INTEGRATED BRAIN IMAGING CENTER

Using GNU MAKE for Neuroimaging Workflow

University of Washington

Using GNU make for Neuroimaging Workflow

Tara M. Madhyastha (madhyt@uw.edu) Zoé Mestre (zlm@uw.edu)
Trevor K. McAllister-Day (tkmday@uw.edu)
Natalie Koh (natkoh@uw.edu)

July 24, 2015

Contents

List of Figures

Part I

Manual

Chapter 1

A Conceptual Introduction to `make` for Neuroimaging Workflow

This guide is intended for scientists working with neuroimaging data (especially a lot of it) who would like to spend less time on workflow and more time on science. The principles of automation and parallelization, taken from computer science, can easily be applied to neuroimaging to make certain parts of the analysis go quickly so that the computer can do what it's best at. This kind of automation supports reproducible science (or at the very least, allows you to rerun your analysis extremely quickly with a variant of the processing stream, or on a new dataset).

Over the last few years we have developed neuroimaging workflows using `make`, a program from the 1970s that was originally used to describe how to compile and link complicated software packages. It is an amazing program that is still in use today. It is fairly easily repurposed to describe neuroimaging workflows, where you specify how you "make" one file, which can depend on several other files, using some set of commands. Once you have expressed these "rules" or "recipes" in a file (a Makefile), you actually have a fully parallelizable program, which allows you to run the Makefile over as many cores as you have (or on a Sun Grid Engine). This is not a new idea: FreeSurfer incorporates `make` into their pipeline "behind the scenes."

Modern tools for neuroimaging workflow (nipype, LONIPipeline) incorporate data provenance (e.g., information about how and when the results were generated), wrappers for arguments so that you do not need to worry about all the different calling conventions of programs, and generation of flow graphs (that may be difficult to achieve with make). However, `make` allows expression of neuroimaging workflow with only the programming concepts of parallelism and variables. T. M. has taught several undergraduates to use it to develop, debug, and execute pipelines. It is simple enough that the basic concepts can be understood by non-programmers. This results in more time teaching and doing neuroimaging than teaching (and doing)

programming.

Conventions Used in This Manual

By convention, actual commands or filenames will be typeset in Inconsolata (for example, `ls`, `make`, FSL's `bet`, and `flirt`.) Makefile examples will also be typeset in Inconsolata.

Commands to be executed in a shell (we assume `bash` throughout this manual) are written on a line beginning with a dollar sign ($) and displayed between two horizontal lines, as follows:

```
$ echo Hello World!
```

Examples of Makefiles are shown in an outlined box with no special prompt character:

```
This is a make command.
```

An example of a `make` command. Additionally, recipes are easily identified by their opening syntax, where the recipe target is usually rendered in blue (see Figure 1.1 or Figure 1.2).

Throughout this manual we assume a UNIX style environment (e.g., Linux or MacOS). All examples have been tested on Ubuntu Linux. Please let us know of problems or typos, as this manual is a work in progress.

Quick Start Example

A makefile is a text file created in your favorite text editor (e.g., `txtedit`, `emacs`, `vi`, `gedit` – *not* Office or OpenOffice). By convention, it is typically called "Makefile" or "foo.mk" (i.e., it has a .mk extension). It is like a little program that is run by the program `make`, in the same way that a shell script is program run by the program `/bin/bash`.

A makefile contains commands that describe how to create files from other files (for example, a skull-stripped image is created from the raw T1 image, or a T1 to standard space registration matrix is created from a standard space skull-stripped brain and a T1 skull-stripped brain).

These commands take the form of "rules" that look like this:

This may be best understood by example. Figure 1.2 is a simple makefile that executes FSL's `bet` command to skull-strip a T1 nifti.

! Recipes in make need to begin with a TAB character. Not eight spaces, not nine, but exactly one TAB character.

```
target: dependencies

        Shell commands to create target from dependencies (a recipe),
        beginning with a [TAB] character.
```

Figure 1.1: Basic `make` syntax.

```
s001_T1_skstrip.nii.gz: s001_T1.nii.gz
        bet s001_T1.nii.gz s001_T1_skstrip.nii.gz -R
```

Figure 1.2: A very basic `make` recipe.

Here, the target file is `s001_skstrip.nii.gz`, which "depends" on `s001_T1.nii.gz`. More specifically, to "depend on" something means that *b)* it cannot be created unless the dependency exits, and *b)* it must be recreated if the dependency exists, but is newer than the target.

The "recipe" is the `bet` command that creates the skull-stripped image from the T1 image. This executes in a shell, just as if you were typing it in the terminal.

If this rule is saved inside a file called `Makefile` in the same directory as `s001_T1.nii.gz`, then you can create the target as follows:

```
$ make s001_T1_skstrip.nii.gz
```

Alternatively, in this instance, you could call:

```
$ make
```

If you specify a target as an argument to make, it will create that target. If you do not, `make` will build the first target in the file. In this case, they are the same.

A More Realistic Example

In the example above, we specified exactly what file to create and what file it depended upon. However, in neuroimaging analyses we normally have many subjects and want to do the same things for all of them. It would be a royal pain to type in every rule to create every file explicitly, although it would work. Instead we can use the concepts of variables and patterns to help us. A variable is a name that is used to store other things. You can set a variable to something and then refer to it later. A pattern is a substring of text that can appear in different names. Suppose we have a

directory that contains the T1 images from 100 subjects with identifiers s001 through s100. The T1 images are named `s001_T1.nii.gz`, `s002_T1.nii.gz` and so on. This makefile will allow you to skull strip all of them (and on as many processors as you can get a hold of – see chapter 3 – but I'm getting ahead of myself here).

`%`	matches a pattern.
`$`	is the target.
`$<`	is the first dependency.
`$(VAR)`	is a `make` variable.

Figure 1.3: Automatic `make` variables

The first two lines in Figure 1.4 set variables for Make. Yes, the syntax is icky but it is well explained in the GNU `make` manual. We will summarize here. The first variable is assigned to the result of a "wildcard" operation that expands to all files with the pattern `s????_T1.nii.gz`. If you are not familiar with wildcards, if you do a directory listing of that same pattern, it will match all files that begin with an "s," are followed by exactly three characters, and then "_T1.nii.gz." In other words, the `T1files` variable is set to be all T1 files belonging to those subjects in the current directory.

```
T1files=$(wildcard s???_T1.nii.gz)
T1skullstrip=$(T1files:%_T1.nii.gz=%_T1_skstrip.nii.gz)
all: $(T1skullstrip)

%_T1_skstrip.nii.gz: %_T1.nii.gz
        bet $< $@ -R
```

Figure 1.4: A more realistic example

The second variable is set using pattern substitution on the list of T1 files, substituting the file ending (`_T1.nii.gz`) for a new file ending (`_T1_skstrip.nii.gz`) to create the list of target files. Note that the percent sign (`%`) matches the subject identifier. It is necessary to use the percent sign to match only the subject identifier (and not the subject identifier plus the following underscore, or some other extension) when matching parts of file names in this way.

We have now introduced a new type of rule (`make all`) which has no recipe. Make will look for a file called `all` and this file will not exist. It will then try to create all the things that all depends on (and those files, the skull stripped images, don't exist either). So it will then take names of each skull stripped file, one by one, and

look for a rule to make it. When it has done that, it will execute the (nonexistent) recipe to make target `all` and be finished. Because the file `all` still does not exist (by intention), trying to make the target will always result in trying to make all the skull stripped files.

This brings us to the final rule. The percent sign in the target matches the same text in the dependency. This target matches the name of each of the skull stripped files desired by target all. So one by one, they will be created. In the recipe for the final rule, we have used a shorthand of `make` to refer to both the dependency that triggered the rule (`$<`) and the target (`$@`). We do this because since the rule is generic, we do not know exactly what target we are making. However, we could also write out the variables (as seen in Figure 1.5).

```
%_T1_skstrip.nii.gz: %_T1.nii.gz
        bet $*_T1.nii.gz $*_T1_skstrip.nii.gz -R
```

Figure 1.5: An expansion of Figure 1.4

In this version of the rule, we use the notation `$*` to refer to what was matched in the target/dependency by `%`. If the previous version looked to you like a sneeze, this may be more readable.

Now you can run this makefile in many ways. As before, to create a specific file, you can type:

```
$ make s001_T1_skstrip.nii.gz
```

To do everything, you can type: `make all` or just `make`.

Suppose you do this and later find that the T1 for subject 036 was not properly cropped and reoriented. You regenerate this T1 image from the dicoms and put it in this directory. Now if you type make again, it will regenerate the file `s036_T1_skstrip.nii.gz`, because the skull strip is now newer than the original T1 image. Suppose you acquire four more subjects. If you dump their T1 images into this directory following the same naming convention and type `make`, off you go.

This probably seems like an enormous amount of effort to go to for some skull stripping. However, the benefit becomes clearer as the complexity of the pipelines increases.

What is the Difference between a Makefile and a Shell Script?

This is a really good question. A makefile is basically a specialized program that is good at expressing how X depends upon Y and how to create it. The recipes in a makefile are indeed little shell scripts, adding to the confusion. By making these dependencies explicit, you enable the computer to execute as many of the recipes as it can at once, finishing the work as quickly as possible. It allows the computer to pick right back up if there is a crash or error that causes the job to die somewhere in the middle. It allows you, the scientist, to decide that you want to change some step three-quarters of the way down and redo everything that depends upon that step.

