Directory and file structure

Hans Petter Langtangen[1,2]

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

Oct 20, 2015

Contents

This document describes the file structure of book projects. The setup can of course be used for proceedings and theses as well.

1 Directory structure

We shall outline a directory structure that can be effective when assembling different DocOnce documents into a book:

```
doc
  src
    chapters
      ch2
        fig-ch2
        src-ch2
        mov-ch2
        exer-ch2
    book
  pub
    chapters
    book
  web
```

The root directory for all documentation is called `doc`, with two subdirectories: `src` for all the DocOnce source code, and `pub` for compiled (published) documents

in various formats. A third subdirectory, `web`, is often present as an entry point for the web pages on GitHub. This directory typically contains the autogenerated `index.html` and additional style files on GitHub. The `index.html` file should have links to published documents in `../pub`.

Under `doc/src` we may have a directory `chapters` for the individual chapters and a directory `book` for the assembly into a book. One may also think of more than one book directory if a set of documents (chapters) naturally leads to multiple books. All chapters can then be put in the `chapters` directory.

Each chapter has a short *nickname*, say `ch2` for simplicity for Chapter 2 (a more descriptive name related to the content is obviously much better!). Figures are placed in subdirectory `fig-ch2` and computer code in subdirectory `src-ch2`. These two latter directories may have subdirectories if desired. We may also include a directory `mov-ch2` for video files, `exer-ch2` for answers to exercises, etc.

Under `book`, we typically have a document `book.do.txt` for the complete book. This is a file with a lot of `# #include "...do.txt"` statements for the Preprocess preprocessor for including the files for the various chapters, see Section 3 for details. Additional files in the `book` directory include make files for compiling the book, scripts for packing the book for publishing, perhaps an errata document, etc.

2 Principles and conventions

When starting a bigger project like one or more book projects, alone or with others, it is wise to sit down and agree upon some basic principles and conventions. This note is about a technical infrastructure for writing books as an assembly of chapter components, but much more infrastructure is needed to achieve an efficient work flow in the project. One simply needs rules to make the work flow and end product coherent. This means one must agree upon

- mathematical notation
- conventions for labels in cross referencing
- programming style
- degree of modularization
- writing style
- student guide (slides) style
- style in exercises, problems, and projects

The suggestions here have grown out of the author's own experience with writing books and must be taken as just one possible example on how to deal with the bullet list above.

Observation: LaTeX-based writing styles are very private.

Most authors develop very private ways of using LaTeX in their projects. They often have a vast amount of newcommands and a collection of special LaTeX packages they rely upon. The result is that it might be quite a challenge to combine two LaTeX-based writing projects, because of all the special commands floating around.

With DocOnce and other quite simple markup languages, there are not so many ways to do it and the source code becomes much simpler and easier to integrate across projects and authors.

Note. DocOnce applies *a lot* of fancy LaTeX packages and HTML CSS styles, but the associated LaTeX/HTML code is automatically generated and steered via command-line options such that the complexity of fancy admonitions or syntax highlighting is not visible in the document the authors are writing on.

2.1 Mathematical notation and newcommands

Use a common mathematical notation!

I strongly recommend to spend considerable time on constructing a set of newcommands in LaTeX that defines a *common* mathematical notation for the project (and future projects). Think about newcommands for vectors (arrows or boldface), matrices (uppercase slanted or bold?, tensors, gradient, divergence, curl, etc.

Files with names `newcommands*.tex` are treated by DocOnce as files with definition of newcommands for LaTeX mathematics. These files must reside in the same directory as the DocOnce source files. However, for a book project, I recommend to have a single newcommands file shared by all chapters. This file is placed in `doc/src/chapters/newcommands.p.tex` and copied to a specific chapter by the make script for that chapter. The extension of the file is `.p.tex`, indicating that the file has to be *preprocessed* by `preprocess` prior to being copied. The reason is that one occasionally wants the definitions of the newcommands to depend on the output format: standard LaTeX or MathJax. For example, subscripts in `mbox` font look best with footnotesize font in plain LaTeX, while the larger `small` font is more appropriate for MathJax. We can then put the following definitions in `newcommands.p.tex`:

```
% #if FORMAT in ("latex", "pdflatex")
% Use footnotesize in subscripts
\newcommand{\subsc}[2]{#1_{\mbox{\footnotesize #2}}}
% #else
% In MathJax, a different construction is used
```

```
\newcommand{\subsc}[2]{#1_{\small\mbox{#2}}}
% #endif
```

The make script will then run `preprocess` on this file, typically

```
preprocess -DFORMAT=pdflatex ../newcommands.p.tex > newcommands.tex
# or
preprocess -DFORMAT=html ../newcommands.p.tex > newcommands.tex
```

DocOnce newcommands are for mathematics only!

Note that newcommands in DocOnce context are only used for mathematics, rendered by LaTeX or MathJax. Newcommands for other LaTeX constructions (such as section or boxes) should not be used in the DocOnce source code as these are confined to the LaTeX format. Use instead Mako functions.

Here is an example on some useful constructs in a `newcommands.p.tex` file:

```
\newcommand{\halfi}{{1/2}}
\newcommand{\half}{\frac{1}{2}}
\newcommand{\tp}{\thinspace .}  % right space after equations

% Operators
\newcommand{\Ddt}[1]{\frac{D #1}{dt}}
\newcommand{\E}[1]{\hbox{E}\lbrack #1 \rbrack}
\newcommand{\Var}[1]{\hbox{Var}\lbrack #1 \rbrack}
\newcommand{\Std}[1]{\hbox{Std}\lbrack #1 \rbrack}
\newcommand{\Oof}[1]{\mathcal{O}(#1)}

% Boldface vectors/tensors
\newcommand{\x}{\bm{x}}
\newcommand{\X}{\bm{X}}
\renewcommand{\u}{\bm{u}}
\renewcommand{\v}{\bm{v}}
\newcommand{\w}{\bm{w}}
\newcommand{\V}{\bm{V}}
\newcommand{\e}{\bm{e}}
\newcommand{\f}{\bm{f}}
\newcommand{\F}{\bm{F}}
\newcommand{\stress}{\bm{\sigma}}
\newcommand{\strain}{\bm{\varepsilon}}
\newcommand{\stressc}{{\sigma}}
\newcommand{\strainc}{{\varepsilon}}
\newcommand{\normalvec}{\bm{n}}

% Unit vectors
\newcommand{\ii}{\bm{i}}
\newcommand{\jj}{\bm{j}}
\newcommand{\kk}{\bm{k}}
\newcommand{\ir}{\bm{i}_r}
\newcommand{\ith}{\bm{i}_{\theta}}
\newcommand{\iz}{\bm{i}_z}

% Number sets
\newcommand{\Real}{\mathbb{R}}
```

```
\newcommand{\Integerp}{\mathbb{N}}
\newcommand{\Integer}{\mathbb{Z}}
```

2.2 Label conventions

Books usually contain large amounts of cross referencing using labels (logical names) for sections, equations, exercises, and literature references. I strongly recommend to introduce conventions for how to construct labels as it makes it much easier to find the name of a new label and understand what a given label refers to.

