A minimal Org setup to write scientific notebooks

April 2022

A bit of context first. As a matter of fact, I no longer use Emacs. I switched to (Neo)vim a while ago now, since I found myself more comfortable editing text in Vim than I ever was in Emacs. To be honest, I don't really miss any other fancy parts from Emacs operating system, except maybe the ability to run REPL for multiple languages within a few clicks (actually, I used to use `C-c C-c` most of the times), and I really don't miss the package dependencies mess that occurred from time to time when upgrading everything. To tell the truth I don't have great requirements in terms of text editor, but I want a responsive editor, which facilitates text manipulation and fuzzy search within a few clicks.

However, Emacs is already installed on my machine, with the bare essentials in 30 LOC of `init.el`, and Org is readily available from any decent package manager on Linux distros. The following was written on not so recent versions of Emacs (26.3) and Org mode (9.3.1). Also, I will focus on scientific programming languages, namely R, Stata, Python and Mathematica. In the past I used to use Org mode mostly for functional programming languages (Scheme, Common Lisp and Clojure). For an overview of what Org is good for, take a look at Emacs org-mode examples and cookbook.

Why Org

I tend to view Org as a three-fold utility. First, it is a very good markup language, which also happens to be more clean and rich than Markdown. You don't need to worry about spaces for hard breaklines, there's a verse environment, as well as todo and progress state indicators or even macros, and various other things that can be managed under the umbrella of the `#+PROPERTY` element. Second, Org mode in Emacs comes with handy exporting facilities (think of Pandoc, but built in Emacs directly). Third, Org introduced Babel a while ago, which allows to evaluate code directly into an Emacs buffer or when exporting.[1] As such, this provides a way to do literate programming right into your preferred text editor, even if it's (Neo)vim! Of course, if you do not work under Emacs, you lose the ability to evaluate chunks of code right into Emacs, much like an interactive playground. However, you can still evaluate the whole document and export it to HTML or PDF, much like if you were sourcing the whole buffer in Emacs.[2]

The rationale is as follows: We could use general purpose tool like dexy or noweb, or more specialized one (Sweave, knitr, pweave, staweave), use built-in exporters (e.g., from Mathematica markup language), or all-in-one solution in the browser as in Jupyter notebooks. I don't really like working in my web browser, and for what matters I don't need a digital playground, but rather a way to embed snippets of code and their outputs into my document.

[1] There are many other thing built in Org mode, especially for "getting things done", which motivated the original development of Org, but I am not so much interested in these aspects.

[2] Note that the sniprun Neovim plugin allows to run lines/blocs of code from different languages, mimicking the inline evaluation available in Emacs.

This is not a tutorial on Org, Org mode, or even Emacs. You will find plenty of documentation on the Org site itself, as well as on blog posts or GitHub. Also, keep in mind that this is written from the perspective of someone who works exclusively with Neovim. Although there are some plugins that allow to reproduce part of the Emacs way of working with Org,[3] we assume no external plugins at all.[4] At the time of this writing, the Neovim orgmode plugin does not allow to evaluate code block. Its main focus seems to be on the GTD side of Org, and it does it pretty well, in my own view. This short note rather aims at describing what works for me, on my machine, when it comes to writing Org documents as plain text. Beware that you will lose everything you get when working directly with Emacs: inline evaluation of code block, management of references (labels, bibliographic entries, outline, among others), the Org dispatcher which allows to select the exporting backend, and so on. However, you will be able to export your plain text document with the result of your source block pretty printed in your HTML or PDF output files. All that in (Neo)vim. You'll get the best of both worlds!

[3] See this post, Org in Vim, for example.

[4] Besides syntax highlighting which is provided by the venerable vim-orgmode syntax file.