Shell scripts are more general programs that can do all sorts of things, but inherently do them one at a time. For example, the shell script that is the equivalent to Figure 1.4 could be written something like Figure 1.6.

```
#!/bin/bash
for i in s???_T1.nii.gz
do
name=`basename $i .nii.gz`
bet $i $name_skstrip.nii.gz -R
done
```

Figure 1.6: A makefile expressed in `bash`

There is nothing in this script to explicitly tell the computer that each individual T1 image can be processed independently of the others; the order does not matter. So if you are on a multicore computer that can execute many processes at once, you could not exploit this parallelism with this shell script. Furthermore, you can see that to add a few subjects or redo a few subjects, you will need to edit the script to avoid rerunning everyone.

If you have ever found yourself writing a shell script to do a large batch job, and then commenting out some of the subject identifiers to redo the few that need redoing, then commenting out some other parts and adding new lines to the program, and so forth, you are dealing with the problems that `make` can help with.

Chapter 2

Running `make` in Context

File Naming Conventions

File naming conventions are also critical for scripting in general, and for Makefiles in particular. `make` is specifically designed to turn files with one extension (the extension is the last several characters of the filename) into files with another extension. For example, in Figure 1.2 we turned a file with an extension "T1.nii.gz" into a file with the extension "T1_skstrip.nii.gz."

So we can use naming conventions within a project, and across projects consistently to reuse rules that we write for common operations. In fact, `make` has many built-in rules for compiling programs that are absolutely useless to us. But we can write our own rules.

Thus, it is important to decide upon naming conventions.

! "**Extension**" often refers to the file suffix, e.g., ".csv," but there is no reason it must be limited to a fixed number of characters after a period.

Subject Identifiers

The first element of naming conventions is the subject identifier. This is usually the first part of any file name that we might end up using outside of its directory. For example, when processing resting state data, the final cleaned data begins with the subject identifier, allowing us to dump all those files into a new directory for subsequent analysis with another program, without having to rename them or risk overwriting other subject files. Some features (by no means exhaustive) that may be useful for subject identifiers are:

1. **They should contain some indicator of which project the subjects belong to.**
   This is particularly important if you work on multiple projects, or if you may be pooling data from multiple studies. Typically choose a multi-letter code that is the prefix to the subject ID.

2. **If applicable, it should contain some indicator of treatment group that the subject belongs to.**
   It is helpful to indicate in the subject ID (usually with another letter or digit code) whether the subject is part of a treatment group or a control.
3. **They should be short.**
   Short names are easier to type and to remember, and make it easier to work with lists of directories that are named according to the subject id.
4. **They should be the same length.**
   This is not necessary but helpful for pattern matching using wildcards in a variety of contexts within `make` and without.
5. **They should sort reasonably in UNIX.**
   Many utilities ultimately require a list of filenames, which is easy to generate using wildcards and the treatment group code if the subject ID sort order is reasonable. The classic problem is to name subjects S1 ... S10, S11 – the subjects will not be in numeric order without zero-padding to the subject numbers.
6. **They should be easy to type.**
   `bash` has many tricks to help you avoid typing, but when you have to type, the farther you have to move the longer it takes.
7. **They should be easy to remember for short periods of time.**
   My attention span is very short when doing repetitive tasks; subject IDs that are nine-digit random numbers are harder to remember than subject IDs that are four letter random sequences.
8. **They should be easy to divide into groups using patterns.**
   My favorite set of subject IDs were randomly generated four letter sequences, making it easy to divide the subjects into groups of just about any size using the first letters and wildcards. This makes it very easy to test something on a subset of subjects.
9. **They should be consistently used in all data sets within the project.**
   If the subject identifier is "RC4101" in a directory, it should not be "RC4-101" in the corresponding Redcap database and "101" in the neuropsychological data – it is easier if everyone can decide upon one form of name.

Filenames

File naming conventions begin with converted raw data (nifti files derived from the original DICOMs, for example), and continue on for each level of processing. We find it helpful to give these files specific names and then to perform subject-specific

processing within the subject/session directory. The filenames for a recent active project are shown in Figure 2.1, where the subject ID is "RPI001."

Filename	Description
RPI001_T1_brain.nii.gz	Skull stripped T1 MPRAGE image
RPI001_T1.nii.gz	T1 MPRAGE image
RPI001_read.nii.gz	Reading task scan
RPI001_read_rest.nii.gz	Resting state scan for reading session
RPI001_read_fMRIb0_phase.nii.gz	RPI001_read_fMRIb0_mag.nii.gz
RPI001_read_fMRIb0_mag_brain.nii.gz	Reading task B0map phase image, magnitude image and brain overlaying over magnitude image
RPI001_write.nii.gz	Writing task scan
RPI001_write_rest.nii.gz	Resting state scan for writing session
RPI001_write_fMRIb0_phase.nii.gz	RPI001_write_fMRIb0_mag.nii.gz
RPI001_write_fMRIb0_mag_brain.nii.gz	Writing task B0map phase image, magnitude image and skull stripped magnitude image
RPI001_DTI.nii.gz	DTI image
bvecs.txt	bvalues and bvectors for DTI processing

Figure 2.1: Example of good file naming conventions

The specific file names chosen are not as important as consistency in the use of extensions and names, and documentation of what these files are and how they were processed. The more you can keep things the same between projects, the more you will be able to reuse from one study to another with minimal changes.

Directory Structure

Typically, we create a directory for each project. Within that project directory is storage for scripts, masks, data files from other sources (e.g., cognitive or neuropsychological data) and subject neuroimaging data. Subjects may have different timepoints or occasions of imaging data, as well as physiological data and task-related behavioral data. An example hierarchy (for project Udall) is shown in Figure 2.2.

```
/project_space/Udall
├── bin ..... Holds scripts written for this project
│   ├── R ..... Holds R scripts
│   └── tcl ..... Holds tcl scripts
├── data ..... Other sources of data for all subjects
├── lib ..... Holds things that are used by many scripts and analyses
│   ├── makefiles ..... Makefiles for this project
│   │   ├── networkROIs ..... A set of ROI masks for fMRI analysis
│   │   └── tractographyMasks ..... A set of masks for tractography analysis
├── freesurfer ..... Freesurfer (cortical thickness) runs
├── tbss ..... Group-level Tract-Based Spatial Statistics
├── subjects ..... All subject data collected in neuroimaging sessions
│   ├── SUBJECT ..... A specific subject
│   │   ├── session1 ..... Difference subject sessions
│   │   ├── session2
│   │   └── session3
│   │       ├── behavioral ..... Subject behavioral (e.g., task) data
│   │       ├── Dicoms ..... Original dicoms
│   │       ├── Parrecs ..... PAR/REC files, the native Philips scanner format
│   │       ├── physio ..... Physiological data
│   │       └── xfm_dir ..... Registration files
│   ├── session3/SUBJECT ..... Symbolic link to the subject/session3 directories
│   └── incoming ..... Zip files with subject data from the scanner.
```

Figure 2.2: An example of a project directory.

This directory is typically protected so that only members of the project who are permitted by the Institutional Review Board to access the data may `cd` into the directory, using a specific group related to the project. The sticky bit is set on the project directory and the file permissions are set by default to permit group read, write, and execute permission for each individual so that files people create within the directory substructure are accessible by others in the project.

? The **sticky bit** prevents users from deleting files created by other users.

The structure is set up for a longitudinal project, where each subject is imaged at separate time points (sessions). All subject data is stored under the directory subjects (under subdirectories corresponding to each session). However, since often it is convenient to conduct analyses on a single time point, we create a convenience directory at the top level (e.g., session3) which has symbolic links to all of the subjects' session 3 data. This can make it easier to perform subject-specific processing (see section 2).

Notice that in the project directory, we also keep scripts that we write to process the data, other data files, and miscellaneous support files for workflow (in the `lib/` subdirectory). Your naming conventions may differ and there may be more categories than we have, but it is important to decide upon some structure that everyone agrees upon for the project.

We find it useful to locate the results of some subject-specific processing (e.g. co-registration of files, subject-level DTI processing, subject-level task fMRI processing) in the subject/session directories. However, other types of analyses are more conveniently performed in directories (e.g. `freesurfer/`, `tbss/`) that are stored at the top level of the project.

Data files that are collected from non-neuroimaging sources are usually kept in text (tab or comma-separated) form in the `data/` directory so that scripts can find and use them (e.g., to use the subject age or some cognitive variable as a regressor in an analysis).

Finally, note that we have a directory labeled incoming. This is a scratch directory (it could be a symbolic link to a scratch drive) for placing zip files that come from the scanner, that contain the dicoms for the scan, the Philips-specific PAR/REC files, the physiological data logged (used to correct functional MRI data for heart rate and respiration-related signal) and data from scanner tasks (e.g., EPRIME files).

Setting up an Analysis Directory

We have described, up until this point, little examples of makefiles that process subjects within a single directory. However, real studies include many subjects and many analyses; each analysis often produces hundreds of files per subject. Often, several researchers are trying to conduct different analyses on the same data set at the same time.

For these reasons, we use a collection of makefiles to manage a project. This section describes this basic recipe (which is a little complicated). Typically the steps involving creating subject directories and links are done by a script or an "incoming" makefile at the time that the subject data appears from the scanner. Better, if you use an archival system that you can write programs to access (for example, XNAT) a script can create all the links and name files nicely for you.