My chapter names are of the form `ch:name`, where `name` is the nickname of the chapter (this nickname is used for many other purposes also, see Section 3.1). Section and subsection names are of the form

```
projectname:chaptername:sectionname:subsectionname
```

where the names are short nicknames. Each short name may contain underscores to separate words. For example, the present section has label `setup:rules:conv:labels`, where `setup` is the nickname of the project (this book), `rules` is the nickname of the present chapter, `conv` is the nickname of the present section, and `label` is the nickname of the subsection. Section 3.1 has label `setup:rules:book_assembly:org`, where `book_assembly` is the nickname of the section using underscore to separate two words. Sometimes I leave out the project name.

Equations may start with the current subsection label and continued with `eq:name`, where `name` is some logical name for the equation. Figures are given the same type of name, except that the postfix is `fig:name`. For exercises I use `exer:name` as postfix in the labels. Equations and figures within an exercise have extended names such as `setup:rules:exer:eq:uv_relation`.

Bibliographic references can take the forms `author_year`, `author1_author2_year`, `author1_etal_year`. An alternative is to use names that reflect the topic: `topic:paper` or `topic:url`, e.g., `IPython:paper` for a]paper on IPython or `IPython:url` for the IPython website.

You can run `doconce list_labels *.do.txt` to get a list of labels, categorized under the various section headings, in your DocOnce documents. This command quickly reveals if a clean-up of label names is necessary. You can redirect the output of the command to a file and then add a second column with new label names. This file can be used with the command `doconce replace_from_file` to perform substitutions from old to new labels.

Compilation of DocOnce files can make use of the option `--latex_labels_in_margin` to get the label names of equations and sections to be printed in the margin.

2.3 Programming style

It is fundamental for a book project to stick to one well-defined programming style. I recommend to adopt the most widely accepted style and adapt that to the project. For example, in the world of Python programming, there is a style,

referred to as PEP8. Personally, I am not fond of all the rules in this style, and I intentionally break some of them, especially rules that forces unnecessary vertical space in a book (although vertical space in electronic books is for free, there is a strong tradition of minimizing the vertical length of programs in books). Fortunately, for Python there is are nice tools for checking that a code follows the PEP8 rules, e.g., Flake8 (see [1] for an intro).

For any programming language it is key to agree on how to use white space in indentation, style of loops, identifier names (`my_funtcion` vs `myFunction` vs `MyFunction`), white space in function argument lists, etc.

Tip: make a 1-1 mapping between mathematics and code.

Computer code is very much easier to understand if you have defined the problem it is going to solve in mathematics first. Reuse terms and notation in the program, and try to make the key statements in the code as close as possible to the mathematical formulation.

Typesetting of computer code in DocOnce makes use of the `!bc` construction and a named environment, e.g., `pycod` for a Python snippet and `pypro` for a complete Python program. Users are often confused if a set of statements in a text can be executed as they stand or not. That is why we have introduced the `cod` and `pro` environments: `cod` is for just some code, while `pro` is for a complete program that will run. You may choose the typesetting to be different for the `cod` and `pro` environments.

When preparing text for IPython/Jupyter notebooks, a code snippet cannot run unless previous snippets contain the necessary code. Sometimes this forces you to include more code than would be natural in a book. There is a third type of environment, `hid`, that can be used to insert code segments that are to be hidden in text, but present in notebooks to enable execution of future snippets. For example,

```
# Need to import to run next snippet
!bc pyhid
from numpy import *
from matplotlib.pyplot import *
!ec

We can now generate coordinates by

!bc pycod
x = linspace(0, 1, 101)
y = sin(x)
plot(x, y)
!ec
```

The import statements will only be visible in the notebook output and not in any other format.

2.4 Degree of modularization

My recommendation is to divide the project into as small modules as possible and to make these modules as independent as possible. This is a very difficult optimization problem. There is some kind of gravity force towards big chapters and lots of cross-references internally and to other chapters. For book composition and even more for course composition, smaller modules give much higher degree of flexibility.

To make modules independent, the degree of cross-referencing between modules must be modest, which forces a need to repeat material. Repetition breaks a strong tradition in book writing. However, moving away from a strictly linear chapter-by-chapter book to a more flexible set of modules connected in a graph, will induce repetition. Readers also appreciate repetition, perhaps slightly differently phrased with purpose, rather than being served with lots of references to equations and codes in other chapters.

Tip: Define input-output of modules.

Ideally, modules should start with a well-defined list of required background knowledge and a set of learning outcomes. This information makes it easier to place the module in the knowledge landscape.

Books with widely different writing styles among authors tend to be confusing for readers. If the styles differ much, and it is difficult to converge to a more coherent style, listing the authors together with the chapter title is an idea to point explicitly out that a different team is behind the present chapter.

2.5 Student guide (slides) style

This note suggests to develop a book together with a *study guide*, i.e., a summarizing version of the material. An effective format of a study guide is a set of slides, ideally a set that can be used both for teaching and for self-study. Section 6 explains some infrastructure for producing DocOnce slides.

Some prefer to develop slides for a study guide first and then use this as a skeleton for writing the running text of a chapter. Others prefer to produce the slides from the running text.

The style of slides is even more important than the style of running text. Slides for reading are not so sensitive to the style, but if the slides are also intended to be used in teaching, the style becomes key. Some comments on style are provided in the box in the introduction to Section 6. Rather than having boring headings a la *Assumptions*, followed by a bullet list of assumptions, I recommend to summarize the most important information in a 1-2 line heading, e.g., *We assume a homogeneous material and no external forcing*. The headings will then form a collection of the most important information from each slide

and be a very effective table of contents of the material. The most important thing, though, is that different authors stick to the same slide style.

2.6 Style in exercises, problems, and projects

A central part of the writing style in a book is the division of material between running text and exercises. DocOnce features three types of exercises which can be effectively used in this context:

- Exercise: smaller exercises tightly coupled to the text
- Problem: smaller exercise that live its own life (without references to the text)
- Project: large problem

Exercises are then used to repeat and train the material in the book. Problems are used to explore new problem settings, while projects are collections of closely related problems. (The term "exercise" means in DocOnce context either an exercise in the restricted sense or a common term for what we call exercise, problem, and project above.)

