Multiple documents from one source

LaTEX for lecturers and teachers

Abstract
In general LaTEX will produce only one output document. This paradigm shifts when harnessing the power of the so-called shell escape. We will show how to produce multiple output documents with differing content from one single source document. The principle is developed step by step illustrating a typical application in academic teaching.
Focusing on mathematical problems we then explore two ways of automating calculations by integrating free software into the LaTEX run.

Keywords
mathematics, problem sheet, shell escape

The problem with problem sheets

Composing appealing problem sheets and preparing instructive solutions is a time consuming task. Further time is lost when problems need to be altered after typesetting, requiring updated plots and recalculated solutions. When LaTEX is used, we should optimize our workflow and exploit its capabilities to handle as many tedious tasks as possible.

Set up for failure: bad practice

A typical road to ruin is typesetting problems and solutions in separate files. That way it is hard to keep changes in order or notation in sync. For better comprehension we would also like to have the problem formulation above its solution, but copying them from the problem sheet source will just increase the sync problem.

The `exam` document class by Philip Hirschhorn [3] eliminates the need for two source documents. It defines a `solution` environment that can be included or hidden via a class option. The solutions are typeset in the source document right after the problem which makes for a very natural workflow and easier debugging. Changes in notation can now be handled globally using the text editors "replace all" function.

On the other hand, having just one source document requires us to pay close attention: without additional adjustments both versions will compile to the same output file name. In an inattentive moment we might accidentally upload the version with solutions to our website too early, rendering the homework assignments pointless. Also `exam.cls` may not provide the customization you need, for example if you have to implement a department's corporate design or would like to define a third version of the problem sheet containing additional notes.

Getting multiple outputs

Multiple outputs would usually be achieved by multiple invocations of LaTEX. Before each run we would need to manipulate the source document in order to have differing content. All of this can be done using shell scripts or MakeFiles, but it

- □ requires an additional (script or make) file
- □ requires non-TEX programs
- □ requires knowledge beyond TEX
- □ may not be platform independent

We will introduce a method that handles these shortcomings using only `pdflatex`.

Requirements for a unified workflow

As we have already seen, it is crucial to have just one source document to avoid incongruities and have a natural way of typesetting.

Typeset problems will most certainly be reused in future courses. Therefore we would like to be able to copy & paste parts of the exam with ease. Our approach should also apply to documents that `\input` problems from another source file.

Finally we want to produce multiple outputs differing in content by a single invocation of `pdflatex`. Our derivation will assume the following use cases:

- □ **student** problems only
- □ **tutor** problems and their solutions
- □ **corrector** problems, solutions and instructions on grading the students work

Designing a unified workflow

Our goal is to produce multiple output documents with differing content in a single run. Though this might seem impossible at first glance, Ulrike Fischer found a way [2] to do this using only `pdflatex`. We will introduce her approach step by step, keeping an eye on platform independence.

Since `pdflatex Sheet.tex` will only produce `Sheet.pdf` our task splits into three parts:

1. Find a way to tell LaTeX which parts of the source file to use and which to ignore
2. Change the file name of the output document in order to be able to produce multiple outputs
3. Produce all versions in just one run of `pdflatex`

Selectively in- & excluding code

We would like to wrap code into a LaTeX environment and have some kind of "switch" in the document preamble to control whether to have LaTeX process it or not. This is exactly what `comment.sty` by Victor Eijkhout [1] does.

Usage. The `comment` package provides us with two simple commands

- ☐ `\includecomment{foobar}` defines the environment `foobar` whose content will be included
- ☐ `\excludecomment{foobar}` defines the environment `foobar` whose content will be ignored

Example. A nice feature of environments defined using `comment.sty` will not break the line, so we can use them in midsentence. The minimal example

```
\includecomment{truth}
\excludecomment{nonsense}

Knuth started
\begin{truth}
developing \TeX{}
\end{truth}
\begin{nonsense}
using WinWord
\end{nonsense}
in 1977.
```

will just output

Knuth started developing TEX in 1977.

Implementing our use cases. Knowing the above, we can easily write a document that matches our use cases, but we would still have to adjust the in- and exclusions before compiling.

To reduce such modifications to a bare minimum we will assign a number to each use case. We can then define a macro `\condition` that expands to the number and use `\ifcase` to make the adjustments for corresponding case.