This relies on the concept of a "symbolic link." This is a file that points to another file or directory. If you remove the link, you will not remove the thing that it points to. However, if you remove the target of the link, the link won't work any more.

To create a symlink, use the `ln` command.

```
$ ln -s target linkname
```

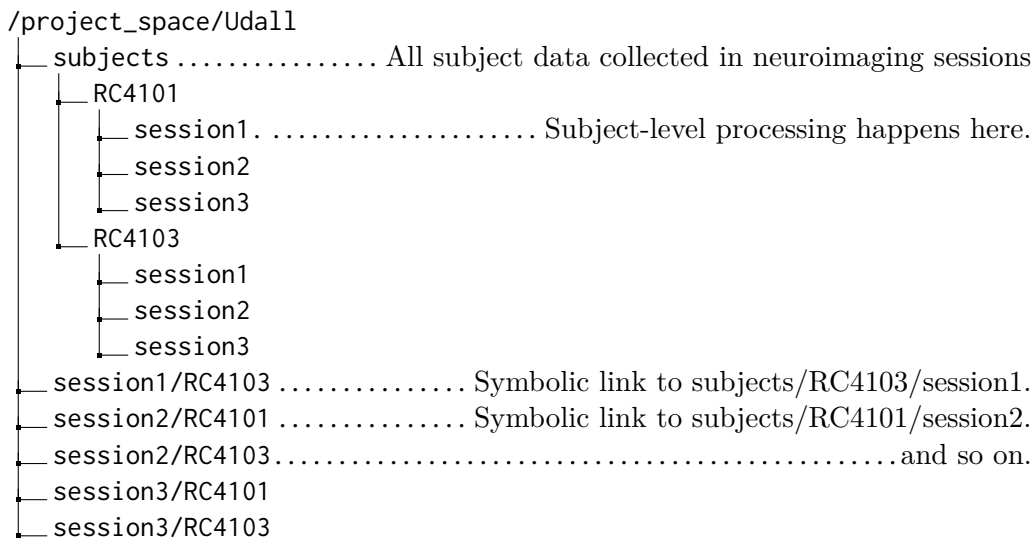

Figure 2.3: A longitudinal analysis directory.

`target` is the file you're linking to, and `linkname` is the name of the new symlink.

Defining the basic directory strucutre

See section 2 for more information on this step, which is important. Let us assume the project home (`PROJHOME`) is called `/project_space/Udall/`. There are three scans per individual. Let us also assume there are two subjects: RC4101 and RC4103. These are the directories that need to be in place. Note we organize the actual files by subject ID and scan session. However, to make processing at each cross-sectional point easier, we create directories with symbolic links to the correct subject/session directories. This flexibility helps in many ways to make cross-sectional and longitudinal analyses easier.

All simple subject-specific processing is well-organized within the subject/session directories (for example, skull-stripping, possibly first level feat, dti analysis, co-registrations). We normally run FreeSurfer in a separate directory, because it likes to put directories in a single place. Analyses that combine data across subjects or timepoints (e.g. TBSS) best go in separate directories.

Figure 2 shows an example directory for a longitudinal study.

Create the session-level makefiles

We will do all the subject-level processing from the PROJHOME/sessionN/ directories. You will need to create a Makefile in PROJHOME/sesion1, PROJHOME/sesion2, and PROJHOME/sesion3 whose only purpose is to run make within all the subject-level directories beneath it. Figure 2.4 shows an example session-level makefile.

? # the make comment character.

```
# Top level makefile
# make recursively all make targets in each subject directory.

# Obtain a list of subject - here all directories of the form RC4
followed by three characters
SUBJECTS=$(wildcard RC4????)

.PHONY: all $(SUBJECTS)

define usage
        @echo Usage:
        @echo "make ...Makes everything in subdirectories"
        @echo "        Remember to specify TARGET=?"
        @echo Other useful targets:
        @echo "make help ...Print this message."
endef

all: $(SUBJECTS)

$(SUBJECTS):
        $(MAKE) --directory=$@ $(TARGET)

help:           $(usage)
```

Figure 2.4: Session-level Makefile

Using the @ before a command stops make from echoing the command before executing it.

Creating the common subject-level makefile for each session

As we described, the session-level makefile only exists to call make within all the subdirectories (symbolic links) in each session. So, we need to create a makefile within each subdirectory. This would be incredibly redundant! In general, we try to

minimize copying of makefile rules where ever possible. We create three files based on a modified version of Figure 2.4:

- `PROJHOME/session1/makefile_common.mk`
- `PROJHOME/session2/makefile_common.mk`
- `PROJHOME/session3/makefile_common.mk`

Note that Figure 2.5 sets a `SESSION` variable. In the example, it is set to "2," but you will tailor this for each of your own sessions. It also sets a `SUBJECT` variable by setting the last part of the file path. This is useful for makefiles you write. If everyone agrees on setting these two variables, you can use them in makefiles that are written to be portable across projects.

We include a little test rule here that merely echoes the subject variable. Most of the rules should normally come from makefiles that are established pipelines, that have been tested with this project (`process_incoming.mk` and `convert_dicoms.mk`). These are stored in `PROJHOME/lib/makefiles/` to make them easier to find and share across sessions.

```
# This is the project home directory
PROJHOME=/project_space/Udall

# Name the session CHANGE THIS FOR EACH SESSION
SESSION=2

# get subject id
subject=$(notdir $(shell pwd))

# put your own rules here

test:
        @echo Testing that we are making $(subject)

# rules from other libraries
include $(PROJHOME)/lib/makefiles/process_incoming.mk
include $(PROJHOME)/lib/makefiles/convert_dicoms.mk
```

Figure 2.5: Subject-level makefile

Creating links to subject-level makefile

The last step now is to create a symbolic link within each subject directory to the appropriate subject-level makefile. For example, within the directory `PROJHOME/session2/RC4101/`, we can type

```
$ ln -s /project_space/Udall/subjects/session2/makefile_common.mk Makefile
```

Running analyses

Once these steps are completed, you can conduct single-subject processing for each session by changing directory to the correct location and issuing the specific `make` command. Here, we illustrate how to make the "test" target for all subjects within `session2/`.

```
cd /project_space/Udall/subjects/session2
make TARGET=test
```

Figure 2.6: Making `test`

Setting Important Variables

`make` uses variables to control a lot of aspects of its own behavior. We describe only a few here; see the manual for the full list. However, it also allows you to set variables so that you can avoid unnecessary changes to Makefiles when you move them to different projects. This is very important, because after going through the hassle of writing a makefile for an analysis once, we would like to reuse as much of it as possible for subsequent studies. It is reasonable to change the recipes to reflect the most appropriate scientific methods or new versions of software, but it's not fun to have to play with naming conventions, etc. We discuss some of the best practices we have found for using variables to improve portability.

Variables that control `make`'s Behavior

`SHELL`

By default the shell used by make recipes is `/bin/sh`. This shell is one of many that you can use interactively in Linux or MacOS, but it probably is not what you are using interactively (because it lacks nice editing capabilities and is less usable than

other shells). Here, we set the default shell to `/bin/bash`, the same as what we use interactively, so that we can be sure when we test something at the command line that it will work similarly in a Makefile.

```
# Set the default shell
SHELL=/bin/bash
export SHELL
```

Figure 2.7: Setting `$(SHELL)`

Note that `SHELL` is accessed `$(SHELL)`, as is other `make` variable.

TARGET

In the strategy that we outline for organizing makefiles and conducting subject-level analyses, we run `make` recursively (i.e., within the subject directories) from the session-level directory. To do this very generally, we call make from the session-level by specifying the TARGET, or what it should "make" within the subject directory. You do not need to do this within the subject directory itself. For example:

(in `subjects/session1`)

```
$ make TARGET=convert_dicoms
```

(in `subjects/session1/s001`)

```
$ make convert_dicoms
```

SUBJECTS

We think it makes life easier to set this to the list of subjects in the study (or subjects for whom data has been collected). For example, given our directory structure, when in the top level for a session, the subject identifiers can easily be found with a wildcard on the directory names. The following statement sets the variable `SUBJECTS` to be all the six-digit files in the current directory (i.e., all the subject directories.)

```
SUBJECTS=$(wildcard [0-9][0-9][0-9][0-9][0-9][0-9])
```

`subject`

Often makefiles are intended to process a single subject. In this case, it is useful to set a subject variable to be the subject identifier.

`SESSION`

When subject data is collected at multiple time points, it is useful to set at `SESSION` variable that can be used to locate the correct subject files.

Other important variables

Ultimately, when it comes time to publish, it is important to state what version of the different software packages you have used. This means that you need to make it difficult to accidentally run a job with a different version of the software. For example, consider this rule to run `DTIPrep`, a program for eddy correction, motion correction, and removal of noise from DTI images.

```
dtiprep/$(subject)_dwi_QCReport.txt: dtiprep$(subject)_dwi.nhdr
        DTIPrep --DWINrrdFile $< -p dtiprep/default.xml --default
--check --outputFolder dtiprep/
```

Figure 2.8: Running DTIPrep in make

Unless you take preventative measures, the version of `DTIPrep` that is used depends on the caller's path. So if the version of `DTIPrep` on one machine is newer than that on another, the results may differ. Alternatively, if my graduate student runs this makefile, and happens to have the newest version fo DTIPrep installed in their `bin/`, that is the version that will be used.