The DocOnce exercise format has several useful features:

- hints
- multiple-choice questions (quiz)
- remarks
- short answers
- (longer) complete solutions

Many students struggle with identifying the problem setting (question) when too much information comes at once. Hints can be effectively used to separate the question from additional information that is just supposed to help the reader. Remarks fulfill a similar purpose and can separate fun facts or information that puts the problem into a wider perspective. Short answers and full solutions can be taken out of the document at compile time. (HTML Bootstrap styles can fold/unfold hints and solutions/answers.)

Multiple-choice questions are typeset with the quiz environment in DocOnce. All quizzes can be extracted and uploaded as online Kahoot games where students can participate via their smart phones.

It is possible to extract all exercises in a DocOnce document as a separate document.

3 Assembling different pieces to a book

Many smaller writings in the DocOnce format can be assembled into a single, large document such as a book or thesis. The recipe for doing this appears below.

3.1 Organization of a chapter

Suppose one chapter of the book has the nickname `ch2` and may hold all text or just include text in other DocOnce files, e.g., `part1.do.txt`, `part2.do.txt`, and `part3.do.txt`. In this latter case, `ch2.do.txt` has the simple content

```
# #include "part1.do.txt"

# #include "part2.do.txt"

# #include "part3.do.txt"
```

Note that the `ch2.do.txt` file contains just plain text without any `TITLE`, `AUTHOR`, or `DATE` lines and without any table of contents (`TOC`) and bibliography (`BIBITEM`). This property makes `ch2.do.txt` suitable for being included in other documents like a book. However, to compile `ch2.do.txt` to a stand-alone document, we normally want a title, an author, a date, and perhaps a table of contents. We also want a bibliography if any of the included files have `cite` tags. To this end, we create a wrapper file, say `main_ch2.do.txt`[1], with the content

```
TITLE: Some chapter title
AUTHOR: A. Name Email:somename@someplace.net at Institute One
AUTHOR: A. Two at Institute One & Institute Two
DATE: today

TOC: on

# Externaldocuments: ../ch3/main_ch3, ../ch4/main_ch4

# #include "ch2.do.txt"

======= References =======

BIBFILE: ../papers.pub
```

Recall that DocOnce relies on the Publish software for handling bibliographies. It is easy to import from BibTeX to Publish and create a database of references (`papers.pub`) to get started (but we recommend to continue working with the Publish database directly and collect new items in the `papers.pub` file as Publish is more flexible than BibTeX).

[1] The prefix `main_` is inspired by the main program in computer program: those statements make a program run, like `main_ch2.do.txt` defines the surroundings of the "library text" `ch2.do.txt`. We strip off `main_` when publishing the files in `doc/pub`.

3.2 Figures and source code

As described in Section 1, we recommend to put figures and source codes (to be included in the document) in separate directories. Although such directories could have natural names like `fig` and `src`, it will cause trouble if we do not use unique names for these directories, like `fig-ch2` and `src-ch2`. Otherwise, we would need to copy all figures in all pieces into a common `fig` directory for the book and all source code files into a `src` directory. With unique names, figures and source code files can always reside in their original locations, and we can easily reach them through links. This will be described next.

3.3 Assembly of chapters to a book

All the files associated with the `ch2` document and chapter reside in the `ch2` directory. A fundamental principle of DocOnce is to have just one copy of the files ("document once!"). To include the `ch2` text in a larger document like a book, we just need to include the `ch2.do.txt` file and a chapter heading. Here is an example of a document `book.do.txt` for a complete book:

```
TITLE: This is a book title
AUTHOR: A. Name Email:somename@someplace.net at Institute One
AUTHOR: A. Two at Institute One & Institute Two
DATE: today

TOC: on

========= Preface =========
label{ch:preface}

# #include "../chapters/preface/preface.do.txt"

========= Heading of a chapter =========
label{ch:ch2}

# #include "../chapters/ch2/ch2.do.txt"

# Similar inclusion of other chapters

========= Appendix: Heading of an appendix =========
label{ch:somename}

# #include "../chapters/nickname/nickname.do.txt"

======= References =======

BIBFILE: ../papers.pub
```

When running `doconce format` on `book.do.txt`, the entire document is contained in *one* big file[2] (!). To see exactly what has been included, you can examine the result of running the first preprocessor, `preprocess`, on `book.do.txt`. All the includes are handled by this preprocessor. The result is contained in the file `tmp_preprocess__book.do.txt`, which then contains the entire DocOnce

[2] A single DocOnce file and consequently a single `.tex` file works out well on today's laptops. A book with 900 pages [4] has been tested!

source code of the book. The second preprocessor, `mako`, is thereafter run (if DocOnce detects that it is necessary). The result of that step is available in `tmp_mako__book.do.txt`. It is important to examine this file if there are problems with Mako variables or functions. The `tmp_mako__book.do.txt` file is thereafter translated to the desired output format.

Say we want to produce a LaTeX document:

Terminal

```
Terminal> doconce format pdflatex book [options]
```

If the DocOnce source contains copying of source code from files in `@@@CODE` constructs, it is important that `doconce` finds the files. For example,

```
@@@CODE src-ch2/myprog.py  fromto: def test1@def test2
```

will try to open the file `src-ch2/myprog.py`. Since this file is actually located in `../ch2/src-ch2/myprog.py`, `pdflatex` will report an error message. A local link to that directory resolves the problem:

Terminal

```
Terminal> ln -s ../chapters/ch2/src-ch2 src-ch2
```

Similarly, the LaTeX code in `book.tex` for inclusion of a figure may contain

```
\includegraphics[width=0.9\linewith]{fig-ch2/fig1.pdf}
```

For this command to work, it is paramount that there is a link `fig-ch2` in the present `book` directory where the `pdflatex` command is run to the directory `../chapters/ch2/fig-ch2` where the figure file `fig1.pdf` is located.

It is recommended to use the function `make_links` in `scripts.py` to automatically set up all convenient links from the `book` directory to the individual chapter directories. Provided the *list of chapter nicknames* at the top of `scripts.py` *is correct*, you can just run

```
>>> import scripts
>>> scripts.make_links()
```

to automatically set up all links to all `src-*`, `fig-*`, and `mov-*` directories. You need to rerun this `make_links` function after inclusion of a new chapter in the `chapters` tree.

Identify LaTeX errors in the original chapter files!