Listing 1. Use case implementation

```
\RequirePackage{comment}

\includecomment{problem}
\ifcase\condition
  % \condition = 0, student
  \excludecomment{solution}
  \excludecomment{howtograde}
\or % \condition = 1, tutor
  \includecomment{solution}
  \excludecomment{howtograde}
\or % \condition = 2, corrector
  \includecomment{solution}
  \includecomment{howtograde}
\fi
```

Now all it takes is defining the value of `\condition` to control the content of the output.

Coding discipline. One could of course implement the above using `\ifnum` instead. I prefer to use `\ifcase` here because the cases are "automatically numbered" in order of appearance. That way I got less confused about which number represents which use case, saving me time on debugging.

Rethinking command line calls

We want to control the version of the output without editing the source document. We could write a wrapper document that defines our `\condition` macro followed by the actual document code:

```
\gdef\condition{0}
\input Sheet.tex
```

On second thought we can also pass this code on to LaTeX directly on the command line

```
pdflatex "\gdef\condition{0} \input Sheet.tex"
```

avoiding the additional file.

Changing the output file name

By now we can produce any version of the document by altering a single value but they will still be written to the same output file, thereby overwriting the previous output.

A quick look at `pdflatex`'s manpage[1] provides

```
pdflatex --jobname="student" Sheet.tex
```

which will create `student.pdf` from `Sheet.tex`.

Escaping to the Shell

We know that LaTEX can write to auxiliary files using output stream. There is also the special stream 18 which will execute the output on the system shell.

```
\write18{ping tug.org}
```

This is called *escaping* to the shell. Used like this LaTEX will first read to the end of the document before writing to the shell. If we want the command to be executed immediately when LaTEX reaches that point in the document, we use (see [4], p. 226f)

```
\immediate\write18{ping tug.org}
```

In particular this means we can invoke the commands developed in the previous sections from within one `pdflatex` run.

Warning. Giving LaTEX access to the shell is a gateway for exploits. Hence `\write18` is disabled by default. You can temporarily enable it using

```
pdflatex --shell-escape Sheet.tex
```

or permanently by adjusting the configuration[2] of your TEX distribution.

The UniFlow principle

After introducing all the building blocks we are now able to understand Ulrike Fischer's ingenious construction [2] to produce multiple output documents in one single run.

We start off with a document skeleton to demonstrate the recursive nature of the approach.

Listing 2. General UniFlow template

```
% Beginning of Sheet.tex
\ifx\condition\undefined
  % Pseudo shell script (listing 3)
  \expandafter\stop
\fi

% Use case implementation (listing 1)

% Actual document code begins here
```

Processing this code will have `pdflatex` enter the `\ifx` block as the macro `\condition` has not been defined yet. After executing a "pseudo shell script" LaTEX will first expand the token `\fi` and then `\stop` reading. Note that this run will not produce any output.

In the pseudo shell script, we will invoke `pdflatex` again on this very same document. The file's base name is obtained from `\jobname`

```
pdflatex "\string\input\space\jobname"
```

and we told the parser to interpret `\input` as a `\string`, preventing it from expansion ([4], p. 40).

When we add a definition of `\condition`, LaTEX will ignore the `\ifx` block, apply the use case settings and output the desired version.

Considering all use cases and the change of job name we arrive at

Listing 3. Pseudo shell script in LaTEX

```
\immediate\write18{
  pdflatex
  --jobname=\jobname-student
   "\gdef\string\condition{0}
  \string\input\space\jobname"}
\immediate\write18{
  pdflatex
  --jobname=\jobname-tutor
   "\gdef\string\condition{1}
  \string\input\space\jobname"}
\immediate\write18{
  pdflatex
  --jobname=\jobname-corrector
   "\gdef\string\condition{2}
  \string\input\space\jobname"}
```

Combining the UniFlow template (listing 2) with the implementation of the use cases and the corresponding pseudo shell script (listings 1 and 3) we have constructed the single source document `Sheet.tex`. Enabling shell escape and processing it with `pdflatex` will result in the three output documents `Sheet-student.pdf`, `Sheet-tutor.pdf` and `Sheet-corrector.pdf`, each of them with the desired specific content. Therefore all of our initial requirements are met and we have developed a unified workflow.

Pitfall. Neither the wrapping `pdflatex` run nor script 3 will produce an output file named `Sheet.pdf`. This can cause error messages when using text editors with built-in PDF viewers like TEXmaker and its standard "quick build" feature.

Exercise. If you would like to check your understanding of the UniFlow principle, try to write a template for this scenario:

A school teacher always designs two slightly different versions A and B of an exam. She would like to produce the four versions A, B, A with solutions and B with solutions from a single source document.