A practical way to control for this is to specify the location of the program (if that conveys version information). See Figure 2.9

```
DTIPREPHOME=/usr/local/DTIPrep_1.1.1_linux64/DTIPrep

dtiprep/$(subject)_dwi_QCReport.txt: dtiprep$(subject)_dwi.nhdr
        DTIPrep --DWINrrdFile $< -p dtiprep/default.xml --default
--check --outputFolder dtiprep/
```

Figure 2.9: Controlling software version

What about when the programs are installed in some default location, e.g. `/usr/local/bin/`, with no version information? In our installation, this occurs frequently when using other workflow scripts that express pipelines more complicated than what might reasonably be put into a makefile.

In the case of simple scripts or statically linked programs, it is fairly easy to copy them into a project-specific bin directory, giving them names that indicate their versions. If you cannot do this, you need to check the version (if the program is kind enough to provide an option that will provide the version) or to check the date that the program was installed, to alert yourself to potential errors. It is useful to set variables for things like reference brains, templates, and so forth.

Variable overrides

It is probably a good idea not to edit the makefile too much once it works. But sometimes, it is useful to reissue a command with different parameters. Target-specific variables may be specified in the makefile and overridden on the command line. In the example below, the default flags for FSL bet are specified as `-f .4 -B -S`.

```
%skstrip.nii.gz: BETFLAGS = -f .3 -B -S
%skstrip.nii.gz: %flair.nii.gz
        bet $< $@ $(BETFLAGS)
```

Figure 2.10: Specifying BET flags in make

However, these can be overriden from the command line as follows:

```
make BETFLAGS="-f .4 -R"
```

Suggested targets

These suggestions come from experience building pipelines. Having conventions, so that similarly named targets do similar kinds of things across different neuroimaging workflows, is rather helpful and comforting, especially when you spend a lot of time going between modalities and tools.

We propose splitting functions into multiple makefiles that can then be called from a common makefile. It is helpful to avoid overruling target names for common targets. Therefore, if the makefile is called, for example, `convert_dicoms.mk`, we will add the text string "convert_dicoms" to the end of each target, as illustrated in the examples below. Top level makefiles (Makefile) can then invoke targets "`all`", "`help`", etc., that depend upon the included makefile targets (See General Recipe for

Setting up an Analysis Directory With `make`).

`all_` (e.g, `all_convert_dicoms`)
This is the default, the first target in the file. Nothing in this target should require human intervention.

`help_` (e.g., `help_convert_dicoms`)
This provides a helpful message describing all the targets in the makefile. Therefore, you can approach any makefile by saying "make help" and get something out of it. The message provided by help should also detail the makefile variables that can be used to control the workflow (e.g., BETFLAGS may supply user-preferred flags to a skull stripping analysis with FSL bet, etc)

`clean_` (e.g., `clean_convert_dicoms`)
Typically this target is used to remove all generated files (e.g., .o, dependency lists) and clean up the directory so that one can type "make" again and regenerate. It usually does not delete configuration information. So the idea is that "`make clean`" will bring you back to square one - starting the pipeline from scratch, but without removing any configuration information you may have created. Good for disaster recovery.

`mostlyclean_` (e.g., `mostlyclean_convert_dicoms`)
Same as clean, but does not remove things that are a pain to recreate (e.g., involve hand checking, or time-consuming analysis). We think that this might be a good "archive state" type of target - removing all files that you don't think you would need later to examine the results.

`output_` (e.g., `output_convert_dicoms`)
Often we write a text file containing summary statistics in some reasonable format for import to Your Favorite Program. Thus, typing "make output" will ensure that summary stats are computed and put into an easy to find place. Sometimes it makes sense to generate the output for each individual subject, as that subject is finished (e.g., inserting some numbers or status into a Google spreadsheet). This is the usual case in a study in which we are doing processing "greedily" as soon as subjects come in. We follow the convention that the first column is the subject id number, and subsequent columns hold the output statistics from the analysis.

.PHONY

.PHONY (the period in front is necessary) is a special target used to tell `make` which targets are not real files. For example, common targets are `all` (to make everything)

and `clean` (to remove everything). If you create a file named "all" or "clean" in the directory with the makefile, suddenly `make` will see that the target file "all" exists, and will not do anything if it is newer than its dependencies.

To stop this rather unexpected behavior, list targets that are not real files as .PHONY:

```
.PHONY = all clean anything_else_that_is_not_a_file
```

.SECONDARY

This target is used to define files that are created as intermediate products by implicit rules, but that you don't want deleted. This is critically important - perhaps a good philosophy is to define here all the files that are a pain to recreate.

Chapter 3

Running `make` in Parallel

Although it is very powerful to be able to use makefiles to launch jobs in parallel, some care needs to be taken when writing the makefile, as with any parallel program, to ensure that simultaneously running jobs do not conflict with each other. In general, it is good form to follow these rules for all makefiles, since it seems highly likely that if you ignore them, there will come a deadline, and you will think to yourself "I have eight cores and only a few days" and you will run `make` in parallel, and all of your results will be subtly corrupted due to your lack of forethought, something you won't discover until you only have a few hours left. This is the way of computers.

Guidelines

There are a few key things to remember when setting running `make` in parallel.

Each line in a recipe is, by default, executed in its own shell

This means that any variables you set in one line won't be "remembered" by the next line, and so on. The best way thing to do is to put all lines of a recipe on the same line, by ending each of them with a semicolon and a backslash (`;\`). Similarly, when debugging a recipe in a parallel makefile gone wrong, look first for a situation where you have forgotten to put all lines of a recipe on the same line. For example, the recipe shown in Figure 3.1 will not work, not matter what, because the first line of this recipe is not remembered by the second, and `brain.volume.txt` will be empty.

```
set foo=`fslmaths brain.nii.gz -V | awk 'print $$2'`
cat $$foo > brain.volume.txt
```

Figure 3.1: A non-functional multi-line makefile

Instead, write the script as shown in Figure 3.2.

```
set foo=`fslmaths brain.nii.gz -V | awk 'print $$2'` ;\
cat $$foo > brain.volume.txt
```

Figure 3.2: A non-functional multi-line makefile

Filenames must be unique

It is very tempting to do something like the following, or the previous example, as you would while interactively using a shell script.

```
some_program > foo.out ;\
do something --now --with foo.out
```

Figure 3.3: How not to name files

This will work great sequentially; first `foo.out` is created and then it is used. You have attended to rule #1 and the commands will be executed together in one shell. But consider what happens when four cores run this recipe independently, in the same directory. Whichever recipe completes first will overwrite `foo.out`. This could happen at any time, so it is entirely possible that process A writes foo.out just in time for process B to use it. Meanwhile, process C can come along and rewrite it again. You see the point. The best way to avoid this problem, no matter how the makefile is used, is to create temporary files (like `foo.out`) that include a unique identifier, such as a subject identifier. For longitudinal analysis we play it safe and always use a timepoint identifier as well. Thus, whether you conduct your analysis using one subdirectory per subject or in a single directory for the entire analysis, you can take the recipe you have written and use it without modification.

Many neuroimaging software packages use the convention that files within a subject-specific directory that contains a subject identifier can have the same names. In that case, just go with it: They have done that in part to avoid this confusion while running the software in parallel on a shared filesystem.

Separate time-consuming tasks

Try to separate expensive tasks into separate recipes.

Suppose you have two time-consuming steps: `run_a` and `run_b`. `run_A` takes half an hour to generate `a.out` and `run_b` takes an hour to generate `b.out`. Additionally, `run_a` must complete before the other can begin. You try the rule in Figure 3.4

```
a.out b.out: b.in
        run_a ;\
        run_b
```

Figure 3.4: The wrong way to run two long tasks

This works, but suppose another task only needs `a.out`, and doesn't depend at all on `b.out`. You would spend a lot of extra time generating `b.out`, especially if this was a batch job. So it is better to write the rule in two separate lines, as in Figure 3.5.

```
a.out: b.in
        run a

b.out: a.out
        run_b
```

Figure 3.5: The right way to run two long tasks

Running Makefiles in Parallel

Using multiple cores

Most modern computers have multiple processing elements (cores). You can use these to execute multiple jobs simultaneously by passing the `-j` option to `make`.

First you need to know the number of cores on your machine. The command `nproc` might work, or you can ask your system administrator.

You have to be careful not to start a lot more jobs than you have cores, other wise the performance of all the jobs will suffer. Let us assume there are four cores available. Pass the number of jobs to `make`.

```
$ make -j 4
```

Now you will execute the four cores simultaneously. This will work well if no one else is using your machine to run jobs. If they are, `kill` their jobs. It's your machine. [1]

[1] Mostly joking. Don't kill people's jobs indiscriminately.

Recall `make` will die when it obtains an error. When running jobs in parallel, `make` can encounter an error much sooner. If one job dies, all the rest will be stopped, leaving your process in a bad state. This is rarely the behavior you want, as typically each job is independent of the others. To tell make not to give up when the going gets tough, use the `-k` flag.

```
$ make -j 4 -k
```

Using the grid engine

Four cores (or even eight or 12) is nice, but a cluster is even nicer. Our cluster environment is the Sun Grid Engine (SGE) which is a batch queuing system. This software layer allows you to submit jobs, queues them out, and farms them out to one of the many machines in the cluster.

