Journal of Statistical Software

June 2020, Volume 94, Code Snippet 1. *doi: 10.18637/jss.v094.c01*

Using GNU Make to Manage the Workflow of Data Analysis Projects

Peter Baker
University of Queensland

Abstract

Data analysis projects invariably involve a series of steps such as reading, cleaning, summarizing and plotting data, statistical analysis and reporting. To facilitate reproducible research, rather than employing a relatively *ad-hoc* point-and-click cut-and-paste approach, we typically break down these tasks into manageable chunks by employing separate files of statistical, programming or text processing syntax for each step including the final report. Real world data analysis often requires an iterative process because many of these steps may need to be repeated any number of times. Manually repeating these steps is problematic in that some necessary steps may be left out or some reported results may not be for the most recent data set or syntax.

GNU Make may be used to automate the mundane task of regenerating output given dependencies between syntax and data files. In addition to facilitating the management of and documenting the workflow of a complex data analysis project, such automation can help minimize errors and make the project more reproducible. It is relatively simple to construct `Makefile`s for small data analysis projects. As projects increase in size, difficulties arise because GNU Make does not have inbuilt rules for statistical and related software. Without such rules, `Makefile`s can become unwieldy and error-prone.

This article addresses these issues by providing GNU Make pattern rules for R, `Sweave`, **rmarkdown**, SAS, Stata, Perl and Python to streamline management of data analysis and reporting projects. Rules are used by adding a single line to project `Makefile`s. Additional flexibility is incorporated for modifying standard program options. An overall strategy is outlined for `Makefile` construction and illustrated via simple and complex examples.

Keywords: GNU Make, Make, reproducible research, R, **rmarkdown**, `Sweave`, Stata, SAS.

1. Introduction

Scientists, statisticians and data scientists often work on several data analysis projects simultaneously. Each project will involve a number of steps such as a planning phase, reading and cleaning data, preliminary summaries and plots, statistical analyses and reporting find-

ings. Throughout this process, the data analyst will also need to regularly document their work and back up data files, program files, reports and correspondence. These steps may be repeated any number of times before the project is completed. Sometimes these steps are carried out in a relatively *ad-hoc* manner and may even include a point-and-click cut-and-paste approach to programming and reporting. Instead, to facilitate reproducible research, we advocate breaking down these tasks into manageable chunks by employing separate files of statistical, programming or text processing syntax for each step including the final report.

While efficiently managing these steps is very important for statistical consultants and indeed anyone involved in quantitative research, very few formal resources are available and so most of us learn by trial and error. Exceptions are Long (2009) who outlines the workflow of data analysis using Stata (StataCorp 2019) and Dasu and Johnson (2003) who provide extensive advice on exploratory data mining and improving data quality. Dasu and Johnson also quote the figure well known to statisticians that about 80% of effort in data analysis projects is often spent getting the data cleaned and in suitable shape for analysis. Indeed, there is a growing interest in more efficient methods for so called data wrangling and tidying steps in the data analysis process (Wickham 2014; Boehmke 2016).

Many of the steps in a data analysis project are sequential. For instance, data cleaning may highlight data that need to be revised and then all other downstream steps like data tidying, summarizing, statistical analysis and reporting will need to be repeated. Changes in statistical modeling may update the reporting of the study but will not require rereading, cleaning and summarizing data. For a simple project, it might be quite straightforward to redo the appropriate steps. However, even with good documentation, for larger or more complicated projects it becomes easier to make mistakes and harder to reproduce results.

GNU Make is a *build automation utility* widely used for large programming projects. It helps keep track of interdependencies and aids efficiency in only recompiling altered modules. Here we employ it for rerunning the appropriate steps to regenerate output for data analysis projects. In particular, it is well suited to R (R Core Team 2020) since all steps and reports in a project can be specified in a mixture of R syntax and either `Sweave` (Leisch 2002) or **rmarkdown** (Allaire *et al.* 2020) files for reports or presentations. Indeed, GNU Make rules can even be employed to carry out post analysis tasks like backing up data to a server or sending reports to collaborators although it could be argued that a version control system like **git** may be better suited to such tasks.

With an elementary knowledge of Make, it is straightforward to specify dependencies between files and hence rerun only a minimal number of steps in the process. While, every step can be specified explicitly, it is much more efficient to use (and reuse) pattern rules as outlined in Section 4. The pattern rules developed here, which are available at `https://github.com/petebaker/r-makefile-definitions`, simplify `Makefile`s by providing general rules to generate output given an input file.

While other mechanisms and systems are available for automating builds, GNU Make is widely used and can be considered to be the de-facto standard despite possessing some apparent limitations and shortcomings. Like other practitioners (see Graham-Cumming 2015), having tried alternative Make systems, we have always found ourselves returning to GNU Make because it works for our data analysis, consulting and research projects. It is relatively simple to use and has enough functionality for the majority of such data analysis projects. While various Make tools adopt or borrow features from other versions of Make we employ specific

GNU Make features not generally available in other versions of Make like the `VPATH` variable, reading variables from and communicated through the environment, modified automatic variables including the `$^` construct containing a list of all prerequisites of the current target and phony targets with `.PHONY` among others. Indeed, while there a range of viewpoints, many argue that in their experience, employing alternative more complicated build platforms may over-complicate the process and induce errors and so prefer standard GNU Make (Hacker News 2013; Bostock 2013).

It is straightforward to learn how to write and use `Makefile`s simply by modifying existing examples from the web. There are a number of good blogs or online tutorials providing a nice overview of `Makefile` construction and use including Gillespie (2011), Olson (2017), Jackman and Bryan (2014), Broman (2018), Jones (2018) and Hyndman (2018). Indeed, recent R-centric build tools like **drake**, **maker**, and **MakefileR** look promising but a good working knowledge of GNU Make will undoubtedly prove useful if employing these approaches; see Landau (2018), Gatto, Breckels, Gibb, and Smith (2014) and Müller (2016), respectively. We demonstrate that it is straightforward to employ GNU Make via simple and more complex examples which are also available on GitHub at `https://github.com/petebaker/r-makefile-definitions`. Alternatively, the R package **gnumaker** (Baker 2020) available at `https://github.com/petebaker/gnumaker` to create and graph simple `Makefile`s using the rules outlined here is under development.

This article is organized as follows. Section 2 provides a quick start to using the GNU Make rules provided here. Section 3 outlines basic Make usage and a strategy for writing `Makefile`s. An overview of pattern rules specifically defined for R, **rmarkdown** and other statistical software is given in Section 4. Section 5 provides several extensions and tips. Recursive and non-recursive Make are outlined for more complex projects containing several subdirectories in Section 6 while Sections 7 and 8 discuss this approach. Finally, Appendices A, B, C and D outline how to set up **RStudio** (**RStudio** Team 2015) and **ESS** (**Emacs** speaks statistics; Rossini, Heiberger, Sparapani, Mächler, and Hornik 2004) to use Make easily; a complete `Makefile` for Section 3.2; details of the Make rules provided and Make installation, respectively.

2. Getting started with GNU Make

In general, GNU Make will be installed on Unix/Linux systems but Windows and macOS users may need to install it. Depending on their setup, some Windows 10 users may find they can use GNU Make natively from a Bash shell. Otherwise, to obtain GNU Make, Windows users can install the **Rtools** from `https://CRAN.R-project.org` while macOS users can install XCode available at `https://developer.apple.com/support/xcode/` from the Apple App Store. However, since the versions may be ancient in computing terms, newer versions may be preferred. See Appendix D for details.

2.1. A simple example

My advice is to start simply. For example, create or copy a statistical syntax file for R, Stata, SAS, Perl, Python or PSPP and download the file `r-rules.mk` from `https://github.com/petebaker/r-makefile-definitions`. Next, in the same directory or folder, create a text file called `Makefile` with the single line, namely an `include` statement as follows:

```
include r-rules.mk
```

Including a file simply means that the contents are processed as if they were inserted in the `Makefile` at that point. The file `r-rules.mk` contains numerous rules for producing output files including from R, Stata, SAS, **rmarkdown** and `Sweave` syntax files. Here we mainly describe R and related software but the principle is the same.

For instance, if you are reading in some data using R with a syntax file named `read.R` then typing

```
$ make read.Rout
```

(without the command prompt `$`) in a terminal window would run R in batch mode to process `read.R` to read the data and produce a text output file `read.Rout`. If `read.R` is newer than `read.Rout` or `read.Rout` is not present, then `read.R` is run to produce `read.Rout` and the following would be output to the terminal or integrated development environment (IDE) window to indicate that R is being run.

```
R CMD BATCH --vanilla read.R
```

However, if `read.Rout` has already been produced and is newer than `read.R`, R would not be run and instead we would see the message

```
make: 'read.Rout' is up to date.
```

Typing `make read.docx` would instead run the **rmarkdown** function `render` to produce a word document. These output files are known as *targets*, and are only updated if the *dependency* files have changed or the target file is not present. The following output will be seen.

```
Rscript --vanilla -e
               "library(\"rmarkdown\");render(\"read.R\", \"word_document\")"

processing file: read.spin.Rmd
  |.....................                                          |  33%
  ordinary text without R code
  |.......................................                        |  67%
label: unnamed-chunk-1
  |.............................................................| 100%
  ordinary text without R code
output file: read.knit.md

/usr/bin/pandoc +RTS -K512m -RTS read.utf8.md --to docx --from
 markdown+autolink_bare_uris+ascii_identifiers+tex_math_single_backslash
 --output read.docx --highlight-style tango
Output created: read.docx
```

Here the *target* file `read.Rout` implicitly has the *dependency* file `read.R`, as does the *target* `read.docx`. However, if we were using `read.R` to read a text file containing data for an

analysis then the `Makefile` should also be set up to describe relationships between the targets (outputs) and all dependencies (inputs) in the data analysis process. For instance, if `read.R` is employed to read in data from `simple.csv` then the `.csv` file is also an input and the relationships should be included in the `Makefile`.

Help is included in the file `r-rules.mk`. Type `make help` for further details including rules for non-R software once this file is included in the `Makefile`.

Make can also be run with a number of options to either check what will happen or debug the process when things do not work as expected. Details may be found in Section 4.4.

3. `Makefile` basics

Typically, `Makefile`s are set up using a programmer's editor or IDE. While any text editor will suffice, many R users would use either **RStudio** (**RStudio** Team 2015), **ESS** (**Emacs** speaks statistics; Rossini *et al.* 2004) or **Nvim-R** (de Aquino 2016), all of which provide syntax highlighting for GNU Make files and the ability to assign running `make` to a keyboard short cut or menu. Users of other statistical software may typically use the editor provided or **ESS**. Details of setting up Make in **RStudio** and **ESS** are provided in Appendix A. Tips for streamlining `Makefile`s are provided in Section 4.4.

3.1. Specifying targets and dependencies

Typically, data analysis projects can involve multiple data sets, data cleaning, several analyses and reports. `Makefile`s can easily be constructed to take into account more complicated dependencies between targets and dependency files. In general, a *rule* to produce a `target` file given a number of prerequisites `prereq_1, prereq_2` and `prereq_3` is specified as follows.

```
target: prereq_1 prereq_2 prereq_3 ...
	first command to make target
	second command and so on
```

Firstly, note that the ↦ symbol represents a single TAB character at the start of the line. Commands that we could normally type at the prompt such as running R in batch mode are specified after the TAB character. If we wish to do so then we can include several lines of commands.

Secondly, if there is already a predefined rule to make the target given the first dependency file then we only need to specify the first line since the commands are defined elsewhere. See Section 4.3 for details of rules defined in `r-rules.mk` for languages like R and **rmarkdown** and how to set program options.

Important note. *As noted above, the lines containing commands must start with a* `<tab>` *character shown here as ↦. If you copy and paste a* `Makefile` *from a web site then usually tabs are converted to spaces and give an error message!*

3.2. Simple example (continued)

Consider the single line `Makefile` containing `include r-rules.mk` for the simple example outlined in Section 2.1. We may have several syntax and **rmarkdown** files in the current

directory including `read.R` and `report.Rmd`. The rules contained in `r-rules.mk` allow us to automatically create target files such as `read.pdf` and `report.docx` using `make target` at the terminal command prompt. However, this does not take into account prior steps in the process nor any additional dependency files. Updating data files may also facilitate rerunning various steps in the process. Such dependencies need to be declared explicitly.

The syntax in file `read.R` may read and tidy data from a text file `simple.csv` and subsequently produce `report.pdf` from `report.Rmd`. If either `simple.csv` or `read.R` changed then we would like all output to be regenerated and only the report to be remade if `report.Rmd` is updated. To achieve this, we can construct the minimal `Makefile` to read `simple.csv` and produce `report.pdf` from an R syntax and **rmarkdown** file as follows:

```
report.pdf: report.Rmd read.Rout
read.Rout: read.R simple.csv
include r-rules.mk
```

Note that dependencies are specified on to the right of the colon (`:`). In general, we read the `Makefile` from the bottom up since the last definition is the one employed. Hence, we can regard the rules as being included first although, strictly speaking, they are included in sequence. Next the file `read.R` is run in batch mode to output `read.Rout` and we could also store the data and any plot or analysis objects in the R binary data file `simple.RData`. Finally, the report `report.pdf` is produced by loading `simple.RData` and printing these objects appropriately along with the text of our report.