When you run `pdflatex book` and get LaTeX errors, you need to see where they are in `book.tex` and use this information to find the appropriate DocOnce source file in some chapter. Usually, there are few errors at the

"book level" if each individual chapter has been compiled. To this end, you can use `scripts.py` to automatically compile each chapter separately. The process is stopped as soon as a DocOnce or LaTeX error is encountered.

```
>>> import scripts
>>> scripts.compile_chapters()
```

With heavy use of Mako one can get quite strange error messages. Some ask you to rerun the `doconce format` command with `--mako_strict_undefined` to see undefined Mako variables. Make sure you run the `make.sh` script by `bash -x` if the script does not feature the `set -x` command in the top of the file (for displaying a command prior to running it). Copy the complete `doconce format` with the mouse and add the `--mako_strict_undefined` option. Other error messages point to specific lines that Mako struggles with. Go to the file `tmp_mako__book.do.txt` to investigate the line.

4 Tools

You can start a new, future, potential book project by simply copying the directory structure of the setup4book-doconce repository on GitHub. Then you can follow the instructions below to start writing and adapting the structure to your project's needs.

4.1 Making a new chapter

Under `doc/src/chapters` you find the chapters in this "sample book" as well as a script `doc/chapters/mkdir.sh` that creates a new directory for you with the typical files needed for a new chapter. You can either edit existing chapters, or make a brand new empty chapter by running

Terminal

```
Terminal> sh mkdir.sh mychap
```

This command makes a directory `mychap` for a new chapter with nickname `mychap`. Files from the `template` directory are used to populate `mychap`. You get an empty `mychap.do.txt` where the text is supposed to go, or this file can just include a series of smaller `.do.txt` files, and you get the wrapper file `main_mychap.do.txt` such that you can compile this chapter as a stand-alone document. You also get `make.sh` which calls `../make.sh` with the chapter main document (`main_mychap`) as argument. Optional arguments for running `doconce format pdflatex` can be given to `../make.sh` in `make.sh` if needed (e.g., `-encoding=utf8`).

4.2 Compiling the chapter to LaTeX and PDF

To make a stand-alone document of a chapter, by compiling to LaTeX and PDF, we propose the convention to have a `make.sh` in each chapter directory. This `make.sh` can in most cases just call up a common `../make.sh` script,

```
bash -x ../make.sh main_mychap
```

or optionally with some command-line arguments,

```
bash -x ../make.sh main_mychap --encoding=utf-8
```

The `doc/src/chapters/make.sh` script is quite general and may be edited according to your layout preferences of the LaTeX documents.

The present `make.sh` script creates two PDF files: one for printing and one for electronic viewing. The difference is that all URLs in the version for printing appear as footnotes (and just hyperlinks with a dark blue color in the electronic version). The two files are named `mychap.pdf` and `mychap-4print.pdf`, respectively, and copied to `doc/pub/mychap/pdf` for publishing.

Tip: use tinyurl.com for shortening long URLs.

When compiling a document to LaTeX for *printing on paper* (`-device=paper`), URLs in hyperlinks will appear as footnotes. Very long URLs may then exceed the line width, or worse, extend beyond the physical paper size. Replace such long URLs with short forms using tinyurl.com. I recommend tinyurl.com rather than competitors like `goo.gl` and `bit.ly` because if you have some URL `http://tinyurl.com/oul3xhn` you can easily add more to the path, e.g., `http://tinyurl.com/oul3xhn/index.html`, and this new URL works (`goo.gl` and `bit.ly` do not allow such extensions).

In particular, you can define the tinyurl.com URL as a Mako variable (see `doc/mako_code.txt` for example) and use it as a quick and logical name in the text for the URL and extend its path as appropriate. For example, I always have a Mako variable for the URL of the repository directory for the software associated with a chapter and can then easily add `/myprog.py` to the variable to create a link to the file at GitHub. Readers can then just click to read or download the file.

Remark about LaTeX typesetting of computer code. The suggested `make.sh` file applies the `--latex_code_style=` option to `doconce format` for specifying the typesetting of blocks of computer code in LaTeX. Originally, DocOnce applied the `ptex2tex` program to select such typesetting, but the new method is more flexible and simpler in that it gives cleaner LaTeX code. (With `ptex2tex` one would need a common configuration file `.ptex2tex.cfg` in `doc/chapters` to be copied by `doc/chapters/make.sh` to the chapter directory prior to running 'ptex2tex.)

Cleaning Files. The `make*.sh` files generate a lot of files that can easily be regenerated and that are normally removed from the chapter directories. The script `sh ../clean.sh` can be run in any chapter directory to clean up redundant files.

4.3 Automatic spell checking

The `make.sh` first runs a spell check using `doconce spellcheck`. The first time you run this command there will be many "misspellings" because of unregistered (scientific) words in the dictionary and maybe also because some words are actually misspelled. Invoke the `misspellings.txt` file to see a list of all misspellings. Correct mistakes in the original documents and run the `make.sh` script again. When `misspellings.txt` at some point contains acceptable words only, you update the dictionary by

```
cp new_dictionary.txt~ .dict4spell.txt
```

Make sure `.dict4spell.txt` is version controlled by Git. The `make.sh` script will not proceed with compilation of the documents before the spell check is run without errors.

Finding misspellings can sometimes be a challenge. For a document named `mydoc.do.txt`, the spell check is carried out on a stripped version, named `tmp_stripped_mydoc.do.txt`. Look into this file for misspellings that are not obvious. Strange misspellings such as `APlu` or similar usually arise from missing dollar sign around mathematical formulas. Formulas are stripped away in `tmp_stripped_mydoc.do.txt`, but if a dollar sign is missing, mathematical formulas become words subject to spell checking. To find the relevant file containing a particular misspelling listed in `misspellings.txt` , you may look into the file-wise list of misspellings: the misspellings in `mydoc.do.txt` are listed in `tmp_misspelled_mydoc.do.txt~`.

4.4 Compiling the chapter to HTML

There is also a script `doc/src/chapters/make_html.sh` for making HTML versions of the chapter. Just call this as

Terminal

```
Terminal> bash ../make_html.sh main_mychap
```

to make HTML versions of the `mychap` chapter.

The current version of `make_html.sh` creates four types of HTML layouts and an `index.html` file with a list of links to these three files: 1) HTML plain Bootstrap style, 2) HTML Bootswatch readable style, 3) plain HTML solarized color style, and 4) Sphinx pyramid style. (Note that the latter document is a true Sphinx document, made by `doconce format sphinx`, and from which one could make other formats too.)

It is easy to go into the `make_html.sh` script and generate other HTML or Sphinx styles.

You need to edit the index file!

The `index.html` file generated by `make_html.sh` is made from the DocOnce source file `index_html_files.do.txt`. This is a file utilizing Mako programming (see [3]). There is also a similar file, `index_files.do.txt`, listing all the published documents in various formats associated with a complete book projects (to go to `doc/pub/index.html`).