! You can submit jobs from any grid engine. broca and pons are the easiest to type.

Setting `FSLPARALLEL` for FSL jobs

There are two ways to use `make` to run jobs in parallel on the SGE. The first is to use the scripts for parallelism that are built in to FSL. In our configuration, all you need to do is set the environment variable `FSLPARALLEL` from your shell as follows:

```
$ export FSLPARALLEL=true
```

This must be done before running `make`! Then, you run your makefile as you would on a single machine, on a machine that is allowed to submit jobs to the SGE (check with your system administrator to find out what this is). What will happen is that the FSL tools will see that this flag is set, and use the script `fsl_sub` to break up the jobs and submit them to the SGE. You do not need to set the `-j` flag as above, because FSL will control its own job submission and scheduling.

Note that this trick will only work if you are using primarily FSL tools that are written to be parallel. What happens if you want to use something like `bet` on 900 brains, or other tools that are not from FSL?

Using `qmake`

By using `qmake`, a program that interacts with the SGE, you can automatically parallelize jobs that are started by a makefile. This is a useful way to structure your workflows, because you can run the same neuroimaging code on a single core, a multicore, and the SGE simply by changing your command line. You may need to

discuss specifics of environment variables that need to be set to run `qmake` with your system administrator. If you are using `make` in parallel, you also will probably want to turn off `FSLPARALLEL` if you have enabled it by default.

There are two ways that you can execute qmake, giving you a lot of flexibility. The first is by submitting jobs dynamically, so that each one goes into the job queue just like a mini shell script. To do this, type

```
$ qmake -cwd -V -- -j 20 all
```

The flags that appear before the "--" are flags to `qmake`, and control grid engine parameters. The `-cwd` flag means to start grid engine jobs from the current directory (useful!) and `-V` tells it to pass all your environment variables along. If you forget the `-V`, we promise you that very bad things will happen. For example, FSL will crash because it can't find its shared libraries. Many programs will "not be found" because your path is not set correctly. Your jobs will crash, and that earthquake will kill all of us.

On the opposite side of the -- are flags to `make`. By default, just like normal `make`, this will start exactly one job at a time. This is not very useful! You probably want to specify how much parallelism you want by using the -j flag to make (how many jobs to start at any one time). The above example runs 20 jobs simultaneously. The last argument, `all`, is a target for `make` that is dependent upon the particular makefile used.

One drawback of executing jobs dynamically is that make might never get enough computer resources to finish. For this reason, there is also a parallel environment for `make` that reserves some number of processors for the `make` process and then manages those itself. You can specify your needs for this environment by typing

```
$ qmake -cwd -V -pe make 1-10 -- freesurfer
```

This command uses the `-pe` to specify the parallel environment called `make` and reserves 10 nodes in this environment. The argument to `make` is `freesurfer` in this example.

How long will everything take?

A good thing to do is to estimate how long your entire job will take by running `make` on a single subject and measuring the "wall clock" time, or the time that it takes between when you start running it and when it finishes. If you will be going home for the night, add a command to print the system date (`date`) as the last line in the recipe, or look at the timestamp of the last file created. Suppose one subject takes 12

hours. Probably other subjects will take, on average, the same amount of time. So you multiply the number of subjects by 12 hours, and divide by 24 to get days. For 100 subjects, this job would take 50 days. This calculation tells you that it would be a long time to wait for your results on your four-core workstation (and in the meantime, it would be hard to do much else).

Suppose you have a cluster of 75 processors, and 100 subjects. If you can use the entire cluster, you can divide the number of subjects by processors and round up. This gives you two. If you multiply this by your average job time, you find that you can complete this job on the cluster in one day. If you think about this, you see that if you had 100 processors, you could finish in half the time (12 hours) because you would not have to wait for the last 25 jobs to finish.

Consideration of memory

Processing power is not the only resource to consider when running jobs in parallel. Some programs (for example, `bbregister` in our environment) seems to take a lot of memory. This means that attempts to start more jobs than the available memory can support will cause the jobs to fail. A good thing to do is to look at the system monitor while you are running a single job, and determine what percentage of the memory is used by the program (find a computer that only you are using, start the monitor before you begin the job, and look at how much additional memory is used, at a maximum, while it runs). If you multiply this percentage by the number of cores and find that you will run out of memory, do not submit the maximum number of jobs. Submit only as many as your available resources will support.

Troubleshooting make

One flaw of `make` (and indeed, many UNIX and neuroimaging programs and just life in general) is that when things do not go as expected, it is difficult to find out why. These are some hints and tricks to help you to troubleshoot when things go wrong.

Find out what make is trying to do

Start from scratch, remove the files you are trying to create, then execute `make` with the `-n` flag.

```
$ make -n all
```

You can also place the flag at the end of the command.

```
$ make all -n
```

The -n flag shows what commands will be executed without executing them. This is very handy for debugging problems, as it tells you what make is actually programmed to do. Remember, computers do exactly what you tell them.

Check for line continuation

Ensure that you have connected subsequent lines of a recipe with a semicolon and a backslash (;\). If you don't do this, each line will be run in a separate shell, and variables from one will not be readable in the other.

No rule to make target!

Suppose you write a makefile and include in it what you think should be a rule to create a specific file (say, data_FA.nii.gz). However, when you type make, you get the following message:

```
make:  *** No rule to make target `data_FA.nii.gz', needed by `all'
```

This can be rather frustrating. There are multiple ways this error message could be generated. We recommend the following steps to diagnose your sickly makefile.

1. Read the error message.
   Seriously. It is tempting to try to read a program's mind, but inevitably this fails. Mostly because they don't have minds. In the error above, the key aspects of the error above are as follows:
   (a) The error comes from make, as you can see by the fact that the line starts with make:. It is also possible that your Makefile fails because a program that you called within it fails (and you either did not specify the -k flag to keep going, or there is nothing left to do).
   (b) make claims that there is "no rule" to make the target "data_FA.nii.gz." This tells you that make could not find a target: dependency combination to create this particular target.
   (c) make is trying to create this target to satisfy the target all. This tells you that it was trying to make data_FA.nii.gz because this is a dependency of the target all.
2. If your error is indeed coming from make, then try to pinpoint the rule that is failing. *Look* at your makefile and check that the rule looks correct. Do all the dependencies of the rule (the things to the left of the colon) exist? Are they where they should be? Can you read them?
3. If everything looks ok, you are missing something. Rules that expand patterns and that use variables can be tricky to interpret (the same way that you

generally like a debugger to look at code). To see what make is really doing with your rules, use the `-p` flag to print the database and examine the rules. We suggest doing this in conjunction with `-n` so that you do not actually execute anything. Pattern matching has an odd behavior where, if there is no rule to create a dependency, it will tell you there is no rule to create the target. For example, Figure 3.6 will fail with

```
...  No rule to make target `<subj>_T1_brain.nii.gz' ...
```

not

```
...  No rule to make target `foo' ...
```

```
%_T1_brain.nii.gz: %_T1.nii.gz foo
        bet $< $@
```

Figure 3.6: Pattern-matching error handling

Suspicious continuation in line

If you get this error while trying to save a makefile (using emacs, which is smart enough to help you ensure the syntax of your makefile is correct), it means that you probably have a space after the trailing `;\` at the end of the line. No, you can't have indiscriminate whitespace in a makefile!

`make` keeps rebuilding targets

A more subtle problem occurs when your makefile works the first time, but it also "works" the second time ... and the third ... and so on. In other words, it keeps rebuilding targets even when they do not "need" to be rebuilt. This means that `make` examines the dependencies of the target, decides that they are newer than the target (or that the target does not exist) and executes the recipe.

This cycle never stops if, somewhere, a target is specified that is never actually created (and that is not marked as phony). This is easy to do, for example, when trying to execute more complex neuroimaging pipelines (such as those provided by FSL) that create entire directories of files with specific naming conventions. Check that the commands that keep re-executing really create the targets that would indicate that the command has completed. For example, when running `feat` with a configuration file, the name of the output directory is specified in the configuration file. This must be the same as the target in the Makefile, or it will never be satisfied.

make doesn't rebuild things that it should

This type of error usually indicates a logical problem in the tree of targets that begins with what you have told make to do. Recall that by default, make builds the first target that appears in the makefile. If you have included makefiles at the beginning of your file, it might happen that the default target is not actually what you think it is.

Part II

Practica

Part II will guide you through four `make` classes, using data available on the IBIC system. Each class is structured as a brief lecture followed by a practicum designed to be completed as you read the chapter.

Chapter 1

Overview of make

Lecture: Making Breakfast

make is a tool for implementing a workflow, or a sequence of steps that you need to execute. Probably the way that most of you have been implementing workflows up until this point is with some kind of program, possibly a shell script. The goal of this lecture and practicum is to motivate the reasons why you might want to move from a shell script to some language for better specifying workflow.

A few elements of good workflow systems are:

- Reproducibility, which a shell script can provide.
- Parallelism, which a shell script cannot describe.
- Fault tolerance, which a shell script can absolutely not deal with. If there is a problem in subject nine of your 100-subject loop, subjects 10-100 will not even begin, even if nothing is wrong with them.

