Doconce Description

Author: Hans Petter Langtangen

Date: Dec 11, 2011

What Is Doconce?

Doconce is two things:

1. 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, wiki, LaTeX, PDF, reStructuredText (reST), Sphinx, Epytext, and also plain text (where non-obvious formatting/tags are removed for clear reading in, e.g., emails). From reST you can go to XML, HTML, LaTeX, PDF, OpenOffice, and from the latter to RTF and MS Word. (An experimental translator to Pandoc is under development, and from Pandoc one can generate Markdown, reST, LaTeX, HTML, PDF, DocBook XML, OpenOffice, GNU Texinfo, MediaWiki, RTF, Groff, and other formats.)
2. Doconce is a working strategy for never duplicating information. Text is written in a single place and then transformed to a number of different destinations of diverse type (software source code, manuals, tutorials, books, wikis, memos, emails, etc.). The Doconce markup language support this working strategy. The slogan is: "Write once, include anywhere".

Here are some Doconce features:

- Doconce markup does include tags, so the format is more tagged than Markdown, Pandoc, and AsciiDoc, but less than reST, and very much less than LaTeX and HTML.
- Doconce can be converted to plain *untagged* text, often desirable for computer programs and email.
- Doconce has good support for copying in parts of computer code directly from the source code files.
- Doconce has full support for LaTeX math, and integrates very well with big LaTeX projects (books).
- Doconce is almost self-explanatory and is a handy starting point for generating documents in more complicated markup languages, such as Google wiki, LaTeX, and Sphinx. A primary application of Doconce is just to make the initial versions of a Sphinx or wiki document.
- Contrary to the similar Pandoc translator, Doconce integrates with Sphinx and Google wiki. However, if these formats are not of interest, Pandoc is a superior tool.

Doconce was particularly written for the following sample applications:

- Large books written in LaTeX, but where many pieces (computer demos, projects, examples) can be written in Doconce to appear in other contexts in other formats, including plain HTML, Sphinx, or MS Word. With ptex2tex, Doconce can display computer code in LaTeX in many sophisticated ways (almost 50 styles are available).
- Software documentation, primarily Python doc strings, which one wants to appear as plain untagged text for viewing in Pydoc, as reST for use with Sphinx, as wiki text when publishing the software at web sites, and as LaTeX integrated in, e.g., a thesis.
- Quick memos, which start as plain text in email, then some small amount of Doconce tagging is added, before the memos can appear as MS Word documents, in wikis, or as Sphinx documents.

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 may face problems when transformed to other formats.

Dependencies

If you make use of preprocessor directives in the Doconce source, either Preprocess or Mako must be installed. To make LaTeX documents (without going through the reStructuredText format) you also need ptex2tex and some style files that `ptex2tex` potentially makes use of. Going from reStructuredText to formats such as XML, OpenOffice, HTML, and LaTeX requires docutils. Making Sphinx documents requires of course Sphinx. All of the mentioned potential dependencies are pure Python packages which are easily installed. If translation to Pandoc is desired, the Pandoc Haskell program must of course be installed.

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
```

The `mako` or `preprocess` programs are always used to preprocess the file first, and options to `mako` or `preprocess` 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`. 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"`.

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.

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** `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.

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

```
Terminal> ptex2tex mydoc
```

or just perform a plain copy:

```
Terminal> cp mydoc.p.tex mydoc.tex
```

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
```

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:

```
Terminal> ptex2tex -DLATEX_HEADING=traditional mydoc
```

A separate titlepage can be generate by:

```
Terminal> ptex2tex -DLATEX_HEADING=titlepage mydoc
```

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 cod` for a code snippet, where `cod` is set to a certain environment in `.ptex2tex.cfg` (e.g., `CodeIntended`). There are over 30 styles to choose from.

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 use the `Minted_Python`, `Minted_Cpp`, etc., environments in `ptex2tex` for typesetting code, the `minted` LaTeX package is needed. This package is included by running `doconce format` 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
```

The `-shell-escape` option is required because the `minted.sty` style file runs the `pygments` program to format code, and this program cannot be run from `latex` without the `-shell-escape` option.

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. That is, one can easily go from Doconce to Microsoft Word.

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.sh` for compiling the Sphinx document into an HTML document. This 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.sh` must be edited accordingly. One can either run `automake-sphinx.sh` or perform the steps in the script manually.

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 are the complete manual procedure of generating a Sphinx document from a file `mydoc.do.txt`.

Step 1. Translate Doconce into the Sphinx dialect of the reStructuredText format:

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

Step 2. Create a Sphinx root directory with a `conf.py` file, 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
```

Step 3. Copy the `tutorial.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`.