The `index_html_files.do.txt` and `index_files.do.txt` files rely much on a Mako dictionary `chapters`, defined in `mako_code.txt`. This dictionary maps nicknames to chapter titles. We can then specify a nickname and easily get the corresponding full chapter title. For example, in `index_files.do.txt` we defined a Mako list `published` holding the nicknames of the chapters we want to publish. With a Mako for loop we can then run through these selected chapters and generate the corresponding DocOnce lists with all the formats that is offered for a chapter and its associated slides. This is a nice example on how a potentially quite large DocOnce document with much repetitive constructions can be written with very compact code.

One can imagine that for a large books under constant development with different states of different chapters, this setup makes it easy to take chapters in and out of the book. In addition, with Mako variables in the chapters one can easily defined different state of maturity of the text. With minor Mako programming in `index_files.do.txt` and extension of the `make*.sh` files, authors can generate the various states of the book, e.g., a quality controlled version approved for students and a complete "work-in-progress" version for authors only with all available text and lots of DocOnce square-bracket comments.

The `index_files.do.txt` file gives a table of contents of all documents, so you will normally compile this manually now and then as

Terminal

```
Terminal> doconce format html index_files \
          --html_style=bootstrap \
          --html_links_in_new_window \
          --html_bootstrap_navbar=off
```

and publish it in `doc/pub/index.html`.

4.5 Compiling the chapter to a notebook

Although there is no benefit from interactive computing and visualization in the present document, we may produce an IPython notebook for the fun of it:

Terminal

```
Terminal> doconce format ipynb main_rules \
          CHAPTER=document BOOK=document APPENDIX=document
```

4.6 About figures when publishing HTML

There will be `<img src=fig-ch2/fig1.png>` type of tags in HTML code produced by DocOnce, so it is very important to ensure that the *published* `.html` files have access to a subdirectory `fig-ch2`. Normally, one needs to copy `fig-ch2` from the `ch2` chapter source directory to some publishing directory that stores all the files necessary for accessing the entire HTML document on the web.

4.7 Compiling the book

Go to `doc/src/book` and run `make.sh` to compile the book. This requires that `book.do.txt` performs the right include of chapters, table of contents, and bibliography.

There are many other tools in `doc/src/book` too, e.g., the mentioned library of handy scripts in `scripts.py`, and an example on how to pack all files of the entire book projects for publishing with Springer (`pack_Springer.sh`).

The current book layout created by `make.sh` makes use of a (now outdated) Springer T4 style for textbooks (requires the `.cls` and `.sty` files in the `book` directory). Other Springer styles supported by DocOnce are Lecture Notes in Computational Science and Engineering (monographs and proceedings), Lecture Notes in Computer Science (proceedings), and Undergraduate Texts in Physics. Other book styles will require some manual work, either working out a LaTeX preamble for a special style and use that when compiling `book.do.txt` or actually extending the DocOnce source code.

HTML/Sphinx versions of the book. It is easy to make a standard HTML version of the `book.do.txt` manuscript, but for large books, Sphinx is usually a better alternative since it supports navigation, searching, and has an index. There is a script `doc/src/book/make_html.sh` that creates a Sphinx version of the book. Actually, it generates two versions

- standard Sphinx book
- RunestoneInteractive Sphinx book

5 Cross-referencing across chapters (or books)

A fundamental problem when writing a book *and* stand-alone chapters arises with cross-referencing. In a book file it makes sense to refer to an equation in any chapter, say (4.23), while in a stand-alone chapter references to equations or sections in other stand-alone documents will not work. That is, LaTeX has a native mechanism for this, the `xr` package, where one can register a set of `.aux` files for other LaTeX documents and refer directly to these labels and get them right. It is then possible to write something like

```
see (\eqref{sec:results:u:eq}) in \cite{Hansen_2011b}
```

and get it out as

```
see (2.37) in [12]
```

provided our `.tex` file contains `\externaldocument{myother}` and the label `sec:results:u:eq` is defined in `myother.aux`. DocOnce has generalized this feature so it works for non-LaTeX formats as well. It is called *generalized cross-references*. You can then write such references across chapters and get all labels right whether you produce the entire book or individual chapters.

Here is an example on a generalized reference to an equation in another document:

```
The world's most famous equation is ref[(ref{setup:fake:Emc2})][ in
cite{Langtangen_dobook_fake}][
as found in the document "Some document":
"http://hplgit.github.io/setup4book-doconce/doc/pub/fake"
cite{Langtangen_dobook_fake}].
```

This sentence is rendered as follows in the present format (`pdflatex`):

> The world's most famous equation is (1)in [2].

More detailed information about generalized cross-references is found in the DocOnce manual. In particular, one has to insert `# Externaldocuments:` commands in all `main_*.do.txt` files that includes files with generalized references.

Tip: Limit generalized references to those strictly needed.

Books often contain a lot of cross references, and making generalized references out of all them can be quite some job. A convenient way of saving boring work is to enclose nice-to-have, yet not strictly needed, references in Mako or Preprocess if statements (typically `if BOOK == "book"`) such that they appear in the full book but not in individual chapters.

However, if individual chapters in HTML are to be one official format of the book, you should make the chapters identical to the book and make generalized references out of all references to other chapters.

6 Study guides and slides

DocOnce has good support for creating slides. Especially if you have ordinary DocOnce documents with running text, it is an efficient process to strip down this text to a slide format.

Rather than speaking about slides, we think of *study guides* where the material is presented in a very condensed, effective, summarizing form for overview, use in lectures, and repetition. The slide format is a good way of writing study guides, but by explicitly thinking of study guides the slide format can be made more effective for self-study when overview and repetition are necessary - with a particular emphasis on gaining understanding.

Slides can easily be too crowded or too empty.

It is a very challenging balance between enough information for self-study by reading slides and the modest amount of information you want in slides for oral presentations. For a talk, you will have (very) little text on slides and rely on figures. This is not so effective in a teaching and study guide setting. Some text is indeed necessary, but it has to be minimized. Michael Alley's evidence-assertion slide design is effective: summarize the slide's key point in a heading over 1-2 lines, use figures/equations/code effectively, and work on minimizing text.

Make it an assumption that the reader of a study guide is also a reader of the underlying running text in the chapter.