We will start with a conceptual example of the difference between shell scripts and make. make is organized into code blocks called "recipes," so for our conceptual example, we will create a "waffwich", a delicious[1] breakfast food that one can eat with one hand on the way to the bus stop. We can create a pseudocode example of how to make a waffwich:

! **Pseudocode** does not follow any real programming syntax

This "script" tells you very neatly what to do. However it enforces a linear ordering on the steps that is unnecessary. If you had lots of sandwich-making resources you could use, you would not know from this description that the sandwiches do not have to be made sequentially. You could, for example, toast all the waffles for each person before spreading the peanut butter on them. You could make person c's waffwich before person a's, but the script does not specify this flexibility.

[1] So I'm told.

```
for person in a b c
do
toast waffle
spread peanut_buttter --on waffle
arrange berries --on waffle --in squares --half
cut waffle
fold waffle
done
```

Figure 1.1: Creating a waffwich

You also would have no way of knowing what ingredients the waffwich depends on. If you execute the steps, and realize you don't have any berries, your script will fail. If you go out to the store and get berries, you would have to rewrite your script to selectively not retoast and spread peanut butter on the first person's waffle, or you would just have to do everything again.

This is a conceptual example, but in practice, having this information in place saves a whole lot of time. In neuroimaging, because we often process many subjects simultaneously, exploiting parallelism is essential. Similarly, there are often failures of processing steps (e.g., registrations that need to be hand-tuned, tracts that need to be adjusted). With higher-resolution acquisition, jobs run longer and longer, so the chance of computer failure (or a disk filling up, or something) during the time that you run a job is more likely. But all this information cannot be automatically determined from a shell script.

There has been a lot of work done on automatically inferring this information from languages such as C and MATLAB, but because shell scripts call other programs that do the real work, there is no way to know what inputs they need and what outputs they are going to create.

The information in Figure 1.1 can alternatively be represented as a directed acyclic graph (Figure 1.2).

! There is no circularity in this graph.

A graph like this is a very good way of describing a partially ordered sequence of steps. However, writing a graph is a really nasty way of writing a program, so we need a better syntax for making dependencies. The `make` recipe for a waffwich would look something like Figure 1.3.

We must note: makefiles are not linear, unlike shell scripts. `toastedwaffle` needs to be created before `waffwich`, but does not need to appear before it in the script. `make` will fall through in the order it needs.

! The arrows indicate which file needs to exist to make the next one.

However, we've reached the point where we have to leave this conceptual example because `make` is aware of whether something exists or not, and whether it has to

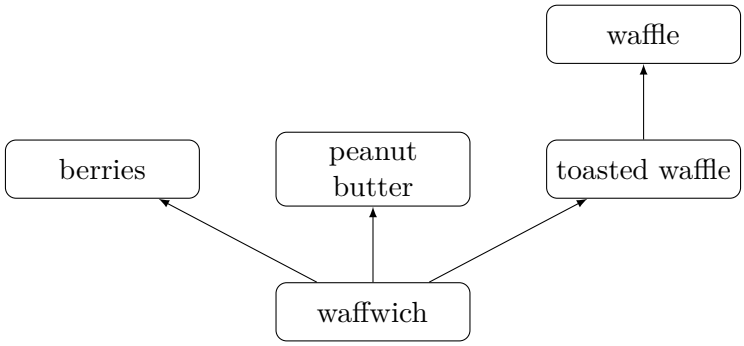

Figure 1.2: How to make a waffwich

```
waffwich: toastedwaffles berries PB
        spread peanut_buttter --on waffle
        arrange berries --on waffle --in squares --half
        cut waffle
        fold waffle

toastedwaffle: waffle
        toast waffle
```

Figure 1.3: Using `make` to make a waffwich

make it. It does this by assuming that the target and dependencies are files, and by checking the dates on those files. There are many exceptions to this; but let's accept this simplification for the moment, and move on to a real example with files.

Practical Example: Skull-stripping

Manipulating a single subject

Follow along with this example. Copy directory
`/project_space/makepipelines/oasis-multisubject-sample/`
to your home directory.

! Don't forget to use `cp -r <old> <new>` to copy all files in a directory.

Your task is to skull strip all of these brains for subsequent use by `FSL`. However, note that first you need to reorient each image (try looking at it in `fslview`). So there are two commands you need to execute:

```
$ fslreorient <subject>_raw.nii.gz <subject>_T1.nii.gz
```

and

```
$ bet <subject>_T1.nii.gz <subject>_T1.skstrip.nii.gz
```

By default, `make` reads files called "Makefile." According to the manual, GNU looks for `GNUmakefile`, `makefile`, and `Makefile` – in that order – but if you use just one convention, you are unlikely to get confused.

Open the makefile that came with the directory. You should have the following:

```
OAS2_0_T1_skstrip.nii.gz: OAS2_0001_T1.nii.gz
        bet OAS2_0001_T1.nii.gz OAS2_0001_T1_skstrip.nii.gz

OAS2_0001_T1.nii.gz: OAS2_0001_raw.nii.gz
        fslreorient2std OAS2_0001_raw.nii.gz OAS2_0001_T1.nii.gz
```

Figure 1.4: Makefile as copied

Play around with this. What happens when you execute `make` from the command line? What is it really doing?

Change the order of the rules. What happens? Note that by default, `make` starts by looking at the first target (the first thing with a colon after it). That is why, when you change the order of the rules, you get different outcomes, but *not* because it is reading them one by one. It's creating a directed acyclic graph, but it has to start somewhere.

You can also tell it where to start. Delete any files that you may have created. Leave the makefile with the rules reordered. Now type:

```
$ make OAS2_0001_skstrip.nii.gz
```

Now `make` starts with this target and goes on from there, working backward to figure out what it needs to do.

Pattern rules and multiple subjects

This is great, but so far we have only processed one subject. You can create rules for each subject by cutting and pasting the two rules you have and editing them. But that sounds like a huge chore. And what if you get more subjects? Yuck.

You can specify in rules that you want to match patterns. Every file begins with the subject identifier so you can use the `%` symbol to replace the subject in both the

target and the dependencies. However, what do you do in the recipe when you don't know the actual names of the file you're presently working with? The `%` won't work there. However, the symbol `$*` does. But that can get visually ugly. One common shortcut is to use the automatic variable `$@` to replace the target, and `$<` for the first dependency.

! Patterns can be matched anywhere in the filename, not just at the beginning.

Using these new variables, our code can now be written:

```
%_T1_skstrip.nii.gz: %_T1.nii.gz
        bet $< $@

%_T1.nii.gz: %_raw.nii.gz
        fslreorient2std $< $@
```

Figure 1.5: Pattern-matched Makefile

But try executing `make` now, with just those rules. You should get an error that there are no targets. Note that `%` does not work as a wildcard as in the `bash` shell. If you are used to that behavior, you might expect `make` will sense that you have a lot of files with subject identifiers, and it should automatically expand the `%` to match them all. It will not do that, so you have to be explicit about what you want to create.

For example, try typing:

```
$ make OAS2_0001_T1_skstrip.nii.gz
```

Now make has a concrete target in hand, and it can go forth and figure out if there are any rules that match this target (and there are!) and execute them.

`make` will first look for targets that match exactly, and then fall through to pattern matching.[2]

It can be helpful to visualize pattern matching as a sieve that catches and "knocks off" the part that it matched, leaving only the stem. It will additionally strip any directory information when present. For example, `foo/%_T1.nii.gz: %.nii.gz` will work just as you would like.

Figure 1.6 shows how `make` identifies a pattern from an input and applies it to match the dependencies. Here, you can see that it is important to not include trailing (or leading!) underscores or other characters in your expected pattern.

[2] It will use more specific rules before more generic ones: see the GNU `make` manual for more information on this behavior.

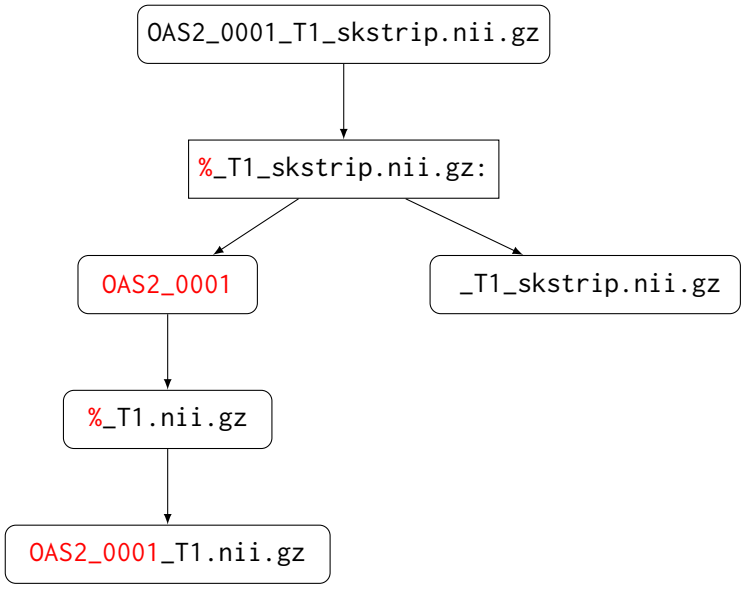

Figure 1.6: Pattern-matching as a sieve

Phony targets

Great, but how do you specify that you want to build all of these subjects? You can create a phony target (best placed at the top) that specifies all of the targets you really want to make. It is called a phony target because it does not refer to an actual file.

```
skstrip: OAS2_0001_T1_skstrip.nii.gz OAS2_0002_T1_skstrip.nii.gz
```

Add as many subjects as you're patient enough to type and execute

```
$ make skstrip
```

Of course, typing out all the subjects (especially with ugly names like `OAS2_0001_T1_skstrip.nii.gz`) is almost as time-consuming as copying the rule multiple times. Thankfully, there are a lot of ways to create lists of variables, shell commands, wildcards, and most other things you might think of. Conveniently, there is a file called `subjects` in this directory. We can get a list of subjects by using the following `make` command.

```
SUBJECTS=$(shell cat subjects)
```

Now we can write a target for skullstripping.

```
skstrip: $(SUBJECTS:=_T1_skstrip.nii.gz)
```

You also now must notify `make` to tell it that `skstrip` is not a real file. Do this by using the following statement:

```
.PHONY: skstrip
```

The pattern substitution for changing subjects to files comes from section 8.2 of the GNU `make` manual. You can look up the other options, but the basic function of this command is to add the affix `_T1_skstrip.nii.gz` to each item in `SUBJECTS`.