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
```

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.

Google Code Wiki

There are several different wiki dialects, but Doconce only support the one used by Google Code. The transformation to this format, called `gwiki` to explicitly mark it as the Google Code dialect, is done by:

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

You can then open a new wiki page for your Google Code project, copy the `mydoc.gwiki` output file from `doconce format` and paste the file contents into the wiki page. Press **Preview** or **Save Page** to see the formatted result.

When the Doconce file contains figures, each figure filename must 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

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. 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 at Segfault, Cyberspace Inc.
AUTHOR: A. Dummy Author
DATE: November 9, 2016
```

Note the 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 [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.

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.

========= 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

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).

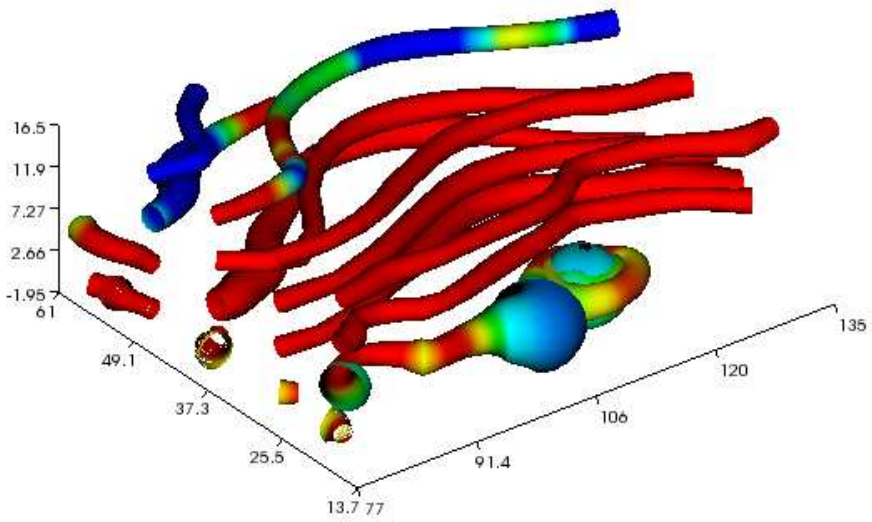

Figure 1: Streamtube visualization of a fluid flow (fig:viz)

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 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 parameters appear when you request the embedded HTML code for the movie on the YouTube page).

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 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*.

Verbatim text, typically used for short inline code, is typeset between backquotes:

```
‘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.

Watch out for mixing backquotes 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).

Web addresses with links are typeset as:

```
some URL like "Doconce": "http://code.google.com/p/doconce"
```

which appears as some URL like Search Google. The space after colon is optional. Links to files ending in `.txt`, `.html`, `.pdf`, `.py`, `.f`, `.c`, `.cpp`, `.cxx`, `.pl`, and `.java` follows the same setup:

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

which appears as see the Doconce Manual.

Links to files with other extensions must be realized *with the filename as link text*, written as the keyword URL, followed by a colon, and then the filename enclosed in double quotes:

```
URL: "manual.xml"
```

resulting in the link manual.xml.

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".
```

resulting in Click on this link: http://code.google.com/p/doconce.

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. (**hpl**: 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 chapter 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 chapter From Doconce to Other Formats).

Inline mathematics is written as in LaTeX, i.e., inside dollar signs. Most formats leave this syntax as it is (including to dollar signs), hence nice math formatting is

only obtained in LaTeX (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 LaTeX, 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."

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 Equation (my:eq1)--Equation (my:eq2). 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.

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:86,Nielsen:99}.
```

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:86` has already appeared in the 3rd citation in the document and `Nielsen:99` 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:`, which is optionally followed by a BibTeX file, having extension `.bib`, a corresponding reST bibliography, having 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:99': """
K. Nielsen. *Some Comments on Markup Languages*.
URL:"http://some.where.net/nielsen/comments", 1999.
""",
'Larsen:86':
"""
O. B. Larsen. On Markup and Generality.
Personal Press*. 1986.
"""
}
```

In the LaTeX format, the `.bib` file will be used in the standard 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, replacing the `BIBFILE:` line in the document.

