How to use Make to automatically update projects dependent upon R and LaTeX

Joshua G. Harrison
1000 E. University Ave.
Department of Botany, 3165
University of Wyoming
Laramie, WY 82071, USA

March 1, 2019

1 Why use Make?

Inevitably, over the course of a project one finds themselves updating data and scripts many times. Some scripts depend upon other scripts, and updating any of them, or the underlying data, requires re-executing all scripts in order. One could make a bash script to do this, but Make offers several advantages over a home-cooked bash script. First, Make allows you to specify dependencies among your files which can expedite re-execution. An example of a dependency structure is: script x depends upon data y. When one runs Make it will search specified files, find any that updated, and execute everything that depends on those files. When you have a large project, with a complex dependency structure, this could provide large speed gains over bash. In this document, I will provide an example of how to use Make to sync updates in ones R scripts or data to a LaTeX manuscript.

Setting up a Makefile for your project will take a few minutes, but this time investment will be repaid on any but the simplest of projects.

1.1 A brief history of Make

According to the all-knowing Wikipedia, Make was invented by Stuart Feldman at Bell labs way back in 1976. Here is a quote from Feldman regarding the genesis of the program (swiped from Wikipedia; give them $5 this year):

> Make originated with a visit from Steve Johnson (author of yacc, etc.), storming into my office, cursing the Fates that had caused him to waste a morning debugging a correct program (bug had been fixed, file hadn't been compiled, cc *.o was therefore unaffected). As I had spent a part of the previous evening coping with the same disaster on a project I was working on, the idea of a tool to solve it came up. It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make that weekend. Use of tools that were still wet was part of the culture. Makefiles were text files, not magically encoded binaries, because that was the Unix ethos: printable, debuggable, understandable stuff.
>
> –Stuart Feldman, The Art of Unix Programming, Eric S. Raymond 2003

Since the 70s Make has come standard with Unix distributions. It is often used to compile software and update parts of software that rely on things that have changed. Thus, you can just compile the necessary parts of complicated software (like an operating system) without recompiling the whole thing and wasting time.

2 The tutorial

2.0.1 Prerequisites

You will need to have git, R, a TeX distribution with LaTeX, and Make installed. On an Apple computer you can install Make with the developer tools (to figure out how to do this Google "installing Xcode tools mac"). To install Make on Windows see this page for a place to start, `http://stat545.com/automation02_windows.html`.

If you wish to render pdfs from LaTeX files outside of Overleaf then you will want to also install pdfTeX (or something similar). If you are on an Apple, then you can install mactex, which will come with pdflatex. Note, that you may need to add the TeX library to your path to make pdfTeX easily executable. You can add this line to your path: "/Library/TeX/texbin". For Windows, my understanding is that a pdf conversion tool should come with most TeX distributions. Alternatively, you can do all rendering in Overleaf.

Second, you will need to link your local project folder (such as an R project) to a git repository. It makes sense to use a remote repository, so your work will be backed up and you can easily access it from other computers or share the repository with collaborators. At the time of writing, both Github and Bitbucket offered free private repositories. Third, if you use Overleaf, then you will need to link your Overleaf project to the aforementioned repository.

For information on how to do all this, see the excellent tutorial created by Jessica Rick (`https://github.com/jessicarick/resources`)

2.0.2 How Make works

When one runs Make it looks for a "Makefile" that includes instructions for how Make should run. Specifically, a Makefile will include dependency structures and commands that define the build order for a project. For example, a Makefile could say script X depends on script Y and data Z, if Z changes then re-execute Y and then re-execute X.

Inside a Makefile are a series of commands, called "rules", following this structure:

target: dependencies
instructions

where "target" is a file in your project, such as a manuscript; "dependencies" are the scripts, data, etc. that the target depends upon (e.g. the figures and results that go into your manuscript); and, "instructions" are things that must be done to the dependencies in order to successfully get the output that the target uses.

Importantly, the instructions line must be indented with a tab, if you use spaces your Makefile will fail with an error.

For this tutorial, we will use a project with the following dependency structure: a LaTeX manuscript that relies on a figure and some results from two R scripts, which, in turn, rely on a single csv data file. We also want to use git as a version control system for all the files in our project, and sync any local changes with a remote repository. In particular, we want to send our LaTeX manuscript to a remote repo so that we can pull it into Overleaf and use all its handy features. *Note, if changes are made in Overleaf then those changes must be "pulled" to the local repository to see them. It is simplest to start every work session by pulling changes from the remote repository, so you stay synced up.*

If you want to take your work in a new direction, then learn about making "branches" in git*. If you don't understand what I am talking about read Jessi's tutorial, linked above, and read a bit aboug* git*.*

The repository at `https://github.com/JHarrisonEcoEvo/Reproducible_workflow_tutorial` emulates the project structure verbally defined above. Clone this repository and use it to practice! The file "main.tex" will be our example manuscript (with figures, tables, and code). This also functions as a simple LaTeX template that one could use for scientific manuscripts. There are some tips and tricks in this file, and a justification for using LaTeX, so it is worth a scan. There are also two Ṙ scripts in the example folder, a directory holding the data, a directory holding the results, this pdf, and of course a Makefile.