To produce the relevant target files, we simply type `make` at the terminal or press the appropriate button in our IDE. By default, Make builds the first target which is `report.pdf`. If none of the intermediate target files are present then Make produces them by running R or **rmarkdown** as necessary. Otherwise, any target file that is not 'up to date' is updated.

While the `Makefile` above is perfectly adequate, it is poorly documented and would normally be extended to include comments and extra functionality. As with any programming, it is always a good idea to include a suitable number of comments and format the syntax to make the file easier to read. Similar to R and Bash, any text following a hash `#` until the end of the line is a comment.

Additionally, we may wish to add other steps to the process. For instance, rather than including all analysis steps in file `read.R`, we could modularize the code by developing a separate syntax file `linmod.R` for statistical analysis. Linear model and plot objects could be stored in `linmod.RData` for report production. We could also produce two separate reports for different audiences `report1.pdf` and `report2.docx` from R Markdown files. Finally, we would normally just place `r-rules.mk` in a standard directory like `~/lib` or `${HOME}/Library` so that we just use a single version of the Make pattern rules for any data analysis project we are working on. A revised `Makefile` is shown in Appendix B. A simple dependency graph may be produced from any `Makefile` using the `makefile2graph` program (Lindenbaum 2014). The resulting graph for the `Makefile` in Appendix B is shown in Figure 1.

The definitions in the revised `Makefile` are now outlined in detail.

Firstly, we could use the R syntax file `read.R` to read the data from a single data file `simple.csv`, perform data validation and cleaning; and also store the cleaned data in binary format. As discussed above, we also include the GNU Make pattern rules and so the last lines of the `Makefile` are:

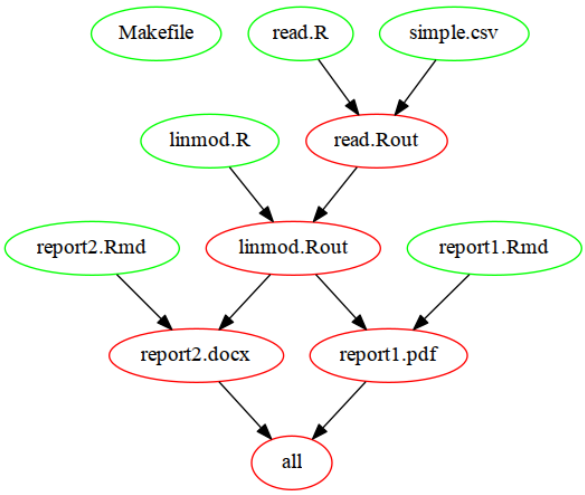

Figure 1: Dependency graph for a simple `Makefile` shown in Appendix B. This is for a simple data analysis project where data stored in `simple.csv` are read in using `read.R`, analyzed with `linmod.R` and reports produced from **rmarkdown** files `report1.Rmd` and `report2.Rmd`. The `.PHONY` target `all` is used to specify the final output files `report1.pdf` and `report2.docx`. Files in green ellipses are input files. Output or intermediate files in red ellipses are generated from dependency files by following the arrows.

```
read.Rout: read.R simple.csv
include ~/lib/r-rules.mk
```

Next, the syntax file `linmod.R` is run to summarize data, fit linear models and produce various plots. Note that in R, output from statistical analysis and plots can be stored in a binary file `linmod.RData` for further use. While this is somewhat hidden from view, in that several files can be produced by running R once, these objects can be used in further R and **rmarkdown** files. This is especially useful if analyses are computationally intensive and time consuming so they will not be rerun just to produce a report or two. See Section 5.6 for a method of explicitly defining multiple target files from the same dependency file(s).

The target file `linmod.Rout` depends on the syntax file `linmod.R` and the previous step of reading in the data which had target file `read.Rout`. The appropriate `Makefile` line is:

```
linmod.Rout: linmod.R read.Rout
```

Next, we define the dependencies for the two target report files with:

```
report1.pdf: report1.Rmd linmod.Rout
report2.docx: report2.Rmd linmod.Rout
```

Note that, in general, we often wish to produce several targets using a command `make all` or simply `make` given that `all` is the first, and hence the default, target.

What happens if there is actually a file called `all`? This could be problematic and so GNU Make provides *phony* targets which are not a file but are always made. At the top of the

`Makefile`, we specify a *phony* target `all` with `.PHONY: all`. The phony target has dependencies `report1.pdf` and `report2.docx`, either of which will be remade if they are not up to date.

```
.PHONY: all
all: report1.pdf report2.docx
```

For this project, if the data, R syntax or report files are changed then we can either type `make` in a terminal or press the build button in an IDE or programmer's editor to rerun the minimal number of steps to update all output. Setting up the build button for **RStudio** and **Emacs** is outlined in Appendix A.

Finally, note that we would normally put `simple.csv`, `read.R`, `linmod.R`, `report1.Rmd`, `report2.Rmd` and `Makefile` under version control like **git** (Loeliger and McCullough 2012) since other files can be recreated using Make. A minimal set of demonstration files for this simple example are available in the directory `simple_demo` at https://github.com/petebaker/r-makefile-definitions. While simply typing `make` will produce output files like those shown in Figure 1, it is recommended that data and syntax files are altered to explore what happens when employing GNU Make to regenerate downstream outputs.

Naturally, an approach this complicated may not be warranted if the project is very simple but most real world data analysis projects are rarely so trivial. When at the computer, we may spend the majority of our time interactively analyzing data and writing reports in an IDE. However, even for relatively simple projects, considerable efficiency is gained by declaring dependencies in Makefile(s) and automating rebuilding at an early stage of a data analysis project. See Section 6 for a more complex project involving multiple subdirectories and several `Makefile`s.

3.3. Variables and search paths

In addition to using predefined pattern rules, another way to simplify `Makefile`s and make them more portable is to use variables. A variable simply holds a text string and then substitutes the text when requested by Make. Variables may prove useful in the following situations.

Firstly, while we do not need to keep all data analysis project files in a directory and its subdirectories it helps to do so. Indeed, projects in **RStudio** and **Emacs** projectile are often set up with all files and subdirectories contained in a single directory which is the project directory. However, `Makefile`s are quite flexible and so we can specify directory names directly or as variables for later use if, for instance, data files or commonly used homegrown R functions are stored elsewhere. Also, we may wish to access system wide files such as those containing citations for bibliographies.

Suppose we keep several homegrown functions for cleaning data and producing specific formatted plots or tables in a file `myFunctions.R` that are not yet put into a package. We may just store the latest and greatest copy in a directory like `~/lib` or using the `HOME` variable instead of `~` to specify our home directory. This means we do not have several different versions floating around and we can also access the most recent version easily. If we retain the same directory structure for data analysis projects then exactly the same definitions, and hence the same `Makefile`, could even be used when we have different home directories on different computers.

In our `Makefile`, we can define the variable `MY_LIBRARY` using

```
MY_LIBRARY = ~/lib
```

We could use the variable `{MY_LIBRARY}` for referring to the directory in the `Makefile` or it could take a different value on a different machine or OS. For instance, to define a rule to clean data using the R syntax in `clean.R` and functions in `~/lib/myFunctions.R` we can employ the following declaration

```
clean.Rout: clean.R ${MY_LIBRARY}/myFunctions.R
```

whereas on Windows, we may set the variable with `MY_LIBRARY = C:/Library`. See Section 5.1 for methods to conditionally set variables.

Other common uses of variables are to set program options, specify (multiple) filenames, accessing environmental variables from the operating system and as so called automatic variables. These and other extensions are further discussed in Section 4.4.

However, if our main aim is just to allow Make to search for dependency files then Make provides the variables `VPATH` and `vpath`. Both variables define a list of directories to search if a dependency file is not found. Directory names can be separated by whitespace or colons (`:`). However, on Windows semi-colons (`;`) are the preferred separators since the colon is used after drive letters (e.g., `C:`). `VPATH` provides directories to search for any missing dependency file whereas `vpath` is to find files with particular extensions using a `%` as a wildcard.

For example, if we had R functions for cleaning data in two files, we can set a variable for the function filenames and search for R syntax files not found in the current directory in `~/lib` using the `vpath` directive as follows:

```
R_FUNCTIONS = myCleanFuncs_01.R cleanDataFuncs_02.R
vpath %.R ~/lib
```

Similarly, we can also set up variables for data files with various filename extensions in a common directory `~/Documents/Project3/Data` using

```
DATA_FILES = simple1.csv simple2.dta simple3.sav simple4.RData
VPATH = ~/Documents/Project3/Data
```

Finally, we can define a rule to clean the data with appropriate R syntax and data files as follows:

```
clean.Rout: clean.R ${R_FUNCTIONS} read.Rout
read.Rout: read.R ${DATA_FILES}
```

which searches the appropriate directories for dependency files when GNU Make to update targets.

Note that if our customized functions are in `myCleanFuncs_01.R` and `cleanDataFuncs_02.R` in directory `~/lib` and data files are in the directory `~/Documents/Project3/Data` these directories will need to also be defined in the R syntax files. These directory names can be either hardwired in syntax or extracted from the relevant environmental variables as outlined in Section 5.4.

If we were using LaTeX for typesetting reports then we could use the environmental variable `BIBINPUTS` to look for changed bibliography files which are used automatically.

Important note. *If a file is likely to be found in multiple directories in the* `VPATH` *or* `vpath` *list then it is better to hardwire the location to avoid updating targets based on the wrong dependency file.*

4. Using pattern rules for R and related software

As outlined in Section 3.1, we can define the commands to update a particular target file given the appropriate dependency files. While we could define the exact commands to be carried out for every individual target, this would be very inefficient. Instead GNU Make provides a way to define pattern rules. Such rules mean that we can use wildcards to define general rules to carry out a series of commands and also provide program and/or compilation options where necessary. For instance, GNU Make provides rules to make an object file with name ending in `.o` from a C program file with name ending in `.c`. We provide pattern rules for R and related programs. The files `r-rules.mk` and `r-rules-functions.mk` can be installed system wide by copying them to a suitable system directory like `/usr/include` or `/usr/local/include` to allow GNU Make to `include` them without specifying the directory.

As an example of the process involved in defining a pattern rule, we outline in Section 4.1 how to explicitly define a Make rule by hardwiring filenames to create a `.Rout` output file from a `.R` syntax file. Next, the generic pattern rule in `r-rules.mk` to generate a `.Rout` output file from any `.R` file is outlined in Section 4.2. Section 4.3 provides a summary of pattern rules available in `r-rules.mk` for data analysis and reporting mainly using R, **rmarkdown** and `Sweave` and also rules for SAS, Stata, PSPP, LaTeX Beamer, Perl and Python. Finally, some further hints to simplify `Makefile` use are briefly described in Section 4.4.

4.1. Making targets explicitly

Without pre-specified pattern rules, we must specify individual commands to produce target files. These rules define the target and dependency files on the first line and are followed by specifying the commands we would actually type at the command prompt. For instance, to produce `read.Rout` from `read.R` we could define a rule with all filenames hardwired with the following two commands.

```
read.Rout: read.R simple.csv
	R CMD BATCH --vanilla read.R
```

As noted previously, the ↦ symbol represents a single TAB character at the start of the line. Commands that we would type at the prompt are specified after the TAB character ↦. Several lines of commands, each with a TAB character ↦ at the beginning may be provided.

However, a `Makefile` of explicit rules will quickly become very verbose and require a lot of maintenance. To circumvent this, pattern rules are provided for standard languages like C and C++ and are easily written for other languages.

4.2. Pattern rules for making targets in general

Rather than needing to define these commands every time, GNU Make provides a mechanism for user defined pattern rules which only need to be defined once. A simple pattern rule can be written to generalize the two line rule in Section 4.1.

An initial pattern rule to produce a `.Rout` output file from `.R` syntax file uses the `%` as a wildcard and automatic variable `$<` which substitutes the value of the first prerequisite dependency file. Wildcarding and automatic variables mean we do not have to explicitly specify individual filenames but can reuse the rule.

```
%.Rout: %.R
	R CMD BATCH --vanilla $<
```

This definition means that the basename of the target `.Rout` and `.R` syntax files are the same and the rule always works even if specific files are not defined in the `Makefile`. This takes care of the filenames but the program and options are still hardwired. For example, if we wished to use a development version of R or different program options then we really do not want to rewrite the rule each time.

Instead, as is common when using GNU Make, we can specify the programs and options with variables as outlined in Section 3.3. We instead define variables `R`, `R_FLAGS` and `R_OPTS` in `r-rules.mk` with

```
R          = R
R_FLAGS    = CMD BATCH
R_OPTS     = --vanilla
```

and the pattern rule as

```
%.Rout: %.R
	${R} ${R_FLAGS} ${R_OPTS} $<
```

and so by changing these variables, we can change program versions and options.

For instance, if we use the variable `R` to define the R program then it can be reset with definitions like `R = /usr/bin/R`, `R = /usr/bin/Rscript` or `R = /usr/local/bin/R-devel`.

Assuming the file `r-rules.mk` is in `~/lib` (recommended) and we wish to override the standard R program with a development version `~/bin/R-devel` then we have several alternatives.