At present, only one file with bibliographic references can be used.

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, or Projects

Doconce has special support for three types of "exercises", named *exercise*, *problem*, or *project*. These are all typeset as special kind of sections. Such sections start with a subsection or subsubsection headline, indicated by 3 or 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", or "Project", 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 it) and a solution file:

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

__Hint 1.__ Do not plan a travel to the moon.

__Hint 2.__ Wikipedia is always helpful.
```

Doconce will recognize the exercise, problem, or project *title*, the *label*, the *result file*, the *solution file* (if any of these three entities is present), the *text*, and a sequence of *hints*. Tailored formatting of exercises in special output formats can make use of this. 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.

Because exercises, problems, and projects are typeset as ordinary sections (this is the most general approach that will work for many formats), one must refer to an exercise, problem, or project 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 ..." works well in all formats ("..." will be a number in LaTeX and the problem title in most other formats).

It is recommended to collect all exercises as subsetions (or subsubsections) under a section (or subsection) named "Exercises", "Problems", or "Projects".

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.

There may be an argument after the `!bc` tag to specify a certain environment (for `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` can define what some environment maps 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. 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):

```
# 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
```

(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.

Here is the result of a `!bt - !et` block:

```
\begin{eqnarray}
{\partial u\over\partial t} &=& \nabla^2 u + f, label{myeq1}\\
```

```
{\partial v\over\partial t} &=& \nabla\cdot(q(u)\nabla v) + g
\end{eqnarray}
```

This text looks ugly in all Doconce supported formats, except from LaTeX and Sphinx. If HTML is desired, the best is to filter the Doconce text first to LaTeX and then use the widely available tex4ht tool to convert the dvi file to HTML, or one could just link a PDF file (made from LaTeX) directly from HTML. For other textual formats, it is best to avoid blocks of mathematics and instead use inline mathematics where it is possible to write expressions both in native LaTeX format (so it looks good in LaTeX) and in a pure text format (so it looks okay in other formats).

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{\mathbfx}[1]{{\mbox{\boldmath $#1$}}}
\newcommand{\Q}{\mathbfx{Q}}
```

and these in `newcommands_keep.tex`:

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

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} &=& {\mbox{\boldmath $Q$}} \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. For example, one may create special LaTeX output for figures, say with multiple plots within a figure, while other formats may apply a separate figure for each plot. Below is an example (see the Doconce source code of this document to understand how preprocess is used to create the example).

