
doconce format sphinx master
# Split master.rst into parts
# as defined by #>>>>> part: name >>>>> lines
files=`doconce split_rst master.rst`

dir=sphinxm-rootdir
```

```
if [ ! -d $dir ]; then
  doconce sphinx_dir dirname=$dir author='me and you' \
          version=1.0 theme=default $files
  sh automake_sphinx.sh
else
  for file in $files; do
    cp $file.rst $dir
  done
  cd $dir
  make html
  cd ..
fi
```

The autogenerated `automake_sphinx.sh` file (by `doconce sphinx_dir`) is compatible with a master `.rst` file split into pieces as long as the complete set of pieces in correct order is given to `doconce sphinx_dir`. This set is the output of `doconce split_rst`, which we catch in a variable `files` above.

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 docoment in another format. Inside this script, we may edit and fine-tune the output from Doconce.

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.

Emacs Doconce Formatter

The file `misc/.doconce-mode.el` in the Doconce source distribution gives a "Doconce Editing Mode" in Emacs. The file is a rough edit of the reST Editing Mode for Emacs. Some Doconce features are recognized, but far from all, and sometimes portions of Doconce text just appear as ordinary text.

Here is how to get the Doconce Editing Mode in Emacs.

Step 1. Download the Doconce tarball from `code.google.com/p/doconce`, pack it out and go to the root directory.

Step 2. Copy the `doconce-mode.el` file to the home directory:

```
cp misc/.doconce-mode.el $HOME
```

Step 3. Add these lines to `$HOME/.emacs`:

```
(load-file "~/hg/.doconce-mode.el")
(setq auto-mode-alist(cons '("\\.do\\.txt$" . doconce-mode) auto-mode-alist
```

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

Troubleshooting

Disclaimer

Doconce has some support for syntax checking. If you encounter Python errors while running `doconce format`, the reason for the error is most likely a syntax problem in your Doconce source file. You have to track down this syntax problem yourself.

However, the problem may well be a bug in Doconce. The Doconce software is incomplete, and many special cases of syntax are not yet discovered to give problems. Such special cases are also seldom easy to fix, so one important way of "debugging" Doconce is simply to change the formatting so that Doconce treats it properly. Doconce is very much based on regular expressions, which are known to be non-trivial to debug years after they are created. The main developer of Doconce has hardly any time to work on debugging the code, but the software works well for his diverse applications of it.

General Problems

Doconce aborts because of a syntax error that is not an error

Doconce searches for typical syntax errors and usually aborts the execution if errors are found. However, it may happen, especially in verbatim blocks, that Doconce reports syntax errors that are not errors. To continue execution, simply add the `--no-abort` option on the command line. You may send an email to the Doconce author at `hpl@simula.no` and report the problem.

The Mako preprocessor is seemingly not run

If you have lines starting with `%` inside code segments (for example, SWIG code or Matlab comment lines), the Mako preprocessor will crash because it thinks these lines are Mako statements. Doconce detects this problem and avoids running Mako. Examine the output from Doconce: warnings are issued if Mako is not run.

Something goes wrong in the preprocessing step

Doconce automatically removes the file `__tmp.do.txt`, which is the resulting of the preprocessing stge and the file to examine if something goes wrong in this stage (i.e., when `mako` and `preprocess` is run). Add the `--debug` flag at the end of the `doconce` command to (both make a debug file and) avoid that `__tmp.do.txt` is deleted.

Figure captions are incomplete

If only the first part of a figure caption in the Doconce file is seen in the target output format, the reason is usually that the caption occupies multiple lines in the Doconce file. The figure caption must be written as *one line*, at the same line as the FIGURE keyword.

Preprocessor directives do not work

Make sure the preprocessor instructions, in Preprocess or Mako, have correct syntax. Also make sure that you do not mix Preprocess and Mako instructions. Doconce will then only run Preprocess.

Problems with boldface and emphasize

Two boldface or emphasize expressions after each other are not rendered correctly. Merge them into one common expression.

Links to local directories do not work

Links of the type:

```
see the "examples directory": "src/examples"
```

do not work well. You need to link to a specific HTML file:

```
see the "examples directory": "src/examples/index.html"
```

Links are not typeset correctly

Not all formats will allow formatting of the links. Verbatim words in links are allowed if the whole link is typeset in verbatim:

```
see the directory "`examples`": "src/examples/index.html".
```

However, the following will not be typeset correctly:

```
see the "`examples` directory": "src/examples/index.html"
```

The back-ticks must be removed, or the text can be reformulated as in the line above it.

Inline verbatim code is not detected

Make sure there is a space before the first back-tick.

Inline verbatim text is not formatted correctly

Make sure there is whitespace surrounding the text in back-ticks.

Strange non-English characters

Check the encoding of the `.do.txt` file with the Unix `file` command or with:

```
Terminal> doconce guess_encoding myfile.do.txt
```

If the encoding is utf-8, convert to latin-1 using either of the Unix commands:

```
Terminal doconce change_encoding utf-8 LATIN1 myfile.do.txt

Terminal> iconv -f utf-8 -t LATIN1 myfile.do.txt --output newfile
```

Wrong Norwegian charcters

When Doconce documents have characters not in the standard ASCII set, the format of the file must be LATIN1 and not UTF-8. See the section "Strange non-English characters" above for how to run `doconce change_encoding` to change the encoding of the Doconce file.

Too short underlining of reST headlines

This may happen if there is a paragraph heading without proceeding text before some section heading.

Found !bt but no tex blocks extracted (BUG)

This message points to a bug, but has been resolved by removing blank lines between the text and the first `!bt` (inserting the blanks again did not trigger the error message again...).

Problems with code or Tex Blocks

Code or math block errors in reST

First note that a code or math block must come after some plain sentence (at least for successful output in reST), not directly after a section/paragraph heading, table, comment, figure, or movie, because the code or math block is indented and then become parts of such constructions. Either the block becomes invisible or error messages are issued.

Sometimes reST reports an "Unexpected indentation" at the beginning of a code block. If you see a `!bc`, which should have been removed when running `doconce format sphinx`, it is usually an error in the Doconce source, or a problem with the rst/sphinx translator. Check if the line before the code block ends in one colon (not two!), a question mark, an exclamation mark, a comma, a period, or just a newline/space after text. If not, make sure that the ending is among the mentioned. Then `!bc` will most likely be replaced and a double colon at the preceding line will appear (which is the right way in reST to indicate a verbatim block of text).

Strange errors around code or TeX blocks in reST

If `idx` commands for defining indices are placed inside paragraphs, and especially right before a code block, the reST translator (rst and sphinx formats) may get confused and produce strange code blocks that cause errors when the reST text is transformed to other formats. The remedy is to define items for the index outside paragraphs.

Something is wrong with a verbatim code block

Check first that there is a "normal" sentence right before the block (this is important for reST and similar "ASCII-close" formats).

Code/TeX block is not shown in reST format

A comment right before a code or tex block will treat the whole block also as a comment. It is important that there is normal running text right before `!bt` and `!bc` environments.

Verbatim code blocks inside lists look ugly

Read the the section Blocks of Verbatim Computer Code above. Start the `!bc` and `!ec` tags in column 1 of the file, and be careful with indenting the surrounding plain text of the list item correctly. If you cannot resolve the problem this way, get rid of the list and use paragraph headings instead. In fact, that is what is recommended: avoid verbatim code blocks inside lists (it makes life easier).

LaTeX code blocks inside lists look ugly

Same solution as for computer code blocks as described in the previous paragraph. Make sure the `!bt` and `!et` tags are in column 1 and that the rest of the non-LaTeX surrounding text is correctly indented. Using paragraphs instead of list items is a good idea also here.

Problems with reST/Sphinx Output

Title level inconsistent

reST does not like jumps in the levels of headings. For example, you cannot have a `===` (paragraph) heading after a `=======` (section) heading without a `=====` (subsection) heading in between.

Lists do not appear in .rst files

Check if you have a comment right above the list. That comment will include the list if the list is indentend. Remove the comment.

Error message "Undefined substitution..." from reST

This may happen if there is much inline math in the text. reST cannot understand inline LaTeX commands and interprets them as illegal code. Just ignore these error messages.

Warning about duplicate link names

Link names should be unique, but if (e.g.) "file" is used as link text several places in a reST file, the links still work. The warning can therefore be ignorned.

Inconsistent headings in reST

The `rst2*.py` and Sphinx converters abort if the headers of sections are not consistent, i.e., a subsection must come under a section, and a subsubsection must come under a subsection (you cannot have a subsubsection directly under a section). Search for ===, count the number of equality signs (or underscores if you use that) and make sure they decrease by two every time a lower level is encountered.

No code environment appears before "bc ipy" blocks

The `!bc ipy` directive behaves this way for `sphinx` output because interactive sessions are automatically handled. If this is not appropriate, shift to `!bc cod` or another specification of the verbatim environment.

Problems with LaTeX Output

LaTeX does not like underscores in URLs

Suppose you have a URL reference like:

```
..which can be found in the file "my_file.txt":
"http://some.where.net/web/dir/my_file.txt".
```

LaTeX will stop with a message about a missing dollar sign. The reason is that underscores in link texts need to be preceded by a backslash. However, this is incovenient to do in the Doconce source since the underscore is misleading in other formats. The remedy is to format the link text with inline verbatim tags (backticks):

```
..which can be found in the file "`my_file.txt`":
"http://some.where.net/web/dir/my_file.txt".
```

Verbatim text in links works fine with underscores.

Error when running latex: You must have 'pygmentize' installed

This message points to the use of the minted style for typesetting verbatim code. You need to include the `-shell-escape` command-line argument when running `latex` or `pdflatex`:

```
Terminal> latex -shell-escape file mydoc.tex
Terminal> pdflatex -shell-escape file mydoc.tex
```

Using `doconce ptex2tex` will turn on the minted style if specified as environment on the command line, while using `ptex2tex` requires the preprocessor option `-DMINTED` to turn on the minted package. When this package is included, `latex` or `pdflatex` runs the `pygmentize` program and the `shell-escape` option is required.

How can I use my fancy LaTeX environments?

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 some flexibility for typesetting computer code.

The LaTeX file does not compile

If the problem is undefined control sequence involving:

```
\code{...}
```

the cause is usually a verbatim inline text (in back-ticks in the Doconce file) spans more than one line. Make sure, in the Doconce source, that all inline verbatim text appears on the same line.

Inline verbatim gives error

Check if the inline verbatim contains typical LaTeX commands, e.g.:

```
some text with `\usepackage{mypack}` is difficult because
ptex2tex will replace this by \code{\usepackage{mypack}} and
then replace this by
{\fontsize{10pt}{10pt}\verb!\usepackage{mypack!}}
which is wrong because ptex2tex applies regex that don't
capture the second }
```

The remedy is to place verbatim LaTeX commands in verbatim blocks - that is safe.

Errors in figure captions

Such errors typically arise from unbalanced curly braces, or dollar signs around math, and similar LaTeX syntax errors.

(Note that verbatim font is likely to cause trouble inside figure captions, but Doconce will automatically replace verbatim text in back-ticks by a proper `texttt` command (since verbatim font constructions does not work inside figure captions) and precede underscores by backslash.)

Chapters are ignored

The default LaTeX style is "article". If you chapters in the Doconce file, you need to run `ptex2tex` with the option `-DBOOK` to set the LaTeX documentstyle to "book".

I want to tune the top of the LaTeX file

The top of the LaTeX file, as generated by Doconce, is very simple. If this LaTeX code is not sufficient for your needs, there are two ways out of it:

1. Make a little Bash script that performs a series of `doconce subst` (regular expressions) or `doconce replace` (regular text) substitutions to change the text automatically (you probably have to repeat these edits so automating them is a good idea).

2. Place the title, author(s), and date of the Doconce file in a separate file and use the preprocessor to include the rest. The rest is then one or more Doconce files without title, author(s), and date. This means that the `doconce format latex` command does not generate the LaTeX intro (preamble) and outro, just the core text, for these files. Make a new file by hand with the appropriate LaTeX intro and outro text and include the various text pieces in this file. To make the LaTeX document, you compile all Doconce files into LaTeX code, except the "top" Doconce file that includes the others. That file is not used for LaTeX output, but replaced by the hand-written LaTeX "top" file.

Problems with gwiki Output

Strange nested lists in gwiki

Doconce cannot handle nested lists correctly in the gwiki format. Use nonnested lists or edit the `.gwiki` file directly.

Lists in gwiki look ugly in the gwiki source

Because the Google Code wiki format requires all text of a list item to be on one line, Doconce simply concatenates lines in that format, and because of the indentation in the original Doconce text, the gwiki output looks somewhat ugly. The good thing is that this gwiki source is seldom to be looked at - it is the Doconce source that one edits further.

Problems with HTML Output

How can I change the layout of the HTML page?

The standard of way of controlling the HTML format is to use an HTML template. The Doconce source is then the body of text (leave out `TITLE:` to get HTML without a header and footer). The `--html-template=filename` command-line option will then embed the Doconce text in the specified template file, where you can use style sheets and desired constructs in the header and footer. The template can have "slots" for a title (`%(title)s`), a date (`%(date)s`), and the main body of text (`%(main)s`). For typesetting code, `pygments` is used (if installed) and can be turned off by `--no-pygments-html` (leaving code in gray boxes).

The easiest way is to get fancy layouts in HTML is to use the `sphinx` format and one its many themes.

A third, more primitive alternative is to edit the style in the top of the HTML file (preferably done automatically via `doconce replace` and `doconce subst` in the script that generates the final documents).

Why do figures look ugly when using HTML templates?

The HTML header that Doconce generates contain special styles for figure captions and the horizontal rule above figures. When using templates these styles are not defined, resulting in a rule that spans the width and a centered caption. Changing the appearance of the rule and caption can either be done by inserting styles or simply by automatic editing of the HTML code in a little shell script:

```
doconce replace '<p class="caption">' \
 '<p style="width: 50%; font-style: italic; color: blue">' mydoc.html
doconce replace '<hr class="figure">' \
 '<hr style="width: 50%">' mydoc.html
```

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.

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.

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.

[Python:Primer:09] H. P. Langtangen. *A Primer on Scientific Programming with Python*. Springer, 2009.

[Osnes:98] H. Osnes and H. P. Langtangen. An efficient probabilistic finite element method for stochastic groundwater flow. *Advances in Water Resources*, vol 22, 185-195, 1998.