Firstly, we can `include ~/lib/r-rules.mk` in our `Makefile` and redefine the variable `R` with

```
read.Rout: read.R simple.csv

include ~/lib/r-rules.mk
R = ~/bin/R-devel
```

Since GNU Make reads the `Makefile` sequentially then the last value of variable `R` is used and so is defined *after* including `r-rules.mk`.

Secondly, we can define the variable at the top of the `Makefile`, or anywhere else, with

```
R := ~/bin/R-devel
```

where the `:=` assignment operator means that `R` is a simple text variable which is expanded immediately and does not change even if redefined with `R = /some-path/R`.

Thirdly, we can make very sure that the correct variable value is used throughout the `Makefile` by using the `override` command. This can either be done by using the `-e` option, in the case of existing environmental variables, when invoking Make or including a line in the `Makefile`, such as,

```
override R=~/bin/R-devel
```

Finally, in a terminal, we can define the variable `R` when invoking Make at the command line with

```
$ R=~/bin/R-devel make
```

noting that this sets the variable then invokes GNU Make with the variable `R` overriding any definitions in the `Makefile`.

4.3. Rules defined in `r-rules.mk`

Table 1 provides an overview of GNU Make pattern rules available in `r-rules.mk`. Further details, such as the commands run, defaults and variables for changing programs and options, may be found in Appendix C.

For instance, in Table 1 we see that from an R syntax dependency file with suffix `.R` or `.r`, we can produce either a standard output text file with `.Rout` suffix or notebook style documents using **rmarkdown** with suffixes of either `.html`, `.docx` or (if LaTeX is installed) `.pdf`. Specific defaults and variables to alter these are shown in Table 3 in Appendix C.

From Table 3, `.Rout` text files are produced using the `R CMD BATCH --vanilla` command, as outlined in Section 3.2. However, the command can be changed by redefining the variables `R`, `R_FLAGS` and `R_OPTS` which are by default set to `R`, `'CMD BATCH'` and `--vanilla`, respectively. As outlined previously, we could set the variable `R` to be a development version of R or change the `--vanilla` option by changing `R_OPTS`.

We may wish to alter defaults generally by using variables in the `Makefile`. By default, the variables for `HTML` output are set up as

```
RSCRIPT = Rscript
RSCRIPT_OPTS = --vanilla
RMARKDOWN_HTML_OPTS = "html_document"
RMARKDOWN_HTML_EXTRAS =
```

with pattern rule (which is actually defined on one line starting with a TAB)

```
%.html: %.Rmd
  ${RSCRIPT} ${RSCRIPT_OPTS} -e "library(\"rmarkdown\");
   render('$@:.html=.Rmd', $RMARKDOWN_HTML_OPTS $RMARKDOWN_HTML_EXTRAS)"
```

Substituting in the variables for `report.Rmd` would run

```
Rscript --vanilla -e "library(\"rmarkdown\");
   render('report.Rmd',  \"html_document\")"
```

Target	Dependency	Purpose
R batch		
.Rout	.R	R batch mode
R Notebooks		
.html	.R or .r	Compile HTML Notebook
.docx		Compile Word document
.pdf		Compile PDF document
R Markdown documents		
.html	.Rmd or .rmd	HTML
.docx		Word document
.rtf		Rich text format document
.odt		Libre Office document
.pdf		PDF document
R Markdown presentations		
_ioslides.html	.Rmd or .rmd	IO Slides presentation
_slidy.html		Slidify presentation
_beamer.pdf		Beamer PDF presentation
_tufte.pdf		Tufte style PDF presentation
R Sweave documents		
.pdf	.Rnw or .rnw	PDF[1]
.tex		LaTeX file
R syntax from R Markdown/Sweave		
-syntax.R	.Rmd or .rmd	Extract R syntax
-syntax.R	.Rnw or .rnw	
R Sweave Beamer		
_Present.pdf	.Rnw or .rnw	Beamer presentation[2]
_Article.pdf	.Rnw or .rnw	Beamer article
_Notes.pdf	.Rnw or .rnw	Beamer notes
_Handout.Rnw	.Rnw or .rnw	intermediate .Rnw file
_Handout.pdf	_Handout.Rnw	handout 1 slide per page
-2up.pdf	_Handout.pdf	handout 2 slides per page
-3up.pdf	_Handout.pdf	handout 3 slides per page
-4up.pdf	_Handout.pdf	handout 4 slides per page
-6up.pdf	_Handout.pdf	handout 6 slides per page

[1]By default **knitr** format is assumed. Set SWEAVE_ENGINE=Sweave if .Rnw files are in Sweave format.
Section 5.2 provides an alternate approach when .Rnw files are a mixture of both formats.
[2]Beamer handouts with varying numbers of slides can be produced directly from a .Rnw R Sweave file but it may be more efficient to produce the intermediate _Handout.pdf file which is then used to produce multiple slides per page handouts using the program **pdfjam** (Linux and macOS).

Table 1: An overview of pattern rules available in r-rules.mk to produce text, HTML, Word or PDF output files from .R syntax, .Rmd **rmarkdown** or .Rnw R Sweave files. For example, target FILE_ioslides.html can be produced by defining FILE.Rmd in a Makefile or by typing make FILE_ioslides.html at the command prompt.

We may wish to set options generally rather than for every file separately. For instance, for `pdf` output on Linux some plots are missing unless `fig_crop` is set to `FALSE`.

We can alter output styles Make pattern rule definitions from `r-rules.mk` for processing **rmarkdown** files by using variables like `RMARKDOWN_HTML_OPTS` and `RMARKDOWN_PDF_OPTS`. By default, these variables are set to `"html_document"` and `"pdf_document"`, respectively. Hence, any specifications set in the YAML header of each individual **rmarkdown** file will be employed. If instead, these variables were set to `html_document()` or `pdf_document()` then specific **rmarkdown** options may be set by specifying them in the definition but the YAML header settings will be ignored. For instance, if we wish to always produce

```
RMARKDOWN_PDF_OPTS = pdf_document(fig_crop=FALSE)
```

and of course other options like figure captions, figure height and width may also be set. Note that by defining the `pdf_document` in this way that all options not defined are set to their default parameters since the YAML PDF document header information is ignored. The pattern rule is similar.

```
%.pdf: %.Rmd
   ${RSCRIPT} ${RSCRIPT_OPTS} -e \
  "library(\"rmarkdown\");
  render('$@:.pdf=.Rmd', ${RMARKDOWN_PDF_OPTS} ${RMARKDOWN_PDF_EXTRAS})"
```

and a similar rule is employed to produce documents for `.R` syntax files.

```
%.pdf: %.R
   ${RSCRIPT} ${RSCRIPT_OPTS} -e \
   "library(\"rmarkdown\");
   render('$@:.pdf=.R', ${RMARKDOWN_PDF_OPTS} ${RMARKDOWN_PDF_EXTRAS})"
```

Further details of variables that we can modify are shown in Table 3. Using **rmarkdown**, the rules for R syntax files and are the same as for `.Rmd` **rmarkdown** files. When a target is made from a `.Rmd` **rmarkdown** file, two lines are passed to `Rscript` by using the `-e` option, namely the `library("rmarkdown")` statement and the `render()` statement with appropriate options set by GNU Make. This sets the options for all target files.

Note that the `Rscript` command is actually all on one line and just broken here for convenience. To aid legibility in GNU Make, long lines can be continued using the backslash character (\) as is commonly used in a Bash script and so the pattern definitions presented above will also work as long as the string enclosed in double quotes is specified on one line.

These options could instead be set in the YAML header of an `.Rmd` **rmarkdown** file as shown in Listing 1. When specified as a YAML header in a file, these options override those specified in the `render()` command defined in `r-rules.mk`. In a `.R` R syntax file, the YAML header can be specified as **roxygen2** (Wickham, Danenberg, Csárdi, and Eugster 2020) comments where the header lines start with a `#'` or `##'`.

Producing Beamer presentations, slides, handouts, notes and articles from the same R `Sweave` (`.Rnw`) file is straightforward using the rules outlined in Table 1. However, there are caveats. Firstly, while standard LaTeX and Beamer are used to produce slides, Beamer articles and notes from a single Beamer `Sweave` (`.Rnw`) file, **pdfjam** is employed to produce multiple slides

```
---
title: "Summary statistics for 'simple.csv'"
author: Peter Baker
date: "Wed Oct 26 17:17:52 2016"
output:
  html_document:
    toc: true
    theme: united
  pdf_document:
    toc: true
    highlight: zenburn
---
```

Listing 1: YAML header at the top of **rmarkdown** file specifying output options. Options specified in the **rmarkdown** file will override `Makefile` variables like `RMARKDOWN_HTML_OPTS` and `RMARKDOWN_PDF_OPTS`. In R syntax files the YAML header can be specified using **roxygen2**.

Target	Dependency	Purpose
SAS batch		
`.lst`	`.sas` or `.SAS`	SAS batch mode
STAT batch		
`.log`	`.do` or `.DO`	Stata batch mode
PSPP batch		
`.list`	`.sps` or `.SPS`	PSPP (rather than SPSS) batch mode
`.pdf`		PDF document
`.odt`		Libre Office document
Perl[1]		
`.txt`	`.pl` or `.PL`	Perl script in batch mode
Python[1]		
`.txt`	`.py` or `.PY`	Python script in batch mode

[1]Perl and Python do not have default output extension so the default is arbitrarily `txt`. Use the variables `PL_OUT_EXT` or `PL_OUT_EXT`, respectively to change the file extension, for example to change the Perl output extension run Make with `PL_OUT_EXT=out make myFile.out`

Table 2: Pattern rules available in `r-rules.mk` to produce output from running SAS, Stata, PSPP, Perl and Python in `batch` mode. Note that a particular version can be set using the `Makefile` variables `SAS`, `STATA`, `PSPP`, `PYTHON`, `PERL`. Also, because Perl and Python do not have default output filename extensions these default to `txt` and can be set with variables `PL_OUT_EXT` and `PY_OUT_EXT`. For example, target `FILE.txt` can be produced from `FILE.py` by typing `make FILE.txt` at the command prompt.

per page. The **pdfjam** program is developed on Linux and is available as part of standard **texlive** distributions and so readily available via the usual Linux package managers. It is also available for macOS as part of the standard **MacTeX texlive** package. However, installation for Windows may be problematic although some success is reported via **cygwin**. Please note also that header files like `preamblePresent.Rnw`, `preambleSlides.Rnw` and so on are supplied at https://github.com/petebaker/r-makefile-definitions and should be modified for personal preferences. These contain the appropriate `documentclass` statement used and so setup is required as outlined using

```
make help-beamer
```

Make rules for Beamer in `r-rules.mk` are currently under development and may change if more generic tools like the **beamerswitch** LaTeX package (Ball 2017) become available. Since **beamerswitch** and the rules defined here both use **latexmk** (Collins 2018) then conflicts may occur and so the safer option of not employing **beamerswitch** has been adopted. If this changes, it is envisaged that the relevant Make rules are unlikely to change much but the setup and pattern rules behind the scenes may be simplified.

Table 2 provides an overview of GNU Make pattern rules available in `r-rules.mk` for several other languages that may be used in data analysis projects. Note that default behavior may be changed by setting specific options variables `SAS_CFG`, `SAS_OPTS`, `STATA_OPTS`, `PSPP_OPTS`, `PL_OPTS` and `PY_OPTS`. Due to computing constraints of not having all programs available on all platforms, these rules are not as well tested as the R related rules.

4.4. Streamlining the make process

Firstly, rather than running `make` and inadvertently overwriting files it can be prudent to just print out what would happen with `make -n`. This allows us to adjust the variables, targets and dependencies to ensure the appropriate commands are run.

In particular,

- `make -n` will print the commands which would be executed but not actually run them,
- `make -j [n]` will run n jobs (commands) simultaneously to speed up execution,
- `make -p` prints the data base of rules and variables and then executes the commands,
- `make -d` prints debugging information,
- `make -v or --version` prints the version of Make, and
- `make -k` means keep going even if there are errors, which is useful for setting up rules in programmer's editors and IDE.

A brief summary of available options for the version you are currently running are available from the manual (`man make`) page. Much more detailed information can be obtained via the info documentation (`info make`).

Secondly, we can simplify `Makefile`s by using variables, whether these are user defined or so-called automatic variables. User defined variables can be set at the top of a `Makefile` and adjusted to the situation. The variable is replaced with the text throughout the `Makefile`.

Variables are ideal for filenames or parts of filenames that may be used in several places because you only need to change one occurrence, namely the variable definition. For instance, we can use a variable to specify several filenames at once (such as `DATAFILES` below) or to specify a particular target like `REPORT` below which could subsequently be updated if an alternative is required. Each month the variables can be redefined to produce a new report.

At the top of the `Makefile`, we could use the two variables to define data files and the **rmarkdown** file `Report_january_2017.Rmd`.

```
DATAFILES = dec_2016.csv nov_2016.dta jan_2017.sav sample_20170101.RData
REPORT = Report_january_2017.pdf
```

The rest of the `Makefile` would not need to be changed although of course the **rmarkdown** file for the report may need to be altered.

```
${REPORT}: ${@:.pdf,.Rmd} analysis.Rout
analysis.Rout: analysis.R read.Rout
read.Rout: read.R ${DATAFILES}
```

Note the use of `$@:.Rmd,.pdf` to change the first prerequiste to be `Report_january_2017.Rmd` from the target `Report_january_2017.pdf`. The target variable ending in .pdf is `$@` and the construct `:.pdf,.Rmd` substitutes the suffix `.Rmd` for the `.pdf` suffix in the target file.