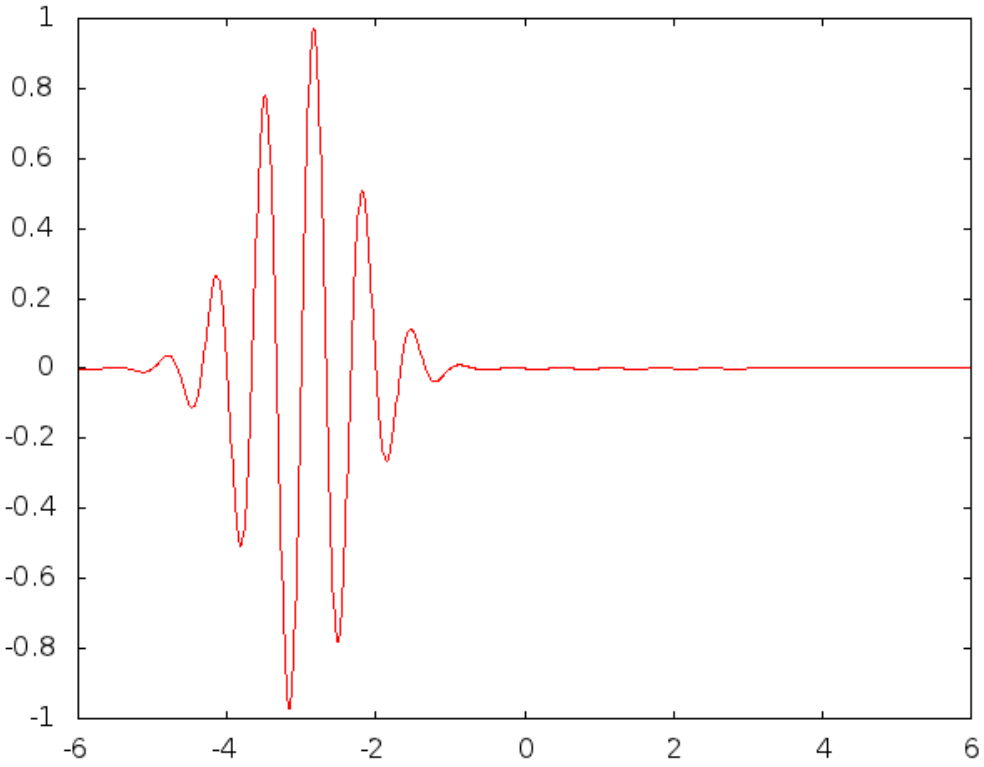

Figure 2: Wavepacket at time 0.1 s

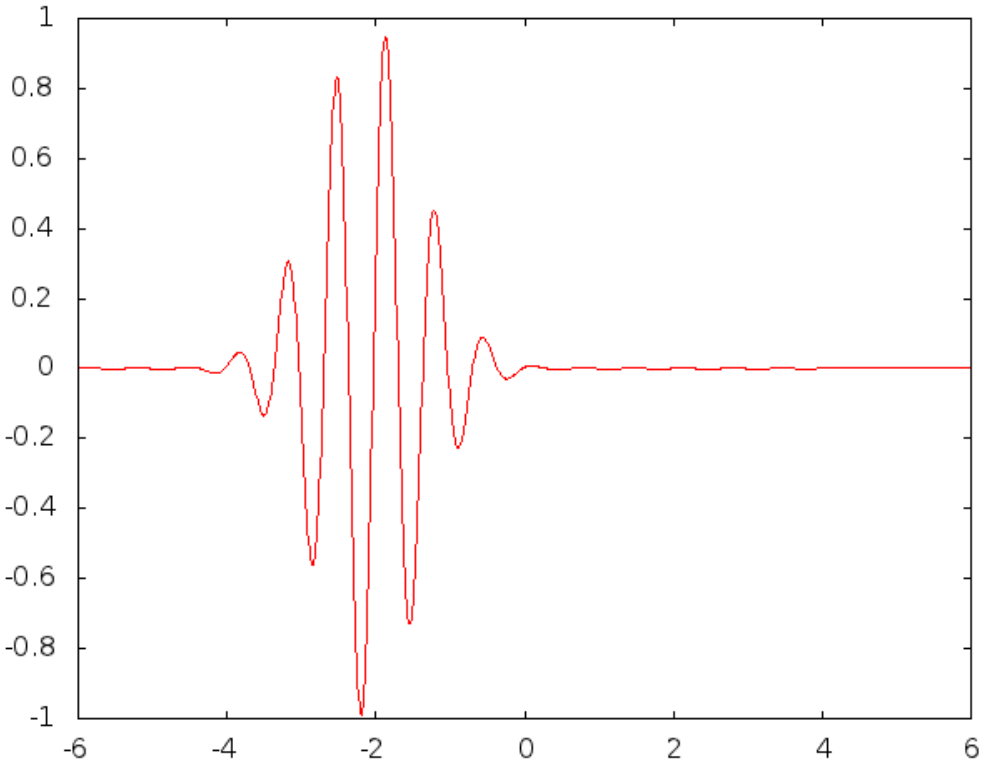

Figure 3: Wavepacket at time 0.2 s

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 num-

ber 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, test on the `FORMAT` variable (`#if FORMAT == "latex"`) in the Doconce code and insert the table in LaTeX format. The `else` or `elif` branches may have the table in other formats or a simplified version in the Doconce table format. 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.