6.1 Slide directory

For each DoOnce file in the chapter `ch2` it can be wise to make a corresponding study guide file in the subdirectory `slides-ch2`. For example, `part1.do.txt` has its counterpart with slides in `slides-ch2/part1.do.txt`. Then there is a file `slides_ch2.do.txt` which assembles the parts if `slides-ch2`, typically with a content like

```
TITLE: Study Guide: Some title
AUTHOR: Author Name Email:somename@someplace.net at Institute One
DATE: today

# #ifdef WITH_TOC
!split
TOC: on
```

```
# #endif
# #include "lec-ch2/part1.do.txt"
# #include "lec-ch2/part2.do.txt"
# #include "lec-ch2/part3.do.txt"
```

6.2 Generating slides from running text

The author has the following work flow for generating slides for a chapter file, say `part1.do.txt`.

1. Copy `part1.do.txt` to `slides-ch2/part1.do.txt`.
2. Make `slides_ch2.do.txt` and include `slides-ch2/part1.do.txt`.
3. Decide on *parts* of the slide collection. Often a part can be a section in the parent `ch2.do.txt` file, but sometimes it can be more natural to have larger parts than sections in the slide collection.
4. Each part in the slide file has a DocOnce section heading with 7 `=`, while each slide has a DocOnce subsection heading with 5 `=`.
5. Edit `slides-ch2/part1.do.txt`:
   - One can keep subsection headings from the running text for the most part, but slides need many more subsection headings.
   - Try to let the heading summarize explicitly a conclusion/rule from the slide (the slide table of contents is then a set of conclusions/rules!)
   - Remember a `!split` right above every slide heading!
   - Compile frequently and look at the slides: they become over-full very quickly so there is a constant need for dividing slides into new ones with new headings.
   - Read a paragraph, focus on its main idea and result, and see how it can be condensed to one sentence or a few bullet points. *Making effective slides is the art of condensing the most important information in the text to a eye-catching format.*
   - Do not remove figures without a very good reason. Figures are important!
   - Add new images to liven up the presentation. In slides you may think of cartoons or entertaining images that would never be suitable in a chapter/book, but they may help attract attention, communicate ideas, and enhance the memory process.
   - Condense every mathematical derivation. Make sure the goal and end result is clear before diving into details.

- Detailed derivations are seldom of interest in a study guide or oral presentation - refer to the underlying running text in the chapter for the details. Focus on ideas and key mathematical steps (if they are important enough).
- Remember that equations are sometimes excellent images for ideas! Complicated equations can therefore be important slide elements although the details will never be addressed.
- It is quite often wise to remove equation numbers in slides. You can automatically remove them by `--denumber_all_equations`, or you can edit the LaTeX math environment manually. Remember that references to equations numbers must be removed from the slides too!
- Movies are effective in slides. It is still a hassle to get them displayed correctly in PDF files, so using a test on `FORMAT` and writing `MOVIE` for HTML output and just a link in PDF output might be necessary. See the manual for how to work with movies in DocOnce.

The slides are to fulfill three purposes:

1. reading as a study guide to get overview before reading the full text of chapter,
2. watching as slides during an oral presentation,
3. reading as a study guide to repeat and enforce overview of the material.

It is highly non-trivial to meet all these purposes: limit the information on the slides, make them as visual as *feasible*, make them self contained, and provide the *sufficient* amount of information. Considerable iterations are always needed. Reading the slides as a study guide is easy to accomplish. The slides' properties in live presentations can only be tested by speaking to them (making a rough draft of a video podcast is a very effective way of testing the slides' quality).

Tip: use quizzes to define a sufficient preparation level.

You want students to study the slides/study guide before a lecture. To measure to what extent this is done, you can insert multiple-choice questions about the most basic concepts in the slides (using the DocOnce `quiz` environment). With quiztools you can extract all such multiple-choice questions, create online games with Kahoot, and let the students answer with their smart phones at the beginning of a lecture. The scores are visible to all on the main screen and communicate the preparation level.

6.3 Slides as IPython/Jupyter notebooks

I would add a fourth requirement to the list in the previous section: a study guide should also be available as an IPython/Jupyter notebook for experimentation, extension, and personal notes. This is technically straightforward by just generating a notebook from the slide source, but a notebook puts some constraints on code snippets and figures such that it is meaningful to execute all the code. Moreover, many figures are inlined and appear as a result of executing code in a notebook. While other formats will show a code snippet and then the corresponding figure, the notebook can leave the figure out and let it appear as the code cell is executed. Technically in DocOnce, this is solved by putting a `FIGURE` construction inside an `# #if FORMAT` test (or `% if FORMAT` if Mako branches are preferred). If 'FORMAT != 'ipynb", you have a `FIGURE` line, otherwise the preceding code cell is supposed to generate the figure.

Notebook from a chapter or from slides?

The book's running text can also be converted to a notebook. However, the notebook then consists of very much text and often a lot of cross-referencing because this is the typically writing style of a book chapter. This style is not so effective for a notebook. Stripped text with focus on formulas, code, and figures is more ideal for a notebook and this is the style of a study guide realized by slides.

6.4 Compiling slides

There is a quite general script in `doc/src/chapters/make_slides.sh` for compiling a slide collection defined in a file like `slides_ch2.do.txt`. Just run

Terminal

```
Terminal> bash ../make_slides.sh slides_ch2
```

from the chapter directory. Note that the script will first spell check the slide files. This is done in the `slides-ch2` directory. Errors are reported in files located in `slides-ch2`. To update the chapter's dictionary for spell checking, you need to do

Terminal

```
Terminal> cp slides-ch2/new_dictionary.txt~ .dict4spell
```

in the `ch2` chapter directory.

Similarly, to look at misspellings, the file `slides-ch2/misspellings.txt` is the relevant file.

The `make_slides.sh` script compiles a variety of slides:

- First a plain LaTeX PDF document to catch as many errors in the DocOnce source as early as possible. This document can also be used for compact printing of the contents of the study guide (and the output looks definitely like a study guide and not slides!).
- HTML5 `reveal.js` slides with different colors.
- HTML5 `deck.js` slides. This format is usually inferior to `reveal.js`, but is also very much personal taste.
- LaTeX Beamer slides. Edit the `theme=red_shadow` line in `make_slides.sh` to control the Beamer theme.
- Remark (Markdown) slides for viewing in a browser.

6.5 IPython/Jupyter notebooks

Since DocOnce documents can be translated to IPython/Jupyter notebooks, hereafter just called notebooks, it is tempting to produce a version of the teaching material also in notebook form. This author's experience is that a more traditional book format with running text is not so ideal for a notebook:

- you simply get too much text in a too long notebook,
- the notebook needs more code snippets than what you want to show in a book (or you just want to show fragments while the notebook requires complete code),
- there are many cross-references between equations, sections, figures, and running text that the notebook does not support well.

