# Save time and `$make` reproducible research with GNU Make

Leo Siqueira
lsiqueira@rsmas.miami.edu

## Reproducible Research

- A lot of the research described in recent papers is not reproducible.
- Only about half of projects can actually be built to begin with.
- Not talking about reproducing the same results as yet.
- Why is it so hard for people who are (and will be) the leaders in the field to publish code that can be easily compiled and reproduced?

# Why Use Make

- Merely a tool for building large binaries (c or fortran) or libraries (e.g., netcdf) and it is, almost to a fault (portability issues).

- Make is much more than that!

**Makefiles are machine-readable documentation that can save you time and make your workflow reproducible.**

# Why Use Make

- It doesn't fundamentally change how you do something, but it encourages to you record each step in the process, enabling you (and your coworkers or reviewers) to reproduce the entire process later.

- The core concept is that generated files depend on other files. When generated files are missing, or when files they depend on have changed, needed files are re-made using a sequence of commands you specify.

- Make is language agnostic. There is not usually one best tool for an entire project. Data analysis often involves shell scripting, a bit of Python (Matlab or R), and who knows what else.

# Why Use Make

- You changed one file in a big project, and you aren't sure which figures and tables need to be remade.

- Makefile will only process the files that have been updated. It is smart enough not to re-run code if it has already been run.

- If nothing has changed, running make does nothing.

- If only say the tex file changes, running make will re-compile the tex document and produce the .pdf

- If the Matlab code has changed, running make will re-run the code to generate the new analysis and graphs, and then re-compile the tex document.

- All I do is type make and it figures out what is required.

# How to use Make

- If you are using a Mac or Linux, you will already have make installed.

- If you are using Windows and have Rtools installed, then you will also have make.

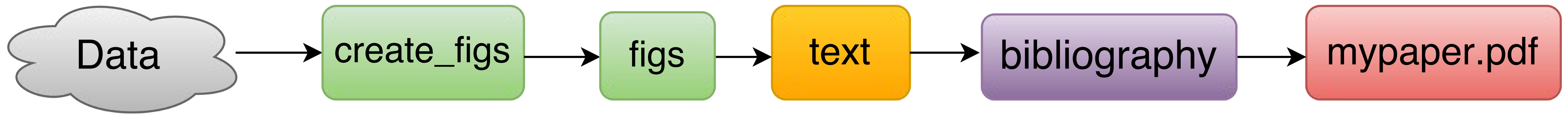
- Otherwise, Windows users will need to install it. One implementation is in GnuWin.

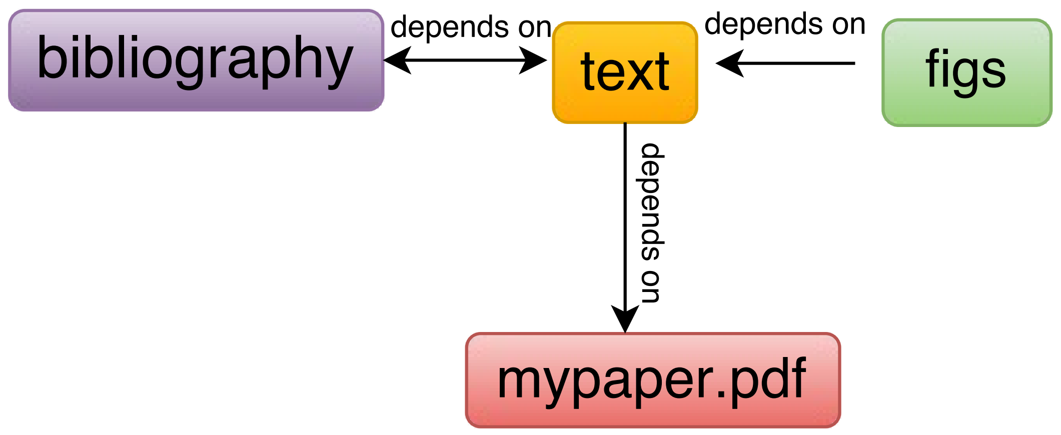
## Managing Complex Research Workflows with Make

Compiling the final PDF of a paper is a bit of work:

- Get the data.
- Run each analysis script in batch mode to produce the relevant figure.
- Run latex and then bibtex and then latex a couple of more times.
- The analysis scripts need to be run before latex if, and only if they've changed.

# Managing Complex Research Workflows with Make





# Using Make

- Go into the the directory for your project.
- Create the Makefile file.
- Every time you want to build the project, type make.

# Basic is enough

- The Syntax Isn't Pretty

- The ugly side of Make is its syntax and complexity; the full manual is a whopping 183 pages. Fortunately, you can ignore most of this, and start with explicit rules of the following form:

```
[targetfile]: [source1 source2 …]

    [command to run] $(VARIABLE) [etc]
```

- Here targetfile is the file you want to generate, sourcefile is the file it depends on (is derived from), and command is something you run on the terminal to generate the target file.

- These terms generalize: a source file can itself be a generated file, in turn dependent on other source files; there can be multiple source files, or zero source files; and a command can be a sequence of commands or a complex script that you invoke.

## [command to run]

- `wget http://www.somedatarepos.gov`
- `python myscript.py`
- `matlab -nojvm -nodisplay -nosplash -r "try, run(‘.scripts/myscript.m’), catch, exit, end, exit”`
- `R CMD BATCH myscript.R`
- `pdflatex -interaction=batchmode`

# Automatic variables

- There are a bunch of automatic variables that you can use to save yourself a lot of typing. Here are the ones that I use most:
- $@    the file name of the target
- $<    the name of the first prerequisite (i.e., dependency)
- $^    the names of all prerequisites (i.e., dependencies)
- $(@D)    the directory part of the target
- $(@F)    the file part of the target
- $(<D)    the directory part of the first prerequisite (i.e., dependency)
- $(<F)    the file part of the first prerequisite (i.e., dependency)

# Tech Note List Targets

Add the following line to the end of your .bash_profile or .bashrc

```
#GNU make list targets
alias mtargets='make -qp | awk -F":" "/^[a-zA-Z0-9][^$#\/
\t=]*:([^=]|$)/ {split(\$1,A,/ /);for(i in A)print
A[i]}"'
```

## Take `$HOME` message

- Update any source file, and any dependent files are regenerated with minimal effort. Keep your generated files consistent and up-to-date without memorizing and running your entire workflow by hand. Let the computer work for you!

- Modify any step in the workflow by editing the makefile, and regenerate files with minimal effort. The modular nature of makefiles means that each rule is (typically) self-contained. When starting new projects, recycle rules from earlier projects with a similar workflow.

- Makefiles are testable. Even if you're taking rigorous notes on how you built something, chances are a makefile is more reliable. A makefile won't run if it's missing a step; delete your generated files and rebuild from scratch to test. You can then be confident that you've fully captured your workflow.

**Do your future self and coworkers a favor, and use Make!**

“Think twice, code once.”

- ANONYMOUS