UniFlow in action

The UniFlow principle can also serve to integrate external programs into the LaTeX run. Due to the author's background the examples are taken from mathematics.

For applications in other subjects see the PythonTeX gallery [5] or Herbert Voß's article on general source code [7].

For the sake of simplicity we now focus on having only one output document. Nevertheless we will still have to define the `\condition` macro (setting it to an arbitrary string value) whenever we want `pdflatex` to ignore the `\ifx` block. The generalization to multiple output versions is left to the reader as an exercise.

Linear Algebra using Sage

Sage is a free and open source computer algebra system. It is best used on Linux since the "Windows version" is actually an Ubuntu virtual machine image containing Sage.

We will give a tiny demonstration of the LaTeX interface called SageTeX and its implementation using the UniFlow principle. Further information on SageTeX can be found in Günter Rau's demonstration [6] or on the Sage homepage[3].

How to compile. SageTeX works similar to BibTeX: First we run LaTeX to extract the Sage commands. These are then processed externally with Sage and the results are included in the second LaTeX run.

```
# Extract Sage commands
pdflatex Example.tex
# Process Sage commands
sage Example.sagetex.sage
# Include Sage outputs
pdflatex Example.tex
```

Implementing UniFlow.

```
\ifx\condition\undefined
  \immediate\write18{
    pdflatex
    "\gdef\string\condition{0}
    \string\input\space\jobname"}
  \immediate\write18{
    sage "\jobname.sagetex.sage"}
  \immediate\write18{
    pdflatex
    "\gdef\string\condition{0}
    \string\input\space\jobname"}
  \expandafter\stop
\fi
```

Exercise. Calculate the eigenvalues of the matrix

$$A = \begin{pmatrix} 19 & 30 & -20 \\ 26 & 39 & -26 \\ 61 & 93 & -62 \end{pmatrix}$$

Solution. The characteristic polynomial of $A$ is

$$\chi\_A(x) = x^3 + 4x^2 + 3x = x \cdot (x+1) \cdot (x+3)$$

hence its eigenvalues are

$$[0, -1, -3]$$

Using `sagetex.sty` we just needed to type

```
% 'sagesilent' returns no output
\begin{sagesilent}
A = matrix(QQ, [[19,30,-20],
    [26,39,-26], [61,93,-62]])
p = A.charpoly()
\end{sagesilent}

\[ A = \sage{A} \]
The characteristic polynomial of
    $A$ is
\[ \chi_A(x) = \sage{p} = \sage{
    factor(p)} \]
hence the eigenvalues of $A$ are
\[ \sage{A.eigenvalues()} \]
```

Note how this assures that the matrix $A$ and the solution will always match in the output document. This is as foolproof as it gets.

Statistics using R and Sweave

Data plotting techniques play an important role in any statistics course: histograms, q-q plots, boxplots etc. are handy tools to analyze measured data.

R is a free statistics software system available for all common operating systems[4]. It comes with the plugin Sweave which "weaves" R (the free successor to S statistics) into LaTeX documents.

We will use an easy example from elementary probability. More advanced examples can be found for example in Uwe Ziegenhagen's demonstration [8] or on Friedrich Leisch's Sweave website[5].

How to compile. As the output of Sweave will be written to `Example.tex` we change the file name of our document to `Example.Rnw` (Rnw = **R** **no**web). Now we can use LaTeX code as usual and insert R code as "chunks" using the noweb syntax. The document is then compiled as follows.

```
# Have R process Example.Rnw and
# create / overwrite Example.tex
R CMD Sweave Example.Rnw
pdflatex Example.tex
```

Implementing UniFlow.

```
% Beginning of Example.Rnw
\ifx\condition\undefined
  \immediate\write18{
    R CMD Sweave \jobname.Rnw}
  \immediate\write18{
    pdflatex
    "\gdef\string\condition{0}
    \string\input\space\jobname"}
  \expandafter\stop
\fi
```

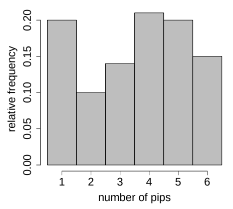
Editing the `Example.Rnw` as source file and using the above code, the correct command line call is

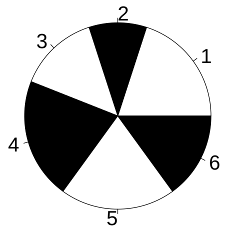
```
pdflatex --shell-escape Example.Rnw
```

If you like to use the tab completion feature of your system shell, it will probably only offer you the `.tex` file. Observe that this will generate the same output because both execute the same pseudo shell script.