When writing `Makefile`s, useful automatic variables are:

- `$@` target filename,
- `$<` filename of the first prerequisite,
- `$?` the names of all prerequisites newer than the target separated by spaces, and
- `$*` the filenames of all prerequisites separated by spaces but with duplicates removed.

See Mecklenburg (2004) or Graham-Cumming (2015) for further details.

Finally, it may prove useful to define the file extension obtained from running R syntax files. This is easily achieved by setting a variable for the output type as follows. Consider, Appendix B which provides a `Makefile` for a simple project outlined in Section 3.2. Rather than having text `.Rout` files as output throughout, we can make a default file extension (say `Rout`, `pdf`, `docx`, `odt` or `html`) by defining a variable such as `R_OUT_EXT` at the top of the `Makefile` and change all the appropriate file extensions.

The output file type variable can be set using something like

```
## set R output by (un)commenting appropriate lines
## R_OUT_EXT=Rout
R_OUT_EXT=pdf
## R_OUT_EXT=html
## R_OUT_EXT=docx
```

The appropriate lines in the `Makefile` in Appendix B for the simple example in Section 3.2 would become

```
linmod.${R_OUT_EXT}: linmod.R read.${R_OUT_EXT}
read.${R_OUT_EXT}: read.R simple.csv
```

Naturally, using variables is a matter of personal preference in that there is a danger of over-complicating the `Makefile` if we overuse them. The `Makefile` may be very flexible but it could become difficult to follow because we have to constantly refer back to variable definitions. On the other hand, for a complex project with many steps, if we were producing monthly reports where just the name of the report markdown file and the data filenames changed then the same `Makefile` could be employed merely by changing the two variables at the top of the `Makefile`.

5. Extensions, limitations and work-arounds

GNU Make has the reputation of being problematic in some situations but has been slowly updated and extended. These limitations are often due to the fact that all `make` commands are executed by the shell. Hence, there may be inherent problems with spaces in filenames, underscores (_) in variable names, difficulties with quoting single (') or double apostrophes (") in strings, substitution of strings containing dollars ($) which may also be used to access variables and finally the use of the star (*) which may be a wildcard in the shell. Hence, care should be taken with these symbols.

Also, GNU Make has the reputation for not handling recursive `make`s well although this has been built in since version 4.0 (see Graham-Cumming 2015, p. 153). Details of both recursive `make` and alternatively non-recursive `make` for multi-directory projects are provided in Sections 6.2 and 6.3.

Casual users of Make may also form the impression that it may not be very flexible but this is not the case. While it can be generally seen as primarily a simple text processing language, GNU Make can be extended using functions. Many built-in functions are provided and user-defined functions are relatively easy to write and reuse. Details of several extensions are provided here but many others are available. Graham-Cumming (2015) has extended the built in GNU Make functions to provide the GNU Make Standard Library (**GMSL**) at `https://gmsl.sourceforge.io/`. These extensions expand GNU Make functions to include logical operators, character manipulation, associative arrays, integer arithmetic and much more.

Section 5 outlines useful functions and extensions to help improve Make workflow. Conditional programming is outlined in Section 5.1, redefining variables for different targets using the same rule in Section 5.2, reusing `Makefile`s in Section 5.3, using `Makefile` variables in R in Section 5.4 and using wildcards in Make itself are provided in Section 5.5. Writing rules for multiple targets is outlined in Section 5.6 and is followed by tips on spaces in filenames and avoiding tabs in single line definitions in Sections 5.7 and 5.8, respectively. Finally, since there are certain situations where there is no easy way to let Make know if targets need to be updated, forcing all targets to be remade is discussed in Section 5.9.

5.1. GNU Make `ifeq` and other functions

Sometimes we may need to customize settings for different computers or different situations. For instance, if we work on different OS'es and we have our project under version control

then we may need to use one `Makefile` for all machines. We can do this using the GNU Make `ifeq` command. For example, the following excerpt sets the `vpath` variable to find LaTeX bibliography files and also set the preferred Make program on Windows, macOS or Linux.

This can be particularly important on Windows where filenames and separators can be quite different to macOS and Linux.

Windows users may wish to use something like

```
ifeq ($(OS),Windows_NT)
   OS_detected := Windows
else
   OS_detected := $(shell uname -s)
endif
```

and then use the following to detect Windows and set variables accordingly.

```
ifeq ($(OS_detected),Windows)
   vpath %.bib C:/Users/peterwork2/texmf/bibtex/bib/myfiles
   MAKE=C:/Rtools/bin/make
endif
```

Mac users might want to use the macOS definitions of path to search for BibTeX files and GNU Make as follows

```
ifeq ($(OS_detected),Darwin)
   vpath %.bib /Users/peterwork/Library/texmf/bibtex/bib/myfiles
   MAKE=gmake
endif
```

Finally, to check for Linux and run Linux commands use

```
ifeq ($(OS_detected),Linux)
   vpath %.bib /home/peter/texmf/bibtex/bib/myfiles
   MAKE=make
endif
```

Another common use of `ifeq` is to use different rules in certain situations. For instance, as outlined in Table 1, `.Rnw` files containing LaTeX syntax with R and other language chunks may either be in `Sweave` or **knitr** (Xie 2020, 2015) format. The GNU Make rules provided in `r-rules.mk` assume by default that a `.Rnw` file is in **rmarkdown** format. However, if the variable `SWEAVE_ENGINE = Sweave` then `Sweave` is used instead. This is achieved by first defining rules for R Markdown. Next rules are redefined if `SWEAVE_ENGINE` is set to `Sweave`. In particular, to produce PDF from a `.Rnw` these rules are specified using:

```
## define R Markdown commands to produce PDF from .Rnw
%.pdf: %.Rnw
RSCRIPT = Rscript
RSCRIPT_OPTS = --vanilla
RMARKDOWN_PDF_OPTS = ()
```

```
   $RSCRIPT $RSCRIPT_OPTS -e "library(\"knitr\");knit2pdf('${@:.pdf=.Rnw}')"
## define Sweave commands to produce PDF from .Rnw
RSWEAVE = ${R} CMD Sweave
RSWEAVE_FLAGS=
## if Sweave then redefine the pattern rule
ifeq ($(SWEAVE_ENGINE), Sweave)
## produce pdf from .Rnw etc
%.pdf: %.Rnw
   ${RSWEAVE} -pdf $< ${RSWEAVE_FLAGS}
endif
```

An alternative approach for `.Rnw` files, which allows a mixture of `Sweave` and **knitr** formats is outlined in Section 5.2.

GNU Make has many other useful functions that are well documented in the Make documentation and Mecklenburg (2004) and Graham-Cumming (2015). These include `subst` for substituting text strings, `wildcard` for finding file and directories based on wildcarding, `call` for calling functions and `shell` for passing commands to the OS.

5.2. Target and pattern specific variables

Generally, due to the way `Makefile`s are processed, GNU Make variables just have one value when running `make`. Firstly, the `Makefile` is processed to assign and expand variables and build a dependency graph. In the second phase, the dependency graph is analyzed and traversed so that scripts and commands are executed in the desired order. In this second phase, all variables are fixed since such processing was carried out in the first phase (Mecklenburg 2004).

However, we may wish to redefine a variable for a single rule or pattern. For instance, consider an example where we wish to pass different parameters to a **rmarkdown** file in order to obtain different reports or targets using the same Markdown file.

Consider the pattern rule to produce a PDF file from a **rmarkdown** file. **rmarkdown** options may be set with the variables `RMARKDOWN_PDF_OPTS` (default: `"pdf_document"`) and `RMARKDOWN_PDF_EXTRAS`

```
%.pdf: %.Rmd
   ${RSCRIPT} ${RSCRIPT_OPTS} -e \
   "library(\"rmarkdown\");render(\"${@:.pdf=.Rmd}\",
   ${RMARKDOWN_PDF_OPTS} ${RMARKDOWN_PDF_EXTRAS})"
```

By default, `RMARKDOWN_PDF_EXTRAS` is blank. Normally we might set this variable globally by means of simply defining it as usual with something like:

```
RMARKDOWN_PDF_EXTRAS =  ", clean = FALSE"
```

However, in a `Makefile` we can also use target specific variables so that this variable is defined separately for different targets `special1.pdf` and `special2.pdf` as follows:

```
RMARKDOWN_PDF_EXTRAS = ", clean = FALSE"
special1.pdf: \
```

```
    RMARKDOWN_PDF_EXTRAS += ", params = list(v1 = \"a\", v2 = \"b\")"
special1.pdf: special.Rmd
special2.pdf: \
    RMARKDOWN_PDF_EXTRAS += ", params = list(v1 = \"C2\", v2 = \"D2\")"
special2.pdf: special.Rmd
```

Note that initially `RMARKDOWN_PDF_EXTRAS` sets the `clean` option to `FALSE`. The `+=` assignment operator is then used to add extra text to the `RMARKDOWN_PDF_EXTRAS` variable on a target specific basis and this is set separately for each target in addition to specifying the same dependency file on a separate line. In this way, different parameters are set for each target report but the same **rmarkdown** file is employed.

5.3. Reusing `Makefiles`

In general, we can use old `Makefiles` and edit these to change targets and dependencies. Alternatively, we may use a nearly identical `Makefile` for similar projects and simply change a few variables. As an example of reusing `Makefiles`, we use the following `Makefile` for course notes that change each year. Course notes are in a `Sweave` format `.Rnw` files processed by **knitr** with appropriate data and BibTeX files. These files are simply LaTeX files containing R code chunks which, when processed, insert both the R syntax and R output to produce a PDF file.

```
TOPIC = Survival02
DATA = whas500.RData
BIBFILES = survival.bib
vpath %.bib ~/texmf/bibtex/bib/myfiles/
```

Next, the phony `all` target along with course notes are defined with

```
.PHONY: all
all:  ${TOPIC}.pdf
${TOPIC}.pdf: ${@:.pdf=.Rnw} ${TOPIC}-functions.Rnw ${DATA} ${BIBFILES}
```

noting that these definitions are common to all topics and do not need to be altered. Finally, `r-rules.mk` is included.

The `.Rnw` input files were originally written to be processed using `Sweave`. However, given that **knitr** extends some capabilities of `Sweave`, especially for graphics, these were recently converted via the package `knitr::Sweave2knitr` and a few simple changes to make them **knitr**-compatible. Make uses the R package **knitr** to produce both student notes in PDF format and an R syntax file. Also, the R syntax file `${TOPIC}-syntax.R` was produced from the `.Rnw` file when processed by using `Stangle` whereas now `purl` is employed. For each topic in the course, we place all relevant files in a directory and use a similar `Makefile` with the appropriate value of the `TOPIC`, `DATA` and `BIBFILES` variables.

Note that the file `survival.bib` is available in the user's standard directory for BibTeX files and so the directory to search is specified with `vpath` as outlined in Section 3.3. Also, in the `.Rnw` file we can use `purl` directly to produce the syntax file `Survival02-syntax.R` noting that we can set `documentation = 0` to remove chunk labels with the following chunk.

```
<<include = FALSE, purl = FALSE>>=
system("Rscript -vanilla -e \"knitr::purl('Survival02.Rnw',
       'Survival02-syntax.R', documentation = 0)\"")
@
```

5.4. Setting and using Make variables in R

When automating regular reports or routine analyses it is straightforward to set environmental variables in either a Bash script or directly in a `Makefile`. For instance, the following definitions can be used to set month and year in the `Makefile`.

```
YEAR = 2010
MONTH = january
DATA = myData_${YEAR}-${MONTH}.csv
REPORT = myReport_${YEAR}-${MONTH}.pdf
```

and the target file is produced with

```
${REPORT}.pdf: ${@:.pdf=.Rmd} ${DATA}
```

To read this in R or an **rmarkdown** file, it is straightforward to use the variable values defined in the `Makefile` using the R function `Sys.getenv`. Strings obtained can then be used to set titles, filenames and so on.

To set the title for report use

```
R> year <- Sys.getenv("YEAR")
R> month <- Sys.getenv("MONTH")
R> myTitle <- paste0("Regular report for ", stringr::str_to_title(month),
+    ", ", year)
```

and we can read in the data with

```
R> dataCsv <- Sys.getenv("DATA")
R> myDat <- read.csv(dataCsv)
```

Also, note that we can use a similar approach for the example in Section 5.3. Instead of hardwiring filenames in the `.Rnw` file for each topic, we can simply use `Sys.getenv`. In each `.Rnw` file, we then `source` a common syntax file `setup.R` for all topics in the course to load packages, set options and produce the appropriate R syntax files. When sourcing the file we would use **knitr** chunk options `include = FALSE, purl = FALSE` to suppress including the source command in the notes and the R syntax file, respectively.