The command uses the more general syntax, which can be used to remove parts of names before adding your new affix.

```
$(var:suffix=replacement)
```

As an esoteric example, this command could be used to replace all the final 5s with the string "five."

```
$(SUBJECTS:5=five)
```

`make` should now fail and tell you that there is no rule to make the target `OAS2_00five_T1.nii.gz`.

Secondary targets

Now if you type `make` here it is going to go through and do everything, if there's anything to do. But it's easier to see the point of secondary targets if you use the `-n` flag for `make`, which asks `make` to tell you what it's going to do, without actually doing it.[3] It's puzzling, isn't it, that all the T1 files are deleted? What's wrong here?

Because we explicitly asked for the skull-stripped brains, `make` figured out it needed to make the T1 brains. This is an *implicit* call. `make`, once it has finished running, goes back and erases anything that was implicitly created. This is a good thing, or a weird thing, depending on how you look at it. You can specify to `make` to keep those intermediary targets by adding it to the `.SECONDARY` target, like so:

```
.SECONDARY: $(SUBJECTS:=T1.nii.gz)
```

[3] Trevor likes to use `make <cmd> -n`, which makes it easier to hit <UP> and remove the flag without any annoying arrowing around.

An alternative method is to pass any targets you want to keep to the phony variable.

```
skstrip: $(SUBJECTS:=_T1_skstrip.nii.gz) $(SUBJECTS:=_T1_skstrip.nii.gz)
```

This is a little easier to interpret, as well. It also provides the advantage of allowing `make` to realize when an intermediary target needs to be remade. For example, if you erased only a T1 image (perhaps to rerun `bet`), and told it to `make skstrip`, it would look and see that all the T1_brain images are newer than the raw images, and therefore, no change needs to be made. While this problem could be solved by also removing the T1_brain image, that's more work and we want to do as little of that as possible.

`make clean`

What if you want to clean up your work? You've been deleting files by hand, but with phony targets, you don't have to. A common thing is to create a target called `clean`, which removes everything that the makefile created to bring the directory to its virgin state.

```
clean:
	rm -f *_T1.nii.gz *_T1_brain.nii.gz
```

Figure 1.7: `make clean`

Note that in the recipe, it's totally fine to use wildcards. Also note the use of the `-f` flag. In this instance, it hides error messages from attempting to erase nonexistent files. Be very careful in its use, however.

This convention also only enforces discipline; you need to add every new file that you create in a makefile to the clean target. More dangerous is the that sometimes to get a neuroimaging subject to a certain state, you need to do some handwork that would be expensive to recreate. For example, no one wants to blow away freesurfer directories after doing hand editing.

For these reasons, there exists the looser convention of `mostlyclean`, which means to remove things that can easily be regenerated, but to leave important partial products (e.g., your images converted from dicoms, freesurfer directories, final masks, and other things).

Also add `clean` to your list of phony targets:

```
.PHONY: skstrip clean
```

Chapter 2

`make` in Context

Lecture: Organizing Subject Directories

In practice, a project will involve multiple subjects, each with several types of scans. One could imagine a universal standard where every NifTI file contained information about where it came from and what information it contained, and all neuroimaging programs understood how to read and interpret this information. We could dump all of our files into a database and extract exactly what we wanted. We wouldn't need to use something as archaic as `make` because we could specify dependencies based on the products of certain processing steps, no matter what they were called.

If you can't imagine this, that's totally fine, because it's a very long way off and won't look like that when it's here. Right now, we need to work with UNIX file naming conventions to do our processing.

Therefore, selecting good naming conventions for files and for directories is key. `make` specifically depends upon naming conventions so that people can keep track of what programs and steps were used to generate what projects.

Many of our studies are longitudinal. Even if they don't start out that way, if you are trying to study a developmental or degenerative disease, and you scan subjects once, it is often beneficial to scan them again to answer new questions about longitudinal progression. However, this aspect of study design poses some challenges for naming conventions.

Because of the way that tools like XNAT like to organize data, we organize multiple visits for each subject as subdirectories under that subject's main data directory. However, this organization is highly inconvenient for processing with `make`.

Practical Example: A More Realistic Longitudinal Directory Structure

Organizing Longitudinal Data

Follow along with this example. Copy directory
`/project_space/makepipelines/oasis-longitudinal-sample-small/`
to your home directory using the following command:

```
$ cp -r /project_space/makepipelines/oasis-longitudinal-sample-small/
```

This is a very small subset of the OASIS data set, which consists of a longitudinal sample of structural images. There are several T1 images taken at each scan session, and several visits that occurred a year or more apart. IâĂŹve reduced the size by taking only one of the T1 images for each person, for each visit, and only one a small sample of subjects.

Look at the directory structure of `subjects`. A useful command to do this is called `find`. For example, if you are in the `subjects` directory you can type:

```
$ find .
```

You can see that as we have discussed, each subject has one to five separate sessions. The data for each session (here, only a single MPRAGE) is stored under each session directory. I realize that creating a directory to store what is right now a single scan seems a bit like overkill, but in a real study there would be several types of scans in each session directory. Here, to focus on the directory organization and how to call make recursively, we are only looking at one type of scan.

Normally there are two types of processing that are performed for a study. The first are subject-level analyses — in short, analyses that are completely independent of all the other subjects. The second are group-level analyses, or analyses that involve all the subjects data. In general, a good rule of thumb is that the results of subject-level analyses are best placed within the subject directories.

Group-level analyses seem to be best found elsewhere in the project directory — either at a specific timepoint or organized at the top level.

Create the directory `visit1` within your copy of `oasis-longitudinal-sample-small`.

```
$ mkdir visit1
```

Now `cd` into it:

```
$ cd visit1
```

What we want to do is create the symbolic links for each subject's first visit here. You can do one link by hand:

```
$ ln -s ../subjects/OAS2_0001/visit1 OAS2_0001
```

You can also do this in bulk. Remove this link that you created and use the program in that directory (`makelinks`) to create all the subject links for visit1.

You can look to see that this is a link with `ls -l`, `ls -F` to see in shorthand that it is indeed a link, or `ls -L` if you want to follow it. With symbolic links is is very helpful to be able to know where you are.

! The current directory won't be in your PATH, so make sure to call it with `./makelinks`.

Recursive `make`

Now let's look at the Makefile (`visit1/Makefile`). This is just a top-level "driver" makefile for each of the subjects. All it does is create a symbolic link, if necessary, to the subject-level makefile, and then it goes in and calls `make` in every subdirectory.

Do you know why is the subject target a phony target? The reason is that we want make to be triggered within the subject directory every time we call it.

Create the symbolic links to the subject-level makefile as follows:

! Your `ls` command may be aliased to something pleasing, in which case you might see slightly different behavior than described here.

```
$ make makefiles
```

Do you know why we use the `$PWD` variable to create the link? If we used a relative path to the target file, what would happen when we go to the subject directory?

Let us see how this works. Look at the subject-level makefile. Go into a subject directory and run `make`.

By now you might be getting really tired of seeing those same `fslreorient2std` and `bet` commands. By default, `make` will echo the commands that it executes to the terminal. If you would like it to stop doing that, you can tell it not to do that by prepending the `@` character to each line.

```
%_T1_skstrip.nii.gz: %_T1.nii.gz
        @bet $< $@
```

Figure 2.1: The `@` character hides commands

Now go find the same subject via the subject subtree:

```
$ cd  /oasis-longitudinal-sample-small/subjects/visit1/OAS2_0001/visit1
```

Type `make` and see that it works. Part of this magic is that we set the subject variable correctly, even though where it appears in a directory path is different in each place.

There is a useful little rule defined in the GNUMake Cookbook (Chapter 2, p. 44) that may be useful for checking that you have set variables correctly. Add the following lines to the subject-level makefile.

```
print-%:
        @echo $* = $($*)
```

Figure 2.2: Printing out the value of variables

Now if you type:

```
$ make print-subject
```

You can see the value that the variable `subject` is set to. Note that even if you place this new rule at the very top of the makefile, it will not execute this rule by default. It will fall through to the next non-pattern rule. This is one of the ways in which pattern rules (implicit rules) differ subtly from explict rules.