> All editors suck, except Emacs and Vim. – Yours truly, circa 2020

How to write your Org documents

Languages

R and Stata should work right out of the box provided you installed the ESS package. Things may be broken for Julia, though. It should be noted that Stata version < 14 does not allow saving SVG or PNG image, which may limit your ability to export images as easily as with other languages. The only option for those who are on Stata 13 like me is to use imagemagick to post-process images saved in Postscript format. The following oneliner shell script will do the work:

```
for i in *.eps; do convert -density 300 -quality 85 "$i" "${i%%.*}.png"; done
```

Python and Mathematica require additional settings. For Python, you need to point `org-babel-python-command` to the relevant Python you want to use, otherwise it will pick the default `python` program available in your `$PATH`. If you are using a virtual environment, or `python3`, then you likely want to update the default settings. For Mathematica, you will need mash.pl,[5] as described in the following article: Using Mathematica with Orgmode.

[5] There may be better option, but even if this Perl script is rather old, it still works like a charm.

Finally, languages need to be loaded for Org to properly works. This can be done in Emacs config file as follows: (more on this in a later section)

```
(org-babel-do-load-languages
 'org-babel-load-languages
 '((R . t)
   (python . t)
```

```
   (mathematica . t)
   (stata . t)))
```

Basic source blocks

The Org website comes with nice tutorials. Read them, you will learn the basic syntax for highlighting and delineating your text. Next comes the Babel aspect of Org. Each chunk of code will read more or less like the following snippet:[6]

[6] Example taken from Nicolas Rougier's 100 numpy exercises.

```
#+BEGIN_SRC python
import numpy as np
Z = np.zeros((10,10))
print("%d bytes" % (Z.size * Z.itemsize))
#+END_SRC
```

Everything between the `#+BEGIN_SRC` and `#+END_SRC` statements is pure Python code, as indicated in the header, just after `#+BEGIN_SRC`. This is much like Markdown fenced code blocks. Normally, such a code chunk can be evaluated in Emacs by pressing `C-c C-c`, and a `#+RESULTS` block will be displayed right after the source code. The header arguments determine how code should be processed and displayed. It can be global (i.e., valid for all code chunks in the current buffer) or local (i.e., only for the current code chunk). In the latter case, it is specified right after the language (here, `python`). Otherwise, we can put a general statement at the beginning of the document, and update header options on the go. Here is some header stuff that you probably want to put at the top of your Org document:[7]

[7] You can do really crazy stuff with Org source headers. For instance, you can invoke imagemagick to post-process your image files, define custom LaTeX commands that will be inserted conditional on the exporting backend (with or without Org macros). See the Org Babel reference card to learn more.

```
#+PROPERTY: header-args :cache no :exports both :results output :session
```

Source blocks evaluation

Here is a the same example again, but with both input (`SRC`) and output (`RESULTS`) enabled:

```
import numpy as np
Z = np.zeros((10,10))
print("%d bytes" % (Z.size * Z.itemsize))
```

```
800 bytes
```

The results are wrapped up in a verbatim block, which shows up nicely when using LaTeX or HTML backend. The above was produced by passing the following header options: `:exports both :results output`. If we only want the code, and not the results (if any), we simply have to write `:exports code`. In fact, there are five defaults options, for which default values are put in parenthesis: `:session` (`none`), `:results` (`replace`), `:exports` (`code`), `:cache` (`no`), `:noweb` (`no`). Instead of `:exports code`, we could also set `:results silent`. There

Language	Available options
R	`colnames`, `file`, `results`, `session`, `R-dev-args`
Stata	*mostly the same as R*
Python	`results`, `return`, `python`, `session`, `var`, `exports`
Mathematica	*mostly the same as R*
Shell	`results`, `session`, `var`, `noweb`, `tangle`, `stdin`, `cmdline`, `shebang`

Table 1: Language-specific header arguments

are also language-specific arguments, like `:tangle`. Available arguments are detailed in Table 1.