In the file `setup.R` we set up filenames with

```
R> TOPIC <- Sys.getenv("TOPIC")
R> RNW_FILE <- paste0(TOPIC, ".Rnw")
R> SYNTAX_FILE <- paste0(TOPIC, "-syntax.R")
```

and the **knitr** environment with

```
R> library("extrafont")
R> library("knitr")
R> knit_theme$set("peaksea")
R> opts_chunk$set(fig.path = "tmp/tmp", split = FALSE, highlight = TRUE,
+    warning = FALSE, tidy = FALSE, background = "grey95",
+    fig.width = 5, fig.height = 5, comment = "\#:")
```

To write the R syntax file and call `purl` directly and set `documentation = 0` to remove chunk labels then we use

```
R> system(paste0("Rscript --vanilla -e \"knitr::purl('", RNW_FILE, "', '",
+    SYNTAX_FILE, "', documentation = 0)\""))
```

Finally, we wish to write the first few lines of syntax file to include filename and date-time with

```
R> LINES <- c(paste("## File:", SYNTAX_FILE), paste("## Date:",
+    DATETIME <- date()), "##", readLines(SYNTAX_FILE))
R> writeLines(LINES, SYNTAX_FILE)
```

5.5. GNU Make wildcard function

Wildcards can be used when defining pattern rules by using a percent (%) as shown in Section 4.2. However, these should not be used for defining non-pattern rules but instead the `$(wildcard)` function is preferred. GNU Make's globbing function is `$(wildcard)`. It can be useful in getting lists of files and is pretty similar to Unix wildcards but with some important limitations. For instance, the following would return a list of all `csv` data files in a directory.

```
CSV_FILES = $(wildcard *.csv)
```

While the `$(wildcard)` function contains standard globbing patterns like `*` which matches 0 or more characters, `?` matches a single character and `$[...]$` matches characters or ranges of characters as we might expect, it will prove problematic if filenames contain spaces since these will look like separate filenames separated by a space. While GNU Make can be made to handle spaces in filenames, this may require considerable effort.

5.6. Multiple targets

> A fundamental law of GNU Make physics is that each rule builds one and only one file (called a target). (Graham-Cumming 2015)

Graham-Cumming refers to these Make rules as atomic rules. Put simply, R often creates more than one output file and this usually creates no need for writing separate Make rules. However, sometimes we need to build this into our `Makefile`s so that it is clear that more than one target is updated and that GNU Make behaves appropriately. We do not want to run R separately for each target but just run R once.

Often we produce multiple outputs from a single `.R` file. For instance, we could produce several graphics files for collaborators. While we may write a separate R syntax file to produce each graphic this may prove to be highly inefficient.

A first try

Consider the following (incorrect) attempt at producing three graphics files from the same R syntax file `plots.R`.

```
.PHONY: all
all: plots.Rout plots0.png  plots1.pdf  plots2.jpg

plots.Rout plots0.png  plots1.pdf  plots2.jpg: plots.R simple.RData
	${R} ${R_FLAGS} ${R_OPTS} $<
```

At first glance, it might look like Make has four targets and so R only needs to be run once. What actually happens is that Make expands the rule to be four separate rules.

```
plots.Rout: plots.R simple.RData
plots0.png: plots.R simple.RData
plots1.pdf: plots.R simple.RData
plots2.jpg: plots.R simple.RData
```

So `make -n` will indicate that all four rules and should run R as follows:

```
$ make -n

R CMD BATCH --vanilla plots.R
R CMD BATCH --vanilla plots.R
R CMD BATCH --vanilla plots.R
R CMD BATCH --vanilla plots.R
```

In reality, this may still work as desired but only if the syntax is very simple and all four targets are essentially produced at the same time. After running the first rule, all targets are up to date and so subsequent rules are not run. The result is not that predicted by `make -n` but instead:

```
$ make

R CMD BATCH --vanilla plots.R
```

However, even for this simple case, there still may be problems if we use the `-j` option to run several jobs in parallel since Make may be tricked into running all four rules at once which is clearly not what we want because R will be run four times simultaneously rather than once.

```
$ make -j4
```

```
R CMD BATCH --vanilla plots.R
R CMD BATCH --vanilla plots.R
R CMD BATCH --vanilla plots.R
R CMD BATCH --vanilla plots.R
```

Other problems can also occur but these are not outlined here. An alternative is to use so called static pattern rules but these may suffer the same problems and, in addition, this limits the target files to having fairly similar names because of the use of wildcards when setting up the rules.

A better way: Using a sentinel file and the atomic function

In essence, the solution is to use the phony target `all` defined previously but to also include `r-rules-functions.r` before defining the following rule to make the multiple targets

```
$(call atomic,plots.Rout plots0.png plots1.pdf plots2.jpg,\
        plots.R simple.RData)
	${R} ${R_FLAGS} ${R_OPTS} $<
```

which, in turn, calls the function `sentinel`, both of which are in `r-rules-functions.r` defined. While users of these functions do not need to know how or why this works, this process is now outlined in some detail.

Graham-Cumming (2015) provides a solution to producing multiple targets by defining a sentinel or indicator file `.sentinel` with

```
plots.Rout plots0.png  plots1.pdf  plots2.jpg: .sentinel
	@:
```

and the rule for a sentinel file

```
.sentinel: plots.R simple.RData
	${R} ${R_FLAGS} ${R_OPTS} $<
	touch .sentinel
```

This uses the unusual Make `@:` command which actually does nothing. Due to this rule construction, the rule indicates whether any one of the target files is older than the indicator file `.sentinel`. If it is, then Make rebuilds the file `.sentinel` and so regenerates all four target files. It does so by running R and then `touch`ing the file `.sentinel` which sets the creation time of the file to the current time or creates an empty file if it does not already exist.

However, there is a catch. If we delete one of the target files and forget to delete the `.sentinel` file then Make will not rerun R. A more general solution is outlined in Graham-Cumming and shown here for a particular example `plots.R`. Note that since the sentinel file is made of target filenames, several calls to atomic can be made in a `Makefile` to define different multiple targets by calling the `atomic` functions with different arguments.

The definitions for `sp`, `sentinel` and `atomic` are provided in `r-rules-functions.mk` on https://github.com/petebaker/r-makefile-definitions. The rules provided are:

```
sp :=
sp +=
sentinel = .sentinel.$(subst $(sp),_,$(subst /,_,$1))
atomic = $(eval $1: $(call sentinel,$1) ; @:)$(call sentinel,$1): \
   $2 ; touch $$@ $(foreach t,$1,$(if $(wildcard $t), ,$(shell rm -f\
   $(call sentinel,$1))))
```

The `atomic` function calls the `sentinel` function which creates a sentinel or indicator filename by appending the target filenames to `.sentinel.` and inserts an underscore between the filenames. Functions in Make are defined quite simply and use arguments separated by commas. Similar to Unix shell scripts, arguments are labeled as `$1, $2, $3`, ... Here the only argument to the `sentinel` function is `$1` whereas `atomic` has two functions `$1` and `$2`. The `subst` function is used here to replace spaces with underscores (_). The `foreach` and `wildcard` functions are used to test for the presence of each target file and if one is not present then deletes the sentinel file using the `rm` command. It does not write any rules but simply deletes the sentinel file if necessary.

Finally, note that `r-rules-functions.mk` directive must be included before calling the `atomic` function since it must be defined prior to use. Hence, if these functions are used then `r-rules-functions.mk` is included near the top of the `Makefile` while, as usual, the file `r-rules.mk` is included near the bottom.

To use the rule it is now evident that it is simply a matter of calling the `atomic` function, as stated at the start of this section.

```
$(call atomic,plots.Rout plots0.png plots1.pdf plots2.jpg,\
        plots.R simple.RData)
        ⇥${R} ${R_FLAGS} ${R_OPTS} $<
```

The `atomic` function has two arguments `$1` and `$2`, which are the target filenames separated by spaces and the dependencies files also separated by spaces, respectively. The `sentinel` function creates the name of the sentinel file, namely

```
.sentinel.plots.Rout_plots0.png_plots1.pdf_plots2.jpg
```

In essence the `atomic` function creates a `Makefile` as before but automatically generates the following rules:

```
plots.Rout plots0.png  plots1.pdf  plots2.jpg:\
    .sentinel.plots.Rout_plots0.png_plots1.pdf_plots2.jpg ; @:
.sentinel.plots.Rout_plots0.png_plots1.pdf_plots2.jpg: \
plots.R simple.RData
        ⇥${R} ${R_FLAGS} ${R_OPTS} $<
        ⇥touch plots.Rout plots0.png plots1.pdf plots2.jpg
```

The line following the call to the `atomic` function contains the R BATCH call and the `$<` automatic variable to specify the first dependency as specified in the `Makefile`. The final line is produced with `; touch $$@` which substitutes the target variable `$@` after the `touch` command to set the times of the target files.

```
read.Rout: read.R simple.csv ; R CMD BATCH --vanilla read.R
```

Listing 2: `Makefile` line to produce `read.Rout` from `read.R` by defining an explicit rule where filenames, programs and options are hardwired. Since there is only one command it may be clearer to use a semicolon than a separate line starting with a TAB character.

Make initially creates a sentinel file if the target files are either not present or older than the sentinel file and R is run to reproduce all targets. Otherwise the sentinel file is newer and so nothing is made as shown below.

```
$ make

touch .sentinel.plots.Rout_plots0.png_plots1.pdf_plots2.jpg
R CMD BATCH --vanilla plots.R

$ make

make: Nothing to be done for 'all'.
```

In Linux shells, such as Bash, the sentinel file is hidden because its name starts with a dot (.) but once made it is not deleted by Make and is used to keep track of multiple target files.

A complete example with `Makefile` and `plots.R` is available in the `multiple_targets` subdirectory at https://github.com/petebaker/r-makefile-definitions.

5.7. Spaces in filenames

Unless you absolutely must use spaces in filenames then it is definitely best to avoid them. While you can use quotes like in `"silly file name.doc"`, it is generally safer to use a backslash to quote spaces as in `silly\ file\ name.doc`. This works in general and may work with the `wildcard` function but it can still be problematic for some automatic variables and especially with filenames longer than the standard 8.3 in Windows (Graham-Cumming 2015).

5.8. Avoiding a tab for single line pattern rule definitions

Lastly, if we are writing rules with only one command to execute then we can remove the need for typing a TAB character ↹ by using a semicolon (;). If we have more than one line then only the first line can be specified using a semicolon. Using a semicolon in GNU Make is the same as using a semicolon to start a new line in the Bash shell or a new command in R except that it removes the need to insert a TAB character ↹. Of course, this is unnecessary if we are only relying on pre-existing pattern rules but useful if writing a specific rule or writing our own pattern rules.

5.9. Forcing Make to rerun everything

Currently, there is no easy way to let Make know if targets need to be updated when statistical software, system libraries or particular packages have been updated. This can be very complex since many R packages have dependencies (other packages) which in turn have dependencies. Indeed, even if we could determine all possible package dependencies using a R package like **miniCRAN** (de Vries 2019) then changes to system libraries or complex data set ups may still make it virtually impossible to determine which targets in a `Makefile` are up to date.

In contrast, dependencies may be automatically generated for large C or C++ projects using tools like makedepend or Automake. Essentially, these tools parse the `.c`, `.cpp`, `.c++` and other source files to obtain a list of header files to use as dependencies in the `Makefile`. However, in R for instance, this is not easy for two main reasons.

Firstly, while we can hardwire and test versions of R or packages easily using say **testthat** (Wickham 2011) when R is run, we would also have to test that all installed package dependency versions and their dependencies and so on had also not changed. It is relatively straightforward to produce warnings and stop program execution. It is also straightforward to obtain version numbers of R and packages and so possible to store these using standard commands like `utils::sessionInfo`. An alternative is to employ **packrat** (Ushey, McPherson, Cheng, Atkins, and Allaire 2018) to ensure packages are at particular versions.

Secondly, obtaining changes in names of all dependency files is much more difficult. For instance, in R, filenames for included files containing R syntax or reading in data can be constructed in all sorts of ways. These include reading them from text files or operating system calls, using concatenation of sequences of numbers or characters to construct filenames or user written functions. For all practical purposes, we would need to run all relevant parts of the `.R` source files and reconstruct any filenames mentioned to include as dependencies in our `Makefile`s. It is quite clear that this process could easily miss some dependencies no matter how cleverly it was programmed.

Instead, a much simpler and more robust solution is to force Make to rerun everything by removing target and intermediate files. We can achieve this by simply running

```
make clean
```

to remove all targets and most intermediate files then run

```
make
```

to rerun everything.

6. Complex projects

As data analysis projects become more complex, to manage our workflow, we often place various aspects of the project in subdirectories. For instance, we may place various steps like reading and merging data; cleaning data; analyzing and plotting data; and reports in subdirectories. The bottom directory contains a master `Makefile` and each subdirectory contains a `Makefile` for each step. Ideally, in addition to making the whole project from the root directory, we would like to be able to change to any subdirectory and run `make` just for that particular step of the process.

```
complex_demo/myRproject
├── admin
├── analysis
│   └── Makefile
├── data
│   ├── codebook
│   ├── derived
│   └── original
├── doc
│   ├── admin
│   ├── original
│   └── references
├── lib
├── readCleanData
│   └── Makefile
├── reports
│   └── Makefile
└── Makefile
```

Listing 3: Typical directory structure for data analysis project using R. Original and derived data files are stored in `data`, R syntax files for reading, cleaning and merging data are stored in `readCleanData`, R syntax files for analyzing data are found in `analysis` and **rmarkdown** and R `Sweave` files are in `reports`.