Instead, making slides from the chapter's text and translating slides to the notebook format is a splendid idea. This requires some tuning, as you want slight differences between classic slides and a notebook. For example, a code snippet that results with a plot should contain the plot in classic slides, while the notebook will automatically produce it when run. This is easily fixed by an if test in Mako, typically `% if FORMAT != 'ipynb':` followed by a `FIGURE:` line that includes the resulting figure for all formats except the notebook.

Also be aware of the DocOnce *hidden* code environment that can be used to declare code blocks that appear in notebooks (because they are needed) but not in other formats: `!bc pyhid` gives a Python hidden snippet.

Using notebooks as a starting point for a traditional textbook might be a good idea, but will enforce a non-conventional style in the textbook. For example, notebooks should be quite small, leading to similarly small modules in the book. Notebooks use cross-referencing to little extent, and this will be reflected in the textbook too. Notebooks also need more code to run, so one has to accept more code in the textbook. However, there is still a problem for the notebook with defining items for an index, fancy admonitions, and other elements that one

would desire in a textbook. More experience is needed to make best practices. Since notebooks can be compiled in Markdown, and DocOnce can read basic Markdown input, it is possible to go from the notebook format to DocOnce, but this is not tested.

Remark.

More best practices for turning teaching material into books and into notebooks are supposed to be collected here in the future.

7 Writing in private repository while publishing in public

Sometimes you want to keep ongoing writing in a *private* repository and make only *selected* chapters and/or files publicly visible. In such cases one can set up the book project structure in a private repository, but use a public repository instead of the `doc/pub` directory for publishing selected compiled documents. This is easy: just change the `dest=` line, where the publishing directory is defined, in all `make*.sh` scripts in `doc/src/chapters`. The files will then be copied to this alternative destination.

Often, you want to publish the software associated with the book project, stored in `doc/src/chapter/nickname/src-nickname`, as a part of the public repository. Such files can also easily be copied, say to `src/nickname` in the public repository. However, software files often change names and locations in subdirectories, and then you need to be very careful with updating the Git commands in the public repository every time you do `git add` or `git rm` locally in the private repository. This problem occurs with text files too, but maybe less often, so the recipe given below applies to all kind of files you want to mirror from a private to a public repository.

We have made a script `rsync_git.py` that can copy files from one repository to another and log files that are removed or deleted and then take the appropriate Git actions. Running

Terminal

```
Terminal> rsync_git.py src-mychap $HOME/repos/pub/mybook/src/mychap
```

will copy all files from `src-mychap` to `$HOME/repos/pub/mybook/src/mychap`, find which files that are new in `src-mychap` and must be added to the destination directory, and which files that are removed in `src-mychap` and should be removed in the destination directory as well. An `rsync` command is run to the physical copy and removal of files, followed by `git add` and `git rm` commands. In this

way, you can automatically keep the public repository as a mirror *of parts* of your private repository![3]

The `rsync_git.py` script is listed below for reference. Note that a file `$HOME/.rsyncexclude` can be made to filter out certain files that you never want to copy (this is always a good idea!).

```
#!/usr/bin/env python
"""
Sync two directory trees with rsync and perform corresponding
git operations (add or rm).
Skip files listed in $HOME/.rsyncexclude.

Usage:   rsync_git.py from-dir to-dir
Example: rsync_git.py src-mychap $HOME/repos/pub/mybook/src/mychap

The from-dir is the source and the to-dir is the destination
(e.g. a public directory where resources are exposed).
The script must be run from a dir within the repo of to-dir.
"""

# Typical rsync output:
"""
sending incremental file list
deleting decay7.py
decay_TULL.py

sent 675 bytes  received 34 bytes  1418.00 bytes/sec
total size is 94788  speedup is 133.69
"""

# Example on $HOME/.rsyncexclude file
"""
.#*
#*
*.rsync~
*.a
*.o
*.so
*~
.*~
*.log
*.dvi
*.aux
*.old
tmp_*
.tmp*
*.tar
*.tar.gz
*.tgz
*.pyc
"""

import commands, os, sys

from_ = sys.argv[1]
to_ = sys.argv[2]
```

[3]This functionality should be a part of Git, but no Git expert I have talked to has ever seen use for merging a flexibly defined subset of a repository with another repository. (The current functionality of Git is not capable of working with, e.g., branches that merge with only parts of another branch.)

```
cmd = 'rsync -rtDvz -u -e ssh -b ' + \
      '--exclude-from=$HOME/.rsyncexclude ' + \
      '--suffix=.rsync~ --delete --force %s/ %s' % (from_, to_)
print cmd
failure, output = commands.getstatusoutput(cmd)
print output

delete = []
add = []
for line in output.splitlines():
    relevant_line = True
    for text in 'sending incremental file list', \
        'sent ', 'total size is':
        if line.startswith(text):
            relevant_line = False
    if relevant_line and line != '':
        if line.startswith('deleting'):
            delete.append(line.split()[1])
        else:
            add.append(line.strip())

print delete
print add

for filename in delete:
    option = '-rf' if os.path.isdir('%s/%s' % (to_, filename)) else '-f'
    cmd = 'git rm %s %s/%s' % (option, to_, filename)
    print cmd
    os.system(cmd)
for filename in add:
    cmd = 'git add %s/%s' % (to_, filename)
    print cmd
    os.system(cmd)
```

8 Book versions with and without solutions to exercises

It is easy to turn solutions to exercises on or off by the options `--without_solutions` (for `!bsol` environments) and `--without_answers` (for `!bans` environments). One often wants to publish a book without solutions to exercises, but also make versions with solutions. One possibility is to password protect the versions with solutions.

It can be wise to create a very complicated filename for the files that may be shown in a browser with the URL visible for a class. One possibility is to create a 40 long SHA1 string as filename and start it with a dot to also make it invisible in most operating systems.

```
hash=82dee82e1274a586571086dca04d00308d3a0d86
html=.trash<built-in function hash>
doconce format html book --html_output=$html ...
```

The file following file, called `password.html`, presents an empty page with a button for providing a password, and if approved, the first page of the book opens up.

```
<html>
<body>
<!-- password protected HTML page --->
<script>
function passWord() {
var testV = 1;
var pass1 = prompt('Please enter your password',' ');
while (testV < 3) {
if (!pass1)
history.go(-1);
if (pass1.toLowerCase() == "PASSWORD") {
alert('You Got it Right!');
window.open('DESTINATION.html');
break;
}
testV += 1;
var pass1 =
prompt('Access denied - password incorrect!', 'Password');
}
if (pass1.toLowerCase()!="password" & testV ==3)
history.go(-1);
return " ";
}
</script>
<center>
<form>
<input type="button" value="Enter Protected Area" onClick="passWord()">
</form>
</center>
</body>
</html>
```

Suppose your HTML file with solutions that you want to password protect has the name `book-sol.html`. Then you can simply do

```
cp password.html book-sol.html
doconce replace DESTINATION "$html" book-sol.html
doconce replace PASSWORD "s!m!art|pass@word" book-sol.html
```

A PDF file can easily be password protected with the use of the `pdftk` tool. If `book.pdf` is with solutions, `book-sol.pdf` will also require a password `a!4`:

```
pdftk book.pdf output book-sol.pdf owner_pw foo user_pw "a!4"
```

9 Special features for teaching material

DocOnce offers some special features that can greatly aid development of effective teaching material:

- fancy admonitions
- simple boxes
- interactive code blocks

- hidden code blocks
- exercises
- multiple-choice questions or quizzes
- movies
- quotes

These are briefly exemplified below. If you are mainly interested in how to structure DocOnce-based books, you can safely jump to Section 3.