For instance, Mathematica allows to display result as LaTeX expression, using a combination of `TeXForm` and `:results raw`. Another approach is to ask for verbatim output, and post-process the `RESULTS` block. Alterantively, we can simply ask for `:results latex`, as shown below:

```
D[2x^2 Exp[x^2/3], x] // TeXForm
```

$$4\,e^{\frac{x^2}{3}}x+\frac{4}{3}e^{\frac{x^2}{3}}x^3$$

We can also ask Mathematica to compute the n-th order Bose integral

$$I_n=\int_0^\infty \frac{x^n}{e^x-1}dx$$

at $n=1$ (which actually is $\frac{\pi^2}{6}$):

```
Integrate[x/(Exp[x]-1),{x,0,Infinity}]
```

```
Pi^2/6
```

Table 2 summarizes the main options that are generally useful depending on the language at hand.

Language	Available options
R	`output`, `raw`, `table`, `html`, `latex`
Stata	`output`
Python	`output`, `value`, `table`
Mathematica	`output`, `latex`

Table 2: Common `:results` options available for each language

Org does not take care of formatting raw results, though, so care must be taken when you have specific formatting requirements. The following illustrate how we could possibly compute the factorial of 200 using Python. Since the result is quite a big integer, it will overflow our text width, unless we format the result ourselves:

```
import math
from textwrap import wrap
value = math.prod(range(1, 200))
print("\n".join(wrap(str(value))))
```

```
39432893368239525177618160696609253114756798884358663164737126662217 97
```

```
249817016714601521420059923119520886060694598194151288213951213185525309633124764149655567314286353816586186984944719612228107258321201270166459320656137141474266387621212037869516201606287027897843301130159520851620311758504293980894611113948118519486873600000000000000000000000000000000000000000000000000
```

Old good noweb

However, you could simply display the above code without the `print` statement, and add an hidden block for displaying the result. Or you could store the result in a variable and later display it. Or you could use the noweb approach. We will need to provide a name to the previous code block (say, `#+NAME: factorial`), and then we can reuse the same code block later on, using the following syntax:

```
#+BEGIN_SRC python :noweb yes
<<factorial()>>
#+END_SRC
```

Here is what we would get:

```
394328933682395251776181606966092531147567988843586631647371266622179724981701671460152142005992311952088606069459819415128821395121318552530963312476414965556731428635381658618698494471961222810725832120127016645932065613714147426638762121203786951620160628702789784330113015952085162031175850429398089461111394811851948687360000000000000000000000000000000000000000000000000
```

Calling the shell

Sometimes it is easier to use the shell directly. For this to work, we also need to add shell to our list of Babel languages:

```
(org-babel-do-load-languages
 'org-babel-load-languages
 '( ...
   (shell . t)
    ...
   ))
```

Note that we use the keyword `shell`, but in source block this should be replaced by any valid shell available on your system (e.g., bash, sh, zsh). Using a shell allows to call a compiled program directly, to preprocess data using Sed or Awk, and so on. And of course, nothing prevents you from calling your preferred program from a shell directly as illustrated below:[8]

[8] You can do more involved stuff using Dirk Eddelbuettel's little r script.

```
Rscript -e 'summary(rnorm(100))'
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-2.25073 -0.64850 -0.10801 -0.01645  0.57676  2.67609
```