This can be problematic if the `Makefile` in a particular subdirectory is self-contained, in that Make will not know about other steps in the process and so Make will need to be run in the parent or root directory of the project.

When Make is run from the root directory which then calls Make in each subdirectory, this is said to be a *recursive make*. While appealingly simple, there can be problems (Miller 1998). For instance, this may be inefficient because all changes will need be remade, not just the relevant dependencies of the current target file. Other problems can arise if parallel Make is employed.

Instead, it may be better to use a *non-recursive make* strategy. This strategy is implemented by using `include` statements and running `make` in the root directory rather than allowing Make to run separate Make subprocesses. Graham-Cumming (2015) provides a more comprehensive method of a *non-recursive make* to allow Make to be run within subdirectories and so work more like a *recursive make* but without the inherent problems.

A standard data analysis project directory structure is outlined in Section 6.1, recursive make is outlined Section 6.2 and a non-recursive alternative is explored in Section 6.3. Any functions to implement these approaches are provided in `r-rules-functions.mk` which is included at the top of the `Makefile` to ensure that functions are defined prior to use.

6.1. Multiple directories

Complex data analysis projects often have multiple subdirectories, for instance one for stor-

ing original, codebook and derived data (`data`), reading data (`readCleanData`) and others for statistical analysis (`analysis`) and reporting (`reports`). Each directory would have a `Makefile` which is self contained but may also refer to other files in relevant directories. A typical set up (with `Makefile`s) is shown in Listing 3.

Our aim is to be able to work in the bottom level directory or a particular subdirectory and employ `make` wherever we are working.

In essence, when employing a recursive `make` strategy, the top-level `Makefile` runs the `Makefile`s in subdirectories as `.PHONY` targets. This means that in each directory, the `Makefile` points to dependencies in other (sub)directories. However, this can create problems as noted above.

An alternative is to use makepp (Pfeiffer and Holt 2013). makepp implements much of GNU Make in Perl but there are minor differences. Apart from appearing to greatly improving recursive `make` for simple rules it sometimes requires minor rewriting of `Makefile`s. Often, this may be as simple as replacing any ∼'s with `${HOME}`. However, in terms of the rules presented here in `r-rules.mk` and `r-rules-functions.mk`, a major difference is the way that makepp handles strings which means that many of the rules written here would need to be modified to be parsed by Perl rather than Make and so two versions would be needed.

Rather than writing two sets of rules, only GNU Make is employed and is briefly outlined for recursive `make` in Section 6.2.

The alternative non-recursive `make` strategy is outlined in Section 6.3. It is equally as effective as the recursive `make` strategy and has the same familiar make-anywhere functionality. However, it does not suffer possible parallel compilation or other potential Make problems while, initially at least, it may appear to be slightly more complicated.

6.2. Recursive Make

An overall `Makefile` in the base directory is employed which can be used to (re)run all data cleaning, statistical analysis and reporting. While working on one particular aspect of the work, such as reporting, we would work in the appropriate subdirectory which also has a `Makefile` pertaining to just that part of the workflow.

When running complex builds we can run `make` with the `-j` or jobs option to specify the number of jobs. This gives the maximum number of jobs (sub-makes) that Make will run in parallel. However, problems can arise when sequential make is required but a particular step is too slow. This is particularly apparent when working with multiple directories.

The traditional way to allow parallel `make` and a number of subdirectories is to use the following construct

```
SUBDIRS := readCleanData analysis reports

.PHONY: all
all:
	for d in ${SUBDIRS};          \
	do                            \
	    ${MAKE} -directory=$$d;   \
	done
```

However, if a sub-make fails then Make will look like it succeeded.

One solution is to replace the `for` loop with a single rule as in the following data analysis project consisting of separate directories for reading, merging and cleaning data using the directories as targets where the `SUBDIRS` is defined as before but the target are specified with.

```
.PHONY: all ${SUBDIRS}
all: ${SUBDIRS}

${SUBDIRS}:
        ${MAKE} --directory=$@
```

The directories do not actually get built and so can be considered as a phony target. Therefore, each directory can run while the others are running, and parallelism is maximized. It is even possible to have dependencies between directories causing some sub-makes to run before others. Directory dependencies can be handy when it is important that one sub-make runs before another such as reading in data before analyzing it and analyzing it before reporting on it.

Often, we have found this approach to be perfectly adequate in that `Makefile`s are easy to write and easy to use. However, if the project becomes quite complicated then a non-recursive strategy will be safer. An example set of `Makefile`s for recursive `make`, along with appropriate R and **rmarkdown** files is available in the directory `complex-demo/recursive/` at https://github.com/petebaker/r-makefile-definitions.

6.3. Non-recursive Make

As outlined in Sections 6.1 and 6.2, once a project becomes quite complex and involves multiple working directories, each with its own `Makefile`, then it is tempting for the master `Makefile` to contain simple calls to `make` using the `Makefile` in each subdirectory. Conceptually, this seems straightforward but in practice there are potential pitfalls as outlined in Miller (1998) and elsewhere in this article. To avoid such problems, these projects can also be managed without employing recursive `make` to call self-contained `Makefile`s which in turn call other self-contained `Makefile`s. Instead, we employ `include` statements to insert secondary definitions carefully constructed to work whichever `Makefile` includes them.

In essence, we usually create `Makefile`s that mostly refer to files in the current directory and therefore are easy to write. The extra complication in non-recursive `make` is making sure that the relevant files are appropriately referenced no matter whether we are working in the base directory or subdirectories. Instead of completely defining the dependency and target files in each `Makefile`, we can define most of the setup about target and dependency files in common files (here `root.mk` which is in the root or base directory and a separate `module.mk` in each working subdirectory).

In the data analysis projects described here, we only employ one level of subdirectories where we do the actual work or cleaning data, analyzing data and reporting (see Listing 3). Extension of the method developed here to extra levels of subdirectories is straightforward. Mecklenburg (2004) and Graham-Cumming (2015) provide more complex examples and Graham-Cumming provides functions to find the base or root directory and also find and write rules to include header files, some of which is more complex than is needed here.

All Makefiles, root.mk and the module.mk files are available as part of the working example provided at https://github.com/petebaker/r-makefile-definitions in directory complex-demo/nonrecursive/.

Relevant target and dependency files are set up by including root.mk as well as a separate module.mk in the Makefile in the base directory and in each working subdirectory. Note that some simple functions are used by including r-rules-functions.mk at the top of Makefile and standard r-rules.mk at the bottom.

Note that we could hardwire in the root or base directory of the project into each Makefile. However, to make this as easy and error free as possible it is best to avoid this approach and instead use the functions _find and _walk (Graham-Cumming 2015) to automatically find and set the root directory in a variable for later use. It does this by searching for the file root.mk contained in the base directory. This also has the advantage of making the whole project portable when transferring it to another computer because, with virtually no overhead, the root directory is automatically found and defined each time make is run.

Non-recursive Make: Makefiles

Each Makefile firstly includes r-rules-functions.mk to firstly define the _find and _walk functions then sets _ROOT with the following two lines

```
include ~/lib/r-rules-functions.mk
_ROOT := $(patsubst %/root.mk,%,$(call _find,$(CURDIR),root.mk))
```

Next, all target and dependency files are defined relative to the root directory. This is achieved by setting the variable RELATIVE. In the base Makefile we use

```
RELATIV = ./
```

whereas in all subdirectory Makefiles we use

```
RELATIV = ../
```

Once the root directory is found and _ROOT and RELATIVE variables defined we can then include the contents of root.mk in any Makefile with

```
include $(_ROOT)/root.mk
```

which is independent of which Makefile is invoked and hence which (sub)directory we are currently working in.

We now describe the details of root.mk.

Non-recursive Make: root.mk

Firstly, the following variables are set both to define the subdirectories and also the targets to be made. The subdirectories are set with

```
READ_SUB=readMergeData
ANALYSIS_SUB=analysis
REPORTS_SUB=reports
```

Secondly, the relative paths of directories are fully defined. This is where the work is done. These paths are defined using the `RELATIVE` variable for directories where syntax for reading, analysis and reports are stored as

```
READ=${RELATIV}${READ_SUB}
ANALYSIS=${RELATIV}${ANALYSIS_SUB}
REPORTS=${RELATIV}${REPORTS_SUB}
```

Thirdly, the relevant data directories are set up with

```
DATA=${RELATIV}data
DATA_ORIG=${DATA}/original
DATA_DERIV=${DATA}/derived
```

which sets up the overall data directory, subdirectory containing original data and subdirectory for saving any derived data, respectively.

Finally, in `root.mk` we also set the default output extensions for running R and **rmarkdown** syntax with

```
R_OUT_EXT = pdf
RMD_OUT_EXT = pdf
```

as described in Section 4.4.

Non-recursive Make: Root `Makefile`

Now that the root directory and subdirectories are fully defined by including `root.mk`, the main `Makefile` then processes each subdirectory in turn. Firstly, the subdirectories are defined with

```
SUBDIRS=$(READ_SUB) $(ANALYSIS_SUB) $(REPORTS_SUB)
```

Next the phony target `all:` is defined with

```
.PHONY: all
all: all_$(READ_SUB) all_$(ANALYSIS_SUB) all_$(REPORTS_SUB)
```

Note that each target is defined in the appropriate `module.mk`. For instance, `all_analysis` is defined in `analysis/module.mk`. All dependencies are defined in the `module.mk` files including various files in other directories.

Next, all `module.mk` files are included with

```
include $(addsuffix /module.mk,$(SUBDIRS))
```

Finally, the root directory `Makefile` contains other rules such as for removing intermediate files and inclusion of the `r-rules.mk`.

Non-recursive Make: `module.mk`

Each subdirectory has a `module.mk`. Since these are called from either the root directory or the current directory then appropriate variables are set in `root.mk`. Therefore, variables pointing to the root directory and relative paths are set elsewhere.

Firstly, the phony target `all` is set which is then employed by either the `Makefile` in the current directory or the base directory. For instance in the `analysis` subdirectory, where the definition of `all_analysis` is all on one line as

```
.PHONY: all_analysis
all_analysis: ${ANALYSIS}/summary_simple_csv.${R_OUT_EXT} \
   ${ANALYSIS}/analyse_simple_csv.${R_OUT_EXT}
```

The two output or target files also have their relative path specified with the `{ANALYSIS}` variable. Similarly, for the previous steps, which require reading and cleaning the data in `../readMergeData`, the variable `READ` is set. For example, to summarize data given that it is the latest cleaned version of the data, or rerun to clean it if it is not then the rule is defined all on one line as

```
${ANALYSIS}/summary_simple_csv.${R_OUT_EXT}: \
   ${ANALYSIS}/${@:.${R_OUT_EXT}=.R} \
   ${READ}/clean_simple_csv.${R_OUT_EXT}
```

Non-recursive Make: Analysis `Makefile`

The first few lines are exactly the same as in the base directory `Makefile` in that they set the `_ROOT` and `RELATIV` variables and then include the file `root.mk` in the root directory.

Firstly, a simple phony target `all` is set up with

```
.PHONY: all
all: all_analysis
```

noting that `all_analysis` is the first and phony target defined in `module.mk`.

The next two lines simply include the commands in `module.mk` and the equivalent file in the `../readMergeData` which then defines any dependency files.

```
include module.mk
include ../readMergeData/module.mk
```

These commands are then followed by including `r-rules.mk` and cleaning rules etc.

Note that all files are defined relative to the root or base directory and so include the current subdirectory name as well. Therefore, a command like `make summary_simple_csv.pdf` will not work because the target is actually defined as `../analysis/summary_simple_csv.pdf`. For ease of exposition, assume that that `R_OUT_EXT` is set to `Rout` then running this `make` command gives the output.

```
$ make -n ../analysis/analyse_simple_csv.Rout

R CMD BATCH --vanilla ../readMergeData/read_simple_csv.R
R CMD BATCH --vanilla ../readMergeData/clean_simple_csv.R
R CMD BATCH --vanilla ../analysis/analyse_simple_csv.R
```

It is clear that the appropriate dependencies are made if required. For further details and definitions in other subdirectories, please see the rules defined in the `complex_demo/nonrecursive` example.

6.4. Projects requiring different rules for different targets

As a final example of a complex project, we may be working on projects where a one-size-fits-all approach will not work for all target files. For instance, we may have a mixture of `.Rnw` files in both `Sweave` and **knitr** formats from different collaborators.

While we could construct a `Makefile` to call `make` separately for different `.Rnw` target files using `SWEAVE_ENGINE` set to either `Sweave` or blank, better approaches are available. Following Mecklenburg (2004), we could employ a combination of target specific variables (see Section 5.2) and user defined hooking functions. In essence, for this approach, we can create user defined functions and specify which function to use with each `.Rnw` file. Relevant functions and rules are:

```
define build-sweave
   $(call build-sweave-hook,$@)
endef
```

and now define function hooks `RSWEAVEPDF` and `RKNITRPDF` for `Sweave` and **knitr** formats, respectively

```
define RSWEAVEPDF
   ${RSWEAVE} --pdf $< ${RSWEAVE_FLAGS}
endef
define RKNITRPDF
   ${RSCRIPT} ${RSCRIPT_OPTS} -e "library(\"knitr\");knit2pdf('$<')"
endef
```

where variables `RSCRIPT`, `RSCRIPT_OPTS`, `RSWEAVE` and `RSWEAVE_FLAGS` are defined in Sections 4.3 and 5.1.