As an example, say you want a table of contents in the LaTeX output (Doconce does not support table of contents). By inserting a recognizable comment in the Doconce source, say:

```
# table of contents
```

we can use this comment to edit the LaTeX file. First, we run Doconce `doconce format latex mydoc` to produce `mydoc.p.tex`. Then we use the `doconce replace` and `doconce subst` commands to replace the comment by the comment plus the table of contents command, or just the latter:

```
Terminal> doconce replace '% table of contents'
          '\tableofcontents' mydoc.p.tex
```

The `doconce replace from_text to_text filename` command performs a character-by-character replacement (using the `replace` method in string objects in Python). If we want to preserve the comment and add a new line with `\tableofcontents`, we should use `doconce subst`, which applies regular expressions for substitutions and thereby understands the newline character:

```
Terminal> doconce subst '% table of contents' \
          '% table of contents\n\\tableofcontents' mydoc.p.tex
```

Note the double backshlash in front of the `t` character: without it we would get a tab and no backslash. The `doconce subst` is a powerful way to automatically edit the output from Doconce and fine-tune a LaTeX document. Use of comment lines to identify portions of the file to be edited is a smart idea. Alternatively, the relevant LaTeX constructions can be inserted directly in the Doconce file using if-else preprocessor directives.

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 introduce the convention that if `TITLE:` or `#TITLE:` is found at the beginning of the line (i.e., the document has, or has an intention have, 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

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.

Strange non-English characters

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

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

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

```
Unix> doconce change_encoding utf-8 LATIN1 myfile.do.txt
Unix> iconv -f utf-8 -t LATIN1 myfile.do.txt --output newfile
```

Inline verbatim text is not formatted correctly

Make sure there is whitespace surrounding the text in backquotes.

Too short underlining of reST headlines

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

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 Output

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.

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.

Problems with LaTeX Output

The LaTeX file does not compile

If the problem is undefined control sequence involving:

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

the cause is usually a verbatim inline text (in backquotes 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.

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.

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.

A Glimpse of How to Write a New Translator

This is the HTML-specific part of the source code of the HTML translator:

```
FILENAME_EXTENSION['html'] = '.html'  # output file extension
BLANKLINE['html'] = '<p>\n'           # blank input line => new paragraph
INLINE_TAGS_SUBST['html'] = {         # from inline tags to HTML tags
    # keep math as is:
    'math': None,  # indicates no substitution
    'emphasize':     r'\g<begin><em>\g<subst></em>\g<end>',
```

```
    'bold':          r'\g<begin><b>\g<subst></b>\g<end>',
    'verbatim':      r'\g<begin><tt>\g<subst></tt>\g<end>',
    'URL':           r'\g<begin><a href="\g<url>">\g<link></a>',
    'section':       r'<h1>\g<subst></h1>',
    'subsection':    r'<h3>\g<subst></h3>',
    'subsubsection': r'<h5>\g<subst></h5>',
    'paragraph':     r'<b>\g<subst></b>. ',
    'title':         r'<title>\g<subst></title>\n<center><h1>\g<subst></h1>
    'date':          r'<center><h3>\g<subst></h3></center>',
    'author':        r'<center><h3>\g<subst></h3></center>',
    }

# how to replace code and latex blocks by html (<pre>) environment:
def html_code(filestr):
    c = re.compile(r'^!bc(.*?)\n', re.MULTILINE)
    filestr = c.sub(r'<!-- BEGIN VERBATIM BLOCK \g<1>-->\n<pre>\n', filestr
    filestr = re.sub(r'!ec\n',
                     r'</pre>\n<! -- END VERBATIM BLOCK -->\n', filestr)
    c = re.compile(r'^!bt\n', re.MULTILINE)
    filestr = c.sub(r'<pre>\n', filestr)
    filestr = re.sub(r'!et\n', r'</pre>\n', filestr)
    return filestr
CODE['html'] = html_code

# how to typeset lists and their items in html:
LIST['html'] = {
    'itemize':
    {'begin': '\n<ul>\n', 'item': '<li>', 'end': '</ul>\n\n'},
    'enumerate':
    {'begin': '\n<ol>\n', 'item': '<li>', 'end': '</ol>\n\n'},
    'description':
    {'begin': '\n<dl>\n', 'item': '<dt>%s<dd>', 'end': '</dl>\n\n'},
    }

# how to type set description lists for function arguments, return
# values, and module/class variables:
ARGLIST['html'] = {
    'parameter': '<b>argument</b>',
    'keyword': '<b>keyword argument</b>',
    'return': '<b>return value(s)</b>',
    'instance variable': '<b>instance variable</b>',
    'class variable': '<b>class variable</b>',
    'module variable': '<b>module variable</b>',
    }

# document start:
INTRO['html'] = """
<html>
<body bgcolor="white">
"""
```

```
# document ending:
OUTRO['html'] = """
</body>
</html>
"""
```

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.