Exercise. Roll a dice 100 times in a row recording the number of pips each time. Visualize their relative frequency as a histogram and a pie chart.

Solution. Since this exercise depends on probability, everyone will have a different result. Mine looks like this:





These diagrams where of course generated at compile time from the following code snippet.

```
# Relevant part of Example.Rnw
<< echo=FALSE, fig=TRUE >>=
par(ps=20)
pips <- sample(1:6,100,replace=
    TRUE)
hist(pips, breaks=c(0.5, 1.5,
    2.5, 3.5, 4.5, 5.5, 6.5), col=
    "gray", freq=FALSE, main="",
    xlab="number␣of␣pips", ylab="
    relative␣frequency")
@

<< echo=FALSE, fig=TRUE >>=
pie(table(pips), col=c("white", "
    black"), cex=2)
@
```

Here, due to the use of **sample**(), the output will be different after every compile run.

Aftermath

Of course we could have achieved all of this in a one-call fashion using some kind of shell script, `make`[6] or its LaTeX analogs `latexmk`[7] or `rubber`[8]. On the other hand the UniFlow principle provides a platform independent, script-like alternative without additional (Make)files or non-TeX executables.

The future of UniFlow

To enable anyone to implement the UniFlow principle with ease I will work on developing it into a LaTeX package.

Versatility is UniFlow's biggest asset and every reader will by now have his or her special use case in mind – and most certainly be struggling with the inconvenient syntax of the corresponding `\write18` statement. Hence designing an intuitive command structure will be key and your TeXnical comments and pieces of advice are always welcome.

One step further we could think about a unified interface to integrate virtually any program into the LaTeX run. Herbert Voß [7] already showed how general source code can be extracted from a document and reincluding the output after processing. His approach works with any kind of batching method, allowing for an integration into UniFlow (once developed).

Acknowledgments

Many people have directly or indirectly contributed in the development of the UniFlow principle.

First and foremost I want to thank Ulrike Fischer who provided the core concepts in her short and effective post on StackExchange.

When I was struggling with selectively showing or hiding content, Rolf Niepraschk pointed out the fascinating simplicity of `comment.sty` to me.

Marcus Bitzl is an avid reader of “Die TeXnische Komödie” and drew my attention to the articles on the integration of free math software. He was also the first to encourage the development of a UniFlow package.

The articles by Günter Rau (SageTeX) and Uwe Ziegenhagen (Sweave) were invaluable primers to me and made the vivid demonstrations of UniFlow in action possible.

Finally I would like to express my gratitude to the EuroTeX 2012 organizers for arranging a wonderful and inspiring conference and giving me the opportunity to present the UniFlow principle.

References

[1] V. Eijkhout. comment.sty: Selectively include / excludes portions of text. CTAN:macros/latex/contrib/comment: http://www.ctan.org/tex-archive/macros/latex/contrib/comment.

[2] U. Fischer. Answer to “Can one TeX file output to multiple PDF files?” http://tex.stackexchange.com/a/5265.

[3] P. Hirschhorn. exam.cls: Package for typesetting exam scripts. CTAN:macros/latex/contrib/exam: http://www.ctan.org/tex-archive/macros/latex/contrib/exam.

[4] D. E. Knuth. *The TEXbook*. Addison-Wesley, Eighth printing, August 1986.

[5] G. Poore. PythonTEX: Fast Access to Python from within LaTEX. github:gpoore/pythontex: https://github.com/gpoore/pythontex.

[6] G. Rau. SageTEX. *Die TEXnische Komödie*, 1/2011: http://archiv.dante.de/DTK/PDF/komoedie_2011_1.pdf:17–21.

[7] H. Voß. Einlesen und Ausführen von Quellcode. *Die TEXnische Komödie*, 1/2011: http://archiv.dante.de/DTK/PDF/komoedie_2011_1.pdf:40–54.

[8] U. Ziegenhagen. Datenanalyse mit Sweave, LaTEX und R. *Die TEXnische Komödie*, 4/2010: http://www.dante.de/DTK/Ausgaben/dtk104.pdf:35–45.

Weblinks

1. http://linux.die.net/man/1/pdflatex
2. http://wiki.contextgarden.net/Write18
3. http://www.sagemath.org
4. http://www.r-project.org
5. http://www.statistik.lmu.de/~leisch/Sweave
6. http://www.gnu.org/software/make
7. CTAN:support/latexmk: http://ctan.org/tex-archive/support/latexmk/
8. https://launchpad.net/rubber

Leo Arnold
tex@arney.de