Advanced usage

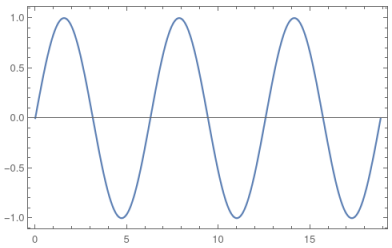
We can do a little better in this case and ask to return a formatted table, using a dedicated package and `:results raw`:

```
library(ascii)
r ← summary(rnorm(100))
print(ascii(r, include.rownames = FALSE), type = "org")
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-2.91	-0.63	-0.07	-0.01	0.68	2.68

The article R Source Code Blocks in Org Mode provides further examples. Other R packages allow to export in HTML (e.g., `xtable`) or LaTeX (e.g., `Hmisc`). Similar options do exist for Python and Pandas, e.g., Get pandas data-frame as a table in org-babel. Here are two approaches to embed graphical output in an Org document. First off, here's some Mathematica where we explicitely save the image before printing it, using `:results file`. Note that we did not export the result for this snippet, but the image is written as `assets/org-setup-sine.png` so that it can be printed anywhere in the document, e.g., in the margin, and customized at will (e.g., by adding a caption).

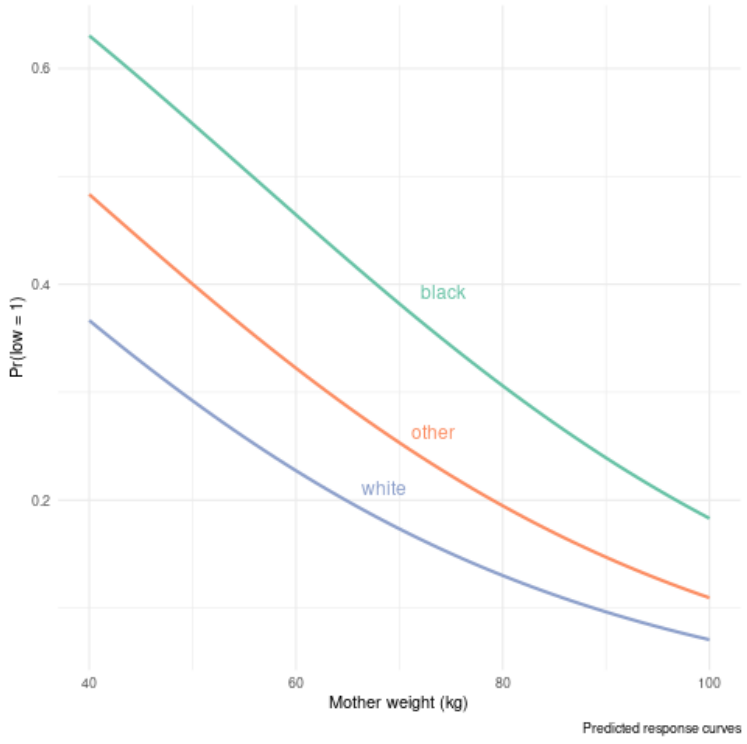
```
p = Plot[Sin[x], {x, 0, 6 Pi}, Frame→True];
Export["assets/org-setup-sine.png", p];
Print["assets/org-setup-sine.png"]
```



And here is some R code, where ask for a direct graphical output using a specific filename (`:file assets/org-setup-bwt.png :results graphics file`). Two things to note: we specify the header option in a separate directive, using `#+HEADER:`, and we wrote the R scipt in a separate file (org-setup-ggplot.r) that is included as is using the `#+INCLUDE:` directive. This directive allows to include any external files into an Org master file, and it is often used for multi-chapter document. However, since we can specify the type of block that will hold the file content, it provides an easy way to embed any kind of code and to keep them seperated without having to tangle them. Together with the shell option discussed in the previous section, it also provides a cheap way to include Scheme or Lisp code and to evaluate code chunk using an interpreter or a compiler.

```
## Little R script that comes along org-setup.org
## http://aliquote.org/articles/notebooks/org-setup.pdf
## Time-stamp: <2022-04-29 21:55:38 chl>
```

