A minimal Org setup to write scientific notebooks

April 2022

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 there 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 wa 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.

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]

[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.

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.

This is not a tutorial on Org, Org mode, or even Emacs. 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. 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 aims at describing what works for me, when it comes to writing Org documents as plain text (syntax highlighting is provided by the venerable vim-orgmode syntax file). You will lose verything you get when working directly from 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.

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

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,[4] as described in the following article: Using Mathematica with Orgmode.

[4] 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:[5]

[5] 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:[6]

[6] You can do really crazy stuff with Org source headers. For instance, you can invoke imagemagick to post-process your image files, define custom 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.

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

Table 1: Results options available for each language

Advanced usage

How to proceed your Org documents

Local and global setup

Again, there are two options to export your Org documents. 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, you then invoke Emacs like this:

```
%.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")
```

Wrapping up everything in a shell script

If you are going to use this 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
```