Using target specific variables and hooking functions the targets are defined as

```
## RSWEAVE_RNW target(s) for .pdf from .Rnw using Sweave
$(RSWEAVE_PDF): build-sweave-hook = $(RSWEAVEPDF) $1
$(RSWEAVE_PDF):
  $(call build-sweave,$^)

## RKNITR_PDF target(s) for .pdf from .Rnw using knitr
$(RKNITR_PDF): build-sweave-hook = $(RKNITRPDF) $1
$(RKNITR_PDF):
  $(call build-sweave,$^)
```

Finally, we define the PDF target files as dependencies to either `RSWEAVE_PDF` or `RKNITR_PDF` and also define the dependency files to the PDF target files as usual. These rules are provided in `r-rules.mk` and help can be obtained with

```
make help-both-sweave-knitr
```

6.5. Rolling your own rules

Naturally, there will always be situations where pattern rules are not available. Using the examples above and modifying existing rules should provide a guide for producing new pattern rules for such situations. Other situations may not require writing pattern rules but may employ phony targets to aid automation of your workflow.

For example, if you use GNU Make to manage the production of web pages and reports from start to finish then you make also wish to include an extra step in a `Makefile` to transfer the final HTML pages and/or PDF files to a server. While there are various ways to do this, one popular method is to employ `rsync`. A `.PHONY` target could be used in the knowledge that it will always be made. However, `rsync` will only transfer files that have changed. For instance,

```
MYFILES = homepage.html report.pdf
RSYNC = rsync
RSYNC_FLAGS = -auvtr

.PHONY: rsync_all
rsync_all:
   ${RSYNC} ${RSYNC_FLAGS} ${MYFILES} myserver.com:.
```

would transfer the appropriate files to the server.

7. Discussion

When employing R and related software, reproducible research is often thought of in terms of a single **rmarkdown** or `Sweave` file or a syntax file where all the required steps may be reproduced given the data. In complex data analysis projects, the workflow can be much more complicated. There may be many steps in the process, each dependent on syntax and data files and also previous steps. In these situations, modularizing the process by employing separate syntax files for each step and a build system like GNU Make combined with a version control system like **git** will prove to be invaluable.

In this article, we have introduced new GNU Make pattern rules to enhance the workflow of data analysis projects when using statistical software like R, PSPP, Stata and SAS. Pattern rules are also provided to produce reports and articles in various formats via `pandoc` from **rmarkdown** files and via **knitr** or `Sweave` for R `Sweave` files. Common formats like Microsoft PowerPoint, IO Slides, Slidy, Beamer and Tufte presentations may be produced via the Make rules provided for **rmarkdown** files and finally for producing Beamer presentations, articles and handouts from R `Sweave` files. GNU Make rules are also provided and described for other commonly used data manipulation languages like Perl and Python which are sometimes employed as an integral part of the data analysis workflow. Other packages and languages may be added as outlined in the article.

With very little knowledge of GNU Make, data analysts can use these rules with minimal effort. Implementation is straightforward, in that the file `r-rules.mk` may be downloaded from GitHub and included in a `Makefile` by incorporating an `include` statement. Simple `Makefile`s may be constructed by specifying relationships between targets and their dependencies. Alternatively, `Makefile`s can be constructed using the **MakefileR** R package (Müller 2016).

Typically, as commonly used by GNU Make practitioners, customized Make rules to provide help are provided for customized rules. After including `r-rule.mk`, help may be obtained by issuing the command `make help` in a terminal.

However, if extra flexibility is required, then program parameters and options may be changed by setting associated variables for any rule provided or even creating new rules. Pattern rules provided here and associated variables are outlined in Table 3. New rules will be added to `r-rules.mk` as the need arises. As an example, rules are provided for simple data backup via `rsync`. One advantage of using Make rules for `rsync` instead of using `rsync` directly is that destinations may be hardwired into the `Makefile` so that we adopt a "set and forget" approach for each project. Other rules may also be employed to clean intermediate files but these rules are not outlined here.

It is our experience that employing the GNU Make pattern rules provided here, in conjunction with a modern version control system like **git**, considerably increases a data analyst's efficiency and aids *reproducible research*. For instance, it is not uncommon to return to a data analysis project after several months or even years have elapsed. In this situation, GNU Make can be employed not only to regenerate output automatically given changes in the data or syntax but also provide an audit trail of the exact steps that were used to produce a final report or journal article. Commonly used methods, such as point and click followed by cut and paste, often suffer from an inability to reproduce the results and have no audit trail. While it does not circumvent the need for good documentation, employing version control and the Make system described here goes a long way towards fixing the inherent limitations of such an ad-hoc approach.

Indeed, to provide clarity of all the steps undertaken in a particular data analysis project, the final version of a report or article should be able to be reproduced by providing a minimal set of the final data, syntax and `Makefile`s as long as the same versions of the software are employed. Differing versions of the software can also be problematic but there are efforts being made to overcome this issue, especially with respect to R packages since some of these may undergo significant changes in a short time (see Ushey *et al.* 2018).

A large data analysis project may employ a team of researchers or data analysts working in conjunction with subject matter researchers and data managers. Following on from the ideas of Long (2009), they may also *publish* intermediate data sets and reports to freeze these at a certain point in time ensure all team members are working on the same cleaned data set or referring to the same report. An alternative is to employ a central **git** repository and in this case GNU Make will also prove useful. While the repository could be set up to only contain the minimal file set required to regenerate output, it can prove advantageous to also put some reports under version control. Team members can then check out a particular version if required. Using version control also ensures that older versions are available whether they are *published* or not.

While it is preferable to have collaborators or reviewers check out a minimal file set from a **git** repository, there may also be the need to email it. Usually when distributing code in such a way, it is common to specify a `cleanall` rule or similar to remove any intermediate files and allow a minimal set of files to be sent.

Given the vast array of software employed for data analysis and reporting, it is clear that new rules will be required for complex data analysis projects. For instance, we may need to produce different formats for vignettes in R packages using alternate vignette engines for

formats like HTML or reStructured Text files. Once an appropriate command can be run from the command line to produce an example target file given example dependency files then it is a straightforward to write a GNU Make pattern rule as outlined above.

There are however still some limitations, even when compared to other programming languages. For instance, when C/C++ projects become very large then manually specifying dependency files may become error prone. Hence, when using Make for C or C++, we may use tools like makedepend, Automake or Make depend to automatically generate dependency lines for Make. While this topic is too advanced to discuss here in any detail, it is sufficient to note that such an approach would be difficult to emulate when using a statistical analysis package like R. This is because in C, dependencies like header (`.h`) files are hardwired into the C files, which are subsequently parsed to obtain header filenames, whereas in R and other statistical packages, data filenames are often constructed by manipulating strings and so the usual way of generating dependency lines in `Makefile`s is unavailable. Hence, it seems more practical to hardwire the filenames, employ R syntax to generate the names or to employ wildcards despite the possibilities of errors.

Other build or automation approaches to GNU Make are available but are not as widely used. Alternatives may work equally as well but it is common for programmers to revert to Make if the alternative proves problematic (Mecklenburg 2004; Graham-Cumming 2015). Indeed, some well known alternatives such as SCons (`http://scons.org/`) require programming another language like Python or are very much Make like build systems. These include makepp (Pfeiffer and Holt 2013), CMake (Hoffman and Martin 2003) and Remake (`http://bashdb.sourceforge.net/remake/`). Superficially, while these systems often have strong advocates and in time one of these or another alternative may supersede GNU Make, these methods would appear to be as complicated as standard GNU Make if not more so. Another drawback of the Make-like systems, e.g., makepp and Remake, is that they are often coded in Perl or Python and so possess slight differences which can cause incompatibilities with standard `Makefile`s. For a reasonably comprehensive discussion of alternatives see `https://bitbucket.org/scons/scons/wiki/SconsVsOtherBuildTools`.

8. Conclusion

In conclusion, pattern rules for GNU Make are provided for several statistical software packages and related computational tools to aid in the workflow of data analysis projects. These rules can aid the management, efficiency and repeatability of such projects, especially when used in conjunction with a version control system like **git**. The examples described here for a simple analysis and a more complex multi-directory project using either recursive or non-recursive `make` are also provided. Other examples are available for multiple targets and producing Beamer presentations and handouts using either **knitr** or `Sweave`. These examples will allow readers to experiment with rerunning Make while altering syntax, data and `Makefile`s in order to gain familiarity with the approach and so tailor it to their own data analysis projects.

The most recent versions of GNU Make pattern rules and all files to reproduce the examples described here are available at `https://github.com/petebaker/r-makefile-definitions`.

Acknowledgments

We would like to thank Bob Forrester, formerly of CSIRO, who alerted us to using Make and version control for managing data analysis projects in the mid 1990s. We would also like to thank participants at recent R Users Conferences for stimulating discussions about using Make for data analysis workflow and, in particular, participants at my R data analysis workflow tutorial (see also http://user2015.math.aau.dk/tutorials#baker) in 2015 and Noam Ross in 2016 for posing the question about multiple targets as discussed in Section 5.6. Finally, we would like to thank an anonymous referee for providing many detailed and useful suggestions to improve this article.

References

Allaire JJ, Xie Y, McPherson J, Luraschi J, Ushey K, Atkins A, Wickham H, Cheng J, Chang W (2020). ***rmarkdown**: Dynamic Documents for R*. R package version 2.1, URL https://CRAN.R-project.org/package=rmarkdown.

Baker P (2020). ***gnumaker**: GNU* `Makefile` *Creation and Plotting for Data Analysis Projects*. R package version 0.0.0.9007, URL https://github.com/petebaker/gnumaker.

Ball A (2017). ***beamerswitch**: Convenient Mode Selection in **Beamer** Documents. Version 1.2*. URL https://www.ctan.org/pkg/beamerswitch.

Boehmke B (2016). *Data Wrangling with R*. Springer-Verlag, Switzerland.

Bostock M (2013). "Why Use Make." Accessed: 2018-10-30, URL https://bost.ocks.org/mike/make/.

Broman K (2018). "Minimal Make." URL http://kbroman.org/minimal_make/.

Collins J (2018). "**latexmk** Version 4.61." CTAN Package **latexmk**. Accessed: 2019-01-02, URL http://www.personal.psu.edu/jcc8/latexmk/.

Dasu T, Johnson T (2003). *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, New Jersey.

de Aquino JA (2016). ***Nvim-R** Improves **Vim**'s Support to Edit R Code*. Accessed: 2018-10-30, URL https://github.com/jalvesaq/Nvim-R.

de Vries A (2019). ***miniCRAN**: Create a Mini Version of CRAN Containing Only Selected Packages*. R package version 0.2.12, URL https://CRAN.R-project.org/package=miniCRAN.

Gatto L, Breckels L, Gibb S, Smith T (2014). "`Makefile` for R Packages." Accessed: 2018-10-30, URL https://github.com/ComputationalProteomicsUnit/maker.

Gillespie C (2011). "`Makefiles` and `Sweave`." Accessed: 2018-10-30, URL https://csgillespie.wordpress.com/2011/05/12/makefiles-and-sweave/.

Graham-Cumming J (2015). *The GNU Make Book*. No Starch Press, San Francisco.

Hacker News (2013). "Why Use Make." Accessed: 2018-10-30, URL https://news.ycombinator.com/item?id=5275313.

Hoffman W, Martin K (2003). "The CMake Build Manager." Accessed: 2017-05-31, URL http://www.drdobbs.com/cpp/the-cmake-build-manager/184405251.

Howell M, De Meo M, Janke A, Cheng X, McQuaid M, Fontaine B, Koonce B, Afanasjew M, Tiller D, Smith T, Dunn A, Nagel J, Vandenberg A (2009). "**Homebrew**." URL https://github.com/Homebrew.

Hyndman RJ (2018). "Makefiles for R/LaTeX Projects." Accessed: 2018-10-30, URL https://robjhyndman.com/hyndsight/makefiles/.

Jackman S, Bryan J (2014). "Automating Data Analysis Pipelines." Accessed: 2018-10-30, URL http://stat545.com/automation00_index.html.

Jones Z (2018). "GNU Make for Reproducible Data Analysis." Accessed: 2018-10-30, URL http://zmjones.com/make.

Landau WM (2018). "The **drake** R Package: A Pipeline Toolkit for Reproducibility and High-Performance Computing." *Journal of Open Source Software*, **3**(21). doi:10.21105/joss.00550.

Leisch F (2002). "Dynamic Generation of Statistical Reports Using Literate Data Analysis." In W Härdle, B Rönz (eds.), *COMPSTAT 2002 – Proceedings in Computational Statistics*, pp. 575–580. Physica Verlag, Heidelberg.

Lindenbaum P (2014). "**makefile2graph**: Creates a Graph of Dependencies from GNU Make; Output Is a Graphiz-Dot File or a Gexf-XML File." Accessed: 2016-04-03, URL https://github.com/lindenb/makefile2graph.

Loeliger J, McCullough M (2012). *Version Control with **git**: Powerful Tools and Techniques for Collaborative Software Development*. 2nd edition. O'Reilly Media, Sebastopol.