```
library(ggplot2)
library(directlabels)

theme_set(theme_minimal())

data(birthwt, package = "MASS")

birthwt$lwt <- birthwt$lwt * 0.45
birthwt$race <- factor(birthwt$race, levels = 1:3, labels = c("white", "black", "other"))

fm <- low ~ lwt + race
m <- glm(fm, data = birthwt, family = binomial)

d <- expand.grid(lwt = seq(40, 100), race = factor(levels(birthwt$race)))
d$yhat <- predict(m, d, type = "response")

p <- ggplot(data = d, aes(x = lwt, y = yhat, color = race)) +
       geom_line(aes(group = race), size = 1) +
       scale_color_brewer(palette = "Set2") +
       guides(color = FALSE) +
       labs(x = "Mother weight (kg)", y = "Pr(low = 1)", caption = "Predicted response curves")
direct.label(p + aes(label = race), method = "smart.grid")
```



Wrapping up

The above sections only showed what's possible with Org using minimal configuration and settings. A lot more can be done. Again, browse the Org website and the internet at large, Google if you really need it. What we learned so far is that the `:results` header argument control many aspects of what is exported and how it may be displayed in the subsequent output. Your options here will depend on the exporting backend itself. If you use PDF via LaTeX, almost everything is possible and you'll get nice tables. Relying on the shell may be interesting in many occasions. Although we only dealed with mainstream scientific programming languages, other data-oriented languages are available, like SQLite.

How to process your Org documents

Local and global setup

There are two options to export your Org document. Either you reuse your own Emacs configuration, or you write one from scratch. The latter is useful in case you want to maintain separate configuration for each project, while the former is the easy way to go. Here is what you could put in a file named `setup.el`:

```
(load (expand-file-name "init.el" user-emacs-directory))
(require 'org)
(load "ox-bibtex.el")
```

The above instructions load your whole Emacs config, via `init.el` in the user Emacs directory. In your Makefile,[9] you then invoke Emacs like this:

[9] Everyone's using a Makefile, right?

```
%.html: %.org
    emacs --batch -l setup.el $< -f org-html-export-to-html --kill
```

If, on the other hand, you prefer to write custom settings for each project directory, then there's a little more work involved. First, you will need to import the relevant Emacs package and load the appropriate languages. This can be done as follows (again we assume everything is stored in a file named `setup.org`):

```
(require 'org)
(require 'ess-site)
(require 'ess-stata-mode)
(require 'ox-bibtex)

(org-babel-do-load-languages
 'org-babel-load-languages
 '((R . t)
```

```
   (python . t)
   (mathematica . t)
   (stata . t)))

(setq ess-ask-for-ess-directory nil)
(setq inferior-R-program-name "/usr/bin/R"
      org-babel-python-command "/usr/bin/python3"
      org-babel-mathematica-command "~/bin/mash"
      mathematica-command-line "~//bin/mash"
      inferior-R-args "-q --no-save --no-restore")
```

There's a bunch of default options that you can specify into your config file. For instance, you can ask to add the `:session` flag automatically for all R source blocks (apart form the global header option, we can add language-specific buffer settings).[10] The `ox-bibtex` package is only required if you need to manage your bibliographic entries (this requires bibtex2html for the HTML exporting backend).

[10] In this configuration, I assume that the program `stata` is available in your `$PATH`. Usually, one of the many versions of Stata (SE, MP, IC) is symlinked into `/usr/local/bin` or `/usr/local/stataXX/`, where `XX` stands for Stata version number, as `stata-se` or `stata-mp`. Sometimes, an additional symlink, `stata → stata-[(mp|se)]`, is automatically created (by the install script or via Stata GUI). This PR assumes that such a link does exist, otherwise it has to be created by the user under his `$HOME` directory or in system-wide `$PATH`.

Wrapping up everything in a shell script

If you are going to use this approach everyday, you are better off writing a little shell script to perform all the work. Here is a simplified illustration:

```
#!/usr/bin/env bash

OPT=$1
FILE=$2

ELISP="/home/chl/Documents/notes/assets/org-babel.el"

case $OPT in
-pdf)
emacs --batch -l "$ELISP" --eval "(progn (find-file \"$FILE\") (org-latex-export-
to-pdf))"
;;
-html)
emacs --batch -l "$ELISP" --eval "(progn (find-file \"$FILE\") (org-html-export-
to-html))"
;;
*)
echo "Unknown export format."
;;
esac
```