2.0.3 Example code

Navigate to the cloned example directory and open the Makefile (use less or a text editor). It should look like this (with a few additions not shown here):

```
manuscript.pdf: main.tex linearModel.R scatterplot.R ./data/testdata.csv
	pdflatex -jobname=manuscript main.tex

main.tex: linearModel.R scatterplot.R
	Rscript linearModel.R
	Rscript scatterplot.R
	latex main.tex

linearModel.R: data/testdata.csv
	Rscript linearModel.R

scatterplot.R: data/testdata.csv
	Rscript scatterplot.R
```

To reiterate, the use of a tab to indent the instruction line following each dependency chain is crucial. Also, the ":" after the target file is critical as well.

To break this code down, we are saying:

1. Our manuscript.pdf depends upon main.tex, which is our LaTeX file, two R scripts and some data. If any of these change, then we will need to rebuild main.tex. If Make sees that the dependencies have been updated, it will next look for how to build these dependencies. This is why the order of rules matters.

2. main.tex depends upon the two R scripts and some data. When any of these change we should run them and rebuild main.tex

3. scatterplot.R, which is a script that makes a scatterplot, depends upon some data. We should rerun it if the data are updated.

4. linearModel.R, which is a script that runs a linear model and outputs the results as a table. As before, if the data change then the script should be rerun.

Underneath each dependency structure is an instruction line for what to do if any of the dependencies have changed. Make sense?

As mentioned, the order of rules matter. One should place the most important target, with the most dependencies, first. For more, see `https://www.gnu.org/software/make/manual/html_node/Rules.html`

Importantly, linearModel.R outputs the results of a linear model into LaTeX format using xtable*, which is a super handy R package that translates matrix-like objects into table format. Read the script for an example of how to do this. It is very useful! Never input data into a table by hand again, saves on errors and time.*

Note, you can put all sorts of bash commands into the instruction lines. If you do need to change directories, then you must put the call to cd on the same line, using the line extension command, the backslash (\). For instance, one could do this to change into the directory R before running a script,

```
target: dependency
    cd R;\
    Rscript myprogram.R
```

Note, this is just an example, in this case it would make more sense to just put the whole relative path to the script in the call to Rscript (i.e. Rscript ./R/myprogram.R"

2.0.4 Adding git to our Makefile

Lets say we want to commit and push any changes to any file whenever we build. NOTE: this may not be ideal for many reasons, including 1.) one normally does not want to commit every single change, however small; 2.) the easiest way to auto commit and push is to use a standard commit message, which is not helpful at all when looking through version history. That said, I often edit work locally, then need to push it to Overleaf and the convenience of doing this automatically in the Makefile trumps the very real concerns I note above. For important script commits, or data commits, it is wise to provide a more detailed commit message!

Caveats aside, one way to call git as needed is to make a variable in the Makefile. A variable can save quite a bit of typing (and of course you could get quite creative with variables, if desired).

We can augment our original Makefile, thus:

```
send_overleaf=git commit -m "auto commit from make" $^ $@;git push

manuscript.pdf: main.tex linearModel.R scatterplot.R ./data/testdata.csv
    pdflatex -jobname=manuscript main.tex
    ${send_overleaf}

main.tex: linearModel.R scatterplot.R
    Rscript linearModel.R
    Rscript scatterplot.R
    latex main.tex
    ${send_overleaf}

linearModel.R: ./data/testdata.csv
    Rscript linearModel.R
    ${send_overleaf}

scatterplot.R: ./data/testdata.csv
    Rscript scatterplot.R
    ${send_overleaf}
```

Note that we assign a variable, called “send_overleaf”, to perform git commit and git push for us. To call the variable we prefix it with $ and wrap it in either brackets or parantheses. The part of the code that says $^ is an automatic

variable that stands for all the dependent files. The $@ is an automatic variable that stands for the target. So we are saying we want to commit and push all the dependent and target files. Lots of other automatic variables exist. See Karl Broman's example (`https://kbroman.org/minimal_make/` for more.

Tip: patterns can be used in Make *as well. For instance, you could use wildcards to select all files ending in csv or fastq. See Karl Broman's example for more.*

2.0.5 How to run Make

Once you have a Makefile in place, that works. All one has to do to run it, is navigate to the directory with the file and type "make" on the command line. That is it! The Makefile will execute and any targets with dependencies that have changed will rebuild.

2.0.6 More than you will ever want to know

For bonus points check out the behemoth Make documentation (`https://www.gnu.org/software/make/manual/make.html`). One can probably find all sorts of time saving tricks within, however keep in mind that simple Makefiles are easy to glance at and understand. Even for collaborators that don't use Make, they can probably figure out what is going on. If one adds too much to a Makefile, then it could become a bit onerous to interpret. Interpretability is better than code compactness.