Long JS (2009). *The Workflow of Data Analysis Using Stata*. StataCorp LP, Texas.

Mecklenburg R (2004). *Managing Projects with GNU Make*. 3rd edition. O'Reilly Me, Sebastopol.

Miller P (1998). "Recursive Make Considered Harmful." *AUUGN Journal of AUUG Inc*, **19**(1), 14–25.

Müller K (2016). ***MakefileR**: Create* Makefiles *Using R*. R package version 1.0, URL https://CRAN.R-project.org/package=MakefileR.

Olson J (2017). "Time for Makefiles to Make a Comeback." Accessed: 2018-10-30, URL https://medium.com/@jolson88/its-time-for-makefiles-to-make-a-comeback-36cbc358bb0a.

Pfeiffer D, Holt G (2013). "Makepp Home Page." Last updated: 2013-10-13, Accessed: 2017-02-17, URL http://makepp.sourceforge.net/.

R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rossini AJ, Heiberger RM, Sparapani RA, Mächler M, Hornik K (2004). "**Emacs** Speaks Statistics: A Multiplatform, Multipackage Development Environment for Statistical Analysis." *Journal of Computational and Graphical Statistics*, **13**(1), 247–261. doi:10.1198/1061860042985.

RStudio Team (2015). ***RStudio**: Integrated Development Environment for R*. **RStudio**, Inc., Boston. URL http://www.rstudio.com/.

StataCorp (2019). *Stata Statistical Software: Release 16*. StataCorp LLC, College Station. URL http://www.stata.com/.

Ushey K, McPherson J, Cheng J, Atkins A, Allaire JJ (2018). ***packrat**: A Dependency Management System for Projects and Their R Package Dependencies*. R package version 0.5.0, URL https://CRAN.R-project.org/package=packrat.

Wickham H (2011). "**testthat**: Get Started with Testing." *The R Journal*, **3**(1), 5–10. doi:10.32614/rj-2011-002.

Wickham H (2014). "Tidy Data." *Journal of Statistical Software*, **59**(10), 1–23. doi:10.18637/jss.v059.i10.

Wickham H, Danenberg P, Csárdi G, Eugster M (2020). ***roxygen2**: In-Line Documentation for R*. R package version 7.1.0, URL https://CRAN.R-project.org/package=roxygen2.

Xie Y (2015). *Dynamic Documents with R and **knitr***. 2nd edition. Chapman & Hall/CRC, Boca Raton.

Xie Y (2020). ***knitr**: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.28, URL https://CRAN.R-project.org/package=knitr.

A. Using GNU Make with IDEs and editors

RStudio (**RStudio** Team 2015), **ESS** (**Emacs** speaks statistics; Rossini *et al.* 2004) and **Vim** (de Aquino 2016) are commonly used development environments for R.

We briefly outline how to set up GNU Make in the first two IDEs.

A.1. GNU Make with RStudio

RStudio (**RStudio** Team 2015) is probably the most commonly used development for R programmers and is always under rapid development for new features. As at version 1.0.153, the build tools can be set up as follows:

1. Either create a new project or begin in an existing project.
2. Use the `Build > Configure Build Tools` menu and then choose 'Makefile' from the list of 'Project Build Tools' options as shown in Figure 2.

You may then use the 'Build' menus or shortcuts like `Ctrl-Shift-B` to build all. Other menu options like 'Clean' or 'Clean and Rebuild' may also be available depending on your setup.

A.2. GNU Make with Emacs ESS

Emacs speaks statistics (**ESS**; Rossini *et al.* 2004) is another commonly used R development environment. By default, `make -k` is available via the compile command. This can be accessed via `M-x compile`. However, we prefer to bind this to a function key by including the following lines in my `init.el` file.

```
;; press f12 to compile using default 'make -k'
(setq compilation-read-command nil)
(global-set-key [f12] 'compile)
```

Depending on the actual project, we also find this key binding useful if we wish to compile from another frame (**Emacs** window).

```
;; probably most useful is to make it Ctrl-f12 as we use f12 to compile
(global-set-key (kbd "C-<f12>") 'other-frame)
```

B. `Makefile` for simple example

A complete `Makefile` for the simple example in Section 3.2 is provided here for a simple data analysis project where file `simple.csv` is read in using `read.R`, analyzed with `linmod.R` and reports produced with `report1.Rmd` and `report2.Rmd`. To trace the dependencies, it is best to read the `Makefile` from the bottom up.

```
## File:    Makefile
## Purpose: Simple Example
```

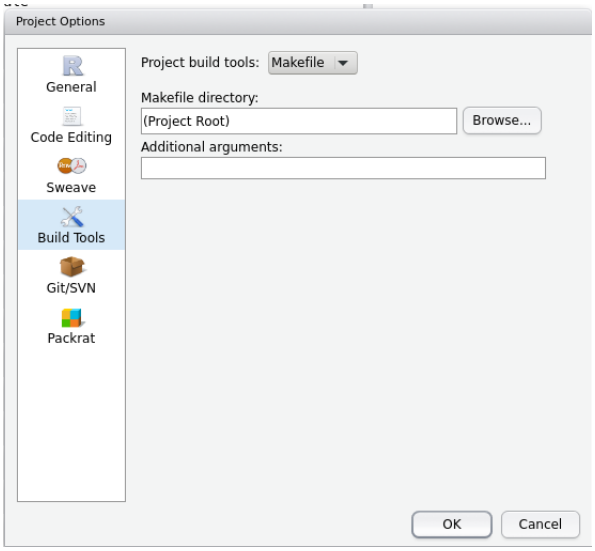

Figure 2: **RStudio** build configuration.

```
.PHONY: all
all: report1.pdf report2.docx

## reports 1 & 2 depend on results of 'linmod.Rout' & '.Rmd' file(s)
report1.pdf: report1.Rmd linmod.Rout
report2.docx: report2.Rmd linmod.Rout

## data analysis:  dependent on 'linmod.R' and 'read.Rout'
linmod.Rout: linmod.R read.Rout

## read in data: depends on 'read.R' and 'simple.csv'
read.Rout: read.R simple.csv

## include R pattern rule definitions from file in $HOME/lib
include ~/lib/r-rules.mk
```

For a dependency graph of the relationships between files see Figure 1.

C. R and related pattern rules in detail

Table 3 provides details of pattern rules to produce text, PDF, Word or HTML output from R files. Currently, rules for several other output formats are provided and new rules will be added when additional output types are added to **rmarkdown**. Variables such as `R`, `R_FLAGS`, `RSCRIPT` and `RSCRIPT_OPTS` may be redefined to override default definitions. Note that when producing PDF files, on some systems, LaTeX defaults may automatically crop figures to be blank and so we may wish to change the `fig_crop` option to be `FALSE`. This can be done by redefining `RMARKDOWN_PDF_OPTS` as outlined in Section 4.3 or Listing 4.

You can include `r-rules.mk` with an `include` statement. Since you will predominantly be

Target	Dependency	Purpose	Commands, variables & notes
R batch			

`.Rout`	`.R` or `.r`	R batch mode	`R CMD BATCH --vanilla file.R` (resulting command)
			`${R} ${R_FLAGS} ${R_OPTS} file.R`, (actual command) where
			`R = R`
			`R_FLAGS = CMD BATCH`
			`R_OPTS = --vanilla`

R Notebooks and documents			

`.html`	`.R`, `.r`, `.Rmd` or `.rmd`	HTML document from R syntax or R Markdown file	`Rscript -vanilla -e "library('rmarkdown'); render('FILE', 'html_document')"`
			where `'FILE'` is `'file.Rmd'`, `'file.rmd'`, `'file.R'` or `'file.r'`
			which can be altered using variables:
			`${RSCRIPT} ${RSCRIPT_OPTS} -e "library('rmarkdown'); \`
			`render('FILE', ${RMARKDOWN_HTML_OPTS} ${RMARKDOWN_HTML_EXTRAS})"`
			`RSCRIPT = Rscript`
			`RSCRIPT_OPTS = --vanilla`
			`RMARKDOWN_HTML_OPTS = "html_document"` and `RMARKDOWN_HTML_EXTRAS` is blank

`.docx`	`.R`, `.r`, `.Rmd` or `.rmd`	Word document from R syntax or R Markdown file	`Rscript -vanilla -e "library('rmarkdown'); render('FILE', 'word_document')"`
			which can be altered using variables:
			`${RSCRIPT} ${RSCRIPT_OPTS} -e "library('rmarkdown'); \`
			`render('FILE', ${RMARKDOWN_DOCX_OPTS} ${RMARKDOWN_DOCX_EXTRAS})"`
			where `'FILE'`, `RSCRIPT` and `RSCRIPT_OPTS` as above and
			`RMARKDOWN_DOCX_OPTS = "word_document"` and `RMARKDOWN_DOCX_EXTRAS` is blank

`.pdf`	`.R`, `.r`, `.Rmd` or `.rmd`	PDF document from R syntax or R Markdown file	`Rscript -vanilla -e "library('rmarkdown'); render('FILE', 'pdf_document')"`
			which can be altered using variables:
			`${RSCRIPT} ${RSCRIPT_OPTS} -e "library('rmarkdown'); \`
			`render('FILE', ${RMARKDOWN_PDF_OPTS} ${RMARKDOWN_PDF_EXTRAS})"`
			where `'FILE'`, `RSCRIPT` and `RSCRIPT_OPTS` as above and
			`RMARKDOWN_PDF_OPTS = "pdf_document"` and `RMARKDOWN_PDF_EXTRAS` is blank

Table 3: Pattern rules to produce `.Rout` output from an `.R` syntax file and HTML, Word or PDF output files from either a `.R` syntax file (`file.R`) or `.Rmd` R Markdown file (`file.Rmd`) provided in `r-rules.mk`. Similar rules are available for Libre Office and Rich Text Format documents and also IO Slides, Slidy, Beamer, Tufte and Microsoft PowerPoint presentation files. Note that variables may be set by defining them after including the file `r-rules.mk` in the `Makefile`. Run `make help-rmarkdown` for details.

```
include ~/lib/r-rules.mk
R = R-development
R_OPTS    = --no-environ --vanilla
RMARKDOWN_PDF_OPTS = pdf_document(fig_crop=FALSE)
```

Listing 4: Including the `r-rules.mk` file stored in directory `/lib` with an `include` statement in a `Makefile`. Note that variables like `R, R_OPTS` and `RMARKDOWN_PDF_OPTS` are redefined after the `include` statement rather than before.

using the same rules for different projects, it makes sense to keep the file in a directory you can access easily. Often this may be the directory `lib` or `Library` in your home directory or in a system wide directory such as `/usr/local/include`. The last directory is automatically searched by GNU Make and so the directory will not need to be specified in the `include` statement.

You can modify rules by redefining them, or more likely, variables after the include statement.

D. Installing GNU Make

To verify that GNU Make is installed, in a terminal window, type

```
make --version
```

If `make` is installed and in your `PATH`, a message will be returned like:

```
GNU Make 4.2.1
Built for x86_64-redhat-linux-gnu
Copyright (C) 1988-2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Linux users should already have a relatively recent version of Make installed as part of a standard set up.

Note that even on the most recent version of other operating systems, Make may be quite old. For instance, on the most recent version of macOS, Make is over 12 years old and make supplied with old versions of **Rtools** for Windows may be even older. While this is somewhat prehistoric for computer software, it should usually have little effect although some of the more modern Make constructs supplied here may not work. Hence, you may wish to upgrade to more recent versions. Both **Rtools** and XCode will also install other compilers and tools which may also prove useful for program development. Versions older than 4.0 may not work appropriately with recursive Make nor some of the rules provided. If your current version limits your approach then you may need to install a more up to date of GNU Make.

macOS users should install XCode from the App Store but this will install an old version of GNU Make. For a newer version you may wish to install GNU Make using the **Homebrew**

package manager (Howell *et al.* 2009). If you do this then you will run the newer version with `gmake` because `make` runs the older version from XCode. Note that some very old versions of GNU Make (earlier than 3.82) installed may not work for some of the rules implemented here. Recent versions will match the pattern that matches most specifically whereas very old ones may not.

Windows users who do not already have GNU Make may find it much easier just to install the **Rtools** available for Windows from a Comprehensive R Archive Network (CRAN; https://CRAN.R-project.org) mirror. However, relatively recent versions of the **Rtools** may contain fairly old versions of GNU Make. For instance, in recent versions of the **Rtools**, GNU Make may be a decade old or in some cases much older but it will probably work just fine in most circumstances. Fortunately, the current **Rtools** (version 3.5) provides the very up to date GNU Make 4.2.1. Windows users who are developing software or who wish to use Unix tools may already have installed a version of Make with gnuwin32, MSYS, MSYS2 or cygwin. The latter two have up to date versions of Make whereas the first two have quite old versions. Finally, Windows 10 users may also install recent versions of Bash and GNU Make by enabling developer mode and installing the Windows Subsystem for Linux.

Affiliation:

Peter Baker
School of Public Health
University of Queensland
Herston 4006 Australia
E-mail: p.baker1@uq.edu.au
URL: http://petebaker.id.au/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
June 2020, Volume 94, Code Snippet 1 *Submitted:* 2017-08-10
doi:10.18637/jss.v094.c01 *Accepted:* 2019-04-15