9.1 Admonitions

Use admonitions!

Need to notify, warn, summarize, ask a question, give a tip, dive into less important details, or really emphasize a result? DocOnce features special *notice*, *warning*, *summary*, and *question* boxes called admonitions. These can be typeset in a variety of versions, depending on the output format (check out `doconce format -help` and the many options with `admon` in the name).

9.2 Simple box

Put a box around something important:

$$1+1=2$$

9.3 Embedded interactive code

The `pyoptpro` code environment. To illustrate program flow, you can step through code (as in a debugger, just more illustrative) using the `pyoptpro` code environment. Here is an example:

```
n = 4
i = 0
while i <= n:
    print i
    i += 1
```

(Visualize execution)

In HTML and Sphinx format this code can be stepped through in the browser, while in LaTeX one gets a link to a web page with this functionality.

The `pyscpro` code environment. The SageMathCell server enables live Python code in web documents. Using the `pyscpro` code environment, Python code can be embedded in a for interactive computing:

```
import random
a = random.randint(1, 7)
print a
```

We can even plot:

```
from numpy import *
from matplotlib.pyplot import *

x = linspace(-2*pi, 2*pi, 801)
y = exp(-0.1*x**2)*sin(4*x)
plot(x, y)
show()
```

HTML and Sphinx output has such interactive Sage Cells, while other output formats can just show the code.

9.4 Exercises

Exercise environments.

DocOnce supports *exercise subsections*, which are subsections with some extra tagging for exercises. Important features are:

- exercises can be divided into subexercises
- one may specify filename(s) for delivering the answer to an exercise
- one may specify filename(s) for the solution of an exercise
- one may specify a remark (fun facts, comments)
- one or more hints can be given to an exercise or subexercise (can be hidden in certain output formats)
- any exercise or subexercise can have a solution field
- any exercise or subexercise can have an answer field (very condensed solution)
- solutions and answers can be removed from the document at compile time
- exercises can have one of four titles: Exercise, Project, Problem, or Example

9.5 Quote

There is also a *quote* environment (invisible box with larger margins).

> We're programmers. Programmers are, in their hearts, architects, and the first thing they want to do when they get to a site is to bulldoze the place flat and build something grand. We're not excited by incremental renovation: tinkering, improving, planting flower beds.
>
> There's a subtle reason that programmers always want to throw away the code and start over. The reason is that they think the old code is a mess. And here is the interesting observation: they are probably wrong. The reason that they think the old code is a mess is because of a cardinal, fundamental law of programming: **It's harder to read code than to write it.** This is why code reuse is so hard. This is why everybody on your team has a different function they like to use for splitting strings into arrays of strings. They write their own function because it's easier and more fun than figuring out how the old function works.
>
> Joel Spolsky, 2000

9.6 Quiz

Question: Does DocOnce feature multiple-choice questions or quizzes?

A. Yes.
B. No.

Answer: A.
Solution:

A: Right. See the quiz manual.
B: Wrong.

Quite often we need mathematics and computer code in questions and answers.

Question: Compute the integral

$$\int_0^{2\pi} e^{-x}\sin^2 x\,dx$$

and report the formula in verbatim LaTeX code,

```
\[ \int_0^{2\pi} e^{-x}\sin^2 x\,dx = ... \]
```

A. There is no closed-form formula for this integral.
B.

```
>>> from sympy import *
>>> x = symbols('x')
>>> F = integrate(exp(-x)*sin(x)**2, (x, 0, 2*pi))
>>> F
-2*exp(-2*pi)/5 + 2/5
>>> latex(F)
'- \\frac{2}{5 e^{2 \\pi}} + \\frac{2}{5}'
```

so the answer is

```
\[ \int_0^{2\pi} e^{-x}\sin^2 x\,dx =
- \frac{2}{5 e^{2 \pi}} + \frac{2}{5} \]
```

C.

```
\[ \int_0^{2\pi} e^{-x}\sin^2 x\,dx =
- \frac{3}{4 e^{2 \pi}} + \frac{3}{4} \]
```

D.

```
\[ \int_0^{2\pi} e^{-x}\sin^2 x\,dx =
- {2\over5}{5 exp^{2 pi}} + \frac{2}{5} \]
```

Answer: B.
Solution:

A: Wrong. Maybe you fiddled around with pen and paper the wrong way... Try `sympy`!
B: Right.
C: Wrong. Something went wrong here with the fractions...
D: Wrong. Well, the math is correct, but the LaTeX code lacks two backslashes, and `\over` is old-fashioned, use `\frac{}{}` in 2015.

The typesetting of a quiz depends on the output format. One can output in plain HTML and Sphinx formats, hover over the choice and get a pop-up tooltip with the right answer and an explanation (if defined). The explanation can only show plain text, so it is empty if it contains math or code blocks, or figures. With HTML Bootstrap styles one can click on a symbol to open up the answer and the explanation. RunestoneInteractive books (Sphinx) offers a button for the same purpose, but the explanation can only be plain text so any math or code block makes the explanation empty. LaTeX output can feature the choices only (no answer, no explanation: `--without_solutions --without_answers`), the short answer only (`--without_solutions`) or both the answer and all explanations (the case in this demo).

9.7 What about a video lecture?

http://youtube.com/PtJrPEIHNJw

Question.

Can you think of applications of the above mentioned features?

References

[1] H. P. Langtangen. Debugging in Python. *http://hplgit.github.io/primer.html/doc/pub/debug*.

[2] H. P. Langtangen. Some document. *http://hplgit.github.io/setup4book-doconce/doc/pub/fake/html*.

[3] H. P. Langtangen. Use of mako to aid book writing. *http://hplgit.github.io/setup4book-doconce/doc/pub/mako/html*.

[4] H. P. Langtangen. *A Primer on Scientific Programming With Python*. Texts in Computational Science and Engineering. Springer, fourth edition, 2014.

Index