Running Make Over Multiple Subject Directories

Now that we have verified that the individual makefile works, we can go to the `visit1` directory and process all the subjects. First, go back and edit the subject-level makefile to remove the `@` characters in front of the commands so that they are printed.

From within the `visit1` directory, type:

```
$ make -n TARGET=skstrip
```

Note that what this is doing is (recursively) going into each subject directory and calling make. It will do this whether or not there is anything to do within the subject directory, because each subject directory has no dependencies. However, because we have specified the `-n` flag, it prints out what commands it will do without actually executing them.

We can do this work in parallel. Bring up a system monitor (`gkrellm` is installed on the IBIC systems). See how many processing units you have and how busy they are. Note these numbers and get familar with the configuration of the computers that you have in the lab. In general, each job will require a certain amount of memory and can use at maximum one CPU. A safe calculation for most things is that you can normally run as many jobs at one time on a computer as you have CPUs. This is a huge oversimplification but it will suffice for now.

You can specify to make how many CPUs to use. For example, if we specify the `-j` flag with an argument of 2 (processors), we can parallelize execution of `make` over two CPUs.

```
$ make -j 2 TARGET=skstrip
```

If you specify `make -j` without any options, it will use as many CPUS as you have on your machine. This is great if all the work you have "fits" into the computing resources that are available. However, if it does not, you can use a computing cluster.

In our environment, we use the Sun Grid Engine (SGE) to distribute jobs across machines that are "clustered" together. To run on a cluster, you need to be logged on one of the machines that is allowed to submit to the grid engine. Once there, you can use the command:

```
$ qmake -cwd -V -- -j 20 TARGET=skstrip
```

Here, we use `qmake` instead of `make` to interact with the grid engine. The `-cwd` flag says to run from the current working directory, and `-V` says to pass along all your environment variables. You will normally want to specify both of these flags. The `--` specifies to `qmake` that we are done specifying `qmake` flags and are now giving normal flags to `make`. For example, in this command we specify 20 jobs.

As an optional exercise, here you might want to set up the same directory structure for `visit2` and build everything from scratch in parallel.

Running FreeSurfer

Now let us look at an example of a subject-level analysis that we typically don't run within the subject directories. FreeSurfer is a program that we use for cortical and subcortical parcellation. It itself is a complicated neuroimaging pipeline that is built using `make`. It likes to put all subject directories for an analysis in one directory, which makes it difficult to enforce the subject-level structure described. But in general, it is much wiser to work around what the programs want to do than

to reorganize their output in ways that might break when the software is updated! This is our approach.

The FreeSurfer example is documented in REFERENCEHERE.

Chapter 3

Documenting a makefile for this manual

This chapter will show you how to document (in LaTeX) a makefile for this manual. While LaTeX may look intimidating if you haven't used it before, it is fairly straightforward to use. We have created a number of tools to help you insert your makefile into this manual.

Makefile to LaTeX

There are a number of things that need to be changed in a bare makefile in order to import it to the LaTeX structure used for this manual. To do the basic stuff, call the script `makeToTex`[1] on your makefile(s). It can accept any number makefiles and outputs `<file>_texed.txt`. It has two flags, `h` and `v`, which call the help menu and verbose mode, respectively.

There are two main things that need to happen: certain symbols need to be escaped in LaTeX, and recipes need to be formatted in a macro to make them pretty. Symbols like "$" and "#" have special functions in LaTeX, and will result in catastrophe if you don't fix them.

The functionality of the script is documented in Appendix A. Please refer to it if you have any troubles.

Now to bring your new makefile into a LaTeXdocument!

[1] Please report bugs to Trevor at tkmday@uw.edu.

Editing a .tex File

Any script compiled with the main document will include the necessary environment `makefileread`. In your LaTeXdocument, create a newline, ensuring the indentation is consistent and insert the command `\begin{makefileread}`. If you are using a LaTeXeditor, it may autocomplete with the last line, `\end{makefileread}`. If not, you'll have to type it yourself.

Paste your code between those two commands. It should be one level more indented.

To insert text comments between sections of code, you may terminate the environment, add your text and begin a new environment with the rest of the code. Refer to the style guide (Appendix B) for stylistic choices we use to keep our text consistent. There are many idiosyncrasies to LaTeX, too many to go into here, but a few of note:

- Quotes are written with two backticks (opening) or two apostrophes (closing).
- Please leave a blank line between paragraphs.
- Special characters should always be escaped (see Appendix A or Internet resources).

Feel free to scroll through chapter files to look for examples of usage if you are in doubt. The answers to further questions can probably be found online (I especially like `http://tex.stackexchange.com`).

How Makefile Documentation Should Look

Notice how documentation will begin on a new page and a new page will be created at the end of the documentations. This is necessary to allow for wider margins that make the code easier to read.

```
PROJHOME=/projects2/act-plus/subjects/session1

cwd=$(shell pwd)

SUBJECTS=$(shell cat /projects2/act-plus/uds/good_subjects.txt)

export OMP_NUM_THREADS=1

export SUBJECTS_DIR=/projects2/act-plus/freesurfer
export QA_TOOLS=/usr/local/freesurfer/QAtools_v1.1

                This is some commentary on your makefile ----------------------------------------------------------

export FREESURFER_SETUP = /usr/local/freesurfer/stable5_3/SetUpFreeSurfer.sh
export RECON_ALL = /usr/local/freesurfer/stable5_3/bin/recon-all $(RECON_FLAGS)
export TKMEDIT = /usr/local/freesurfer/stable5_3/bin/tkmedit

define usage
@echo Usage:
@echo “make  or make interactive                                                    Makes interactive targets”
@echo “make noninteractive                                                  Makes noninteractive targets”
@echo
@echo Noninteractive targets:
@echo “make setup                                                 Copies source files to this directory”
@echo “make freesurfer                                                                  Runs freesurfer”
@echo
@echo Other useful targets:
@echo “make clean                                                       Remove everything!  Be careful!”
@echo “make mostlyclean                                           Remove everything but the good bits.”
@echo “make help                                                                  Print this message.”
@echo
@echo Variables:
@echo “RECON_FLAGS                                                 Set to flags to recon-all  by default”
@echo “WAVE                                                      1  2 or 3 to select subjects  for setup”
endef

export SHELL=/bin/bash

.PHONY: qa clean mostlyclean output noninteractive

noninteractive: setup freesurfer

all: noninteractive

output=$(SUBJECTS:%=%/mri/aparc+aseg.mgz) $(SUBJECTS:%=%/mri/brainmask.nii.gz)
freesurfer: $(output)

qafiles=$(SUBJECTS: %=QA/%)
```

```
qa: $(qafiles)

QA/%: %
	source $$FREESURFER_SETUP ;\
	$(QA_TOOLS)/recon_checker -s $*

%/mri/aparc+aseg.mgz: $(PROJHOME)/%/memprage/T1.nii.gz
	rm -rf `dirname $@`/IsRunning.*
	source /usr/local/freesurfer/stable5_3/SetUpFreeSurfer.sh ;\
	export SUBJECTS_DIR=$(SUBJECTS_DIR) ;\
	/usr/local/freesurfer/stable5_3/bin/recon-all -i $< -subjid $* -all ;\
	/usr/local/freesurfer/stable5_3/bin/recon-all -s $* -T2 $(PROJHOME)/$*/flair/Flair.nii.gz -T2pial

%/mri/brainmask.nii.gz: $(SUBJECTS_DIR)/%/mri/aparc+aseg.mgz
	source /usr/local/freesurfer/stable5_3/SetUpFreeSurfer.sh ;\
	mri_convert $(SUBJECTS_DIR)/$*/mri/brainmask.mgz $@

clean:
	echo rm -rf $(inputdirs)

mostlyclean:
	@echo Here I would delete things that are not necessary after all is said and done.

output:
	@echo Nothing yet

setup: $(SUBJECTS)

%: $(PROJHOME)/%/memprage/T1.nii.gz
	mkdir -p $@/mri/orig ;\
	cp $^$@/mri/orig ;\
	cd $@/mri/orig ;\
	mri_convert T1.nii.gz 001.mgz

help:
	$(usage)
```

Appendix A

makeToTeX

In case your script gives LaTeXproblems, the steps `makeToTex` undertakes are listed here.

---

1. `;\` is replaced with `;\textbackslash`
2. Covert any recipes to a special command to turn the target blue.
3. Strip comments (lines beginning with #).
   (a) At present, it doesn't strip trailing comments. Please remove those yourself.
4. Escape the characters `$`, `%`, and `#`.
5. Add the terminal newline command (\\) to non-blank lines.
6. Replace ^with `\textasciicircum`.
7. Replace double dashes (--) with a macro that stops it from turning them into hyphens (–).
8. Replace single TAB characters with a macro.
9. Replace multiple TAB characters with `\hfill`, a LaTeXcommand that spaces nicely.
10. Leading and trailing quotes are specified differently in LaTeX. Leading quotes are coded with a backtick and trailing with an apostrophe.
11. Extra spaces and multiple blank lines are filtered out.

Please ensure all comments are removed! They are very troublesome to display.

Appendix B

Style Guide

Please refer to this guide when in doubt. It covers typographic and phrasing consistencies.

1. Monospace (use the `\texttt{}` macro) the following things:
   (a) `make`, when referring to the program.
      i. Use the macro `\maken{}`. To remove the trailing space (for example, to use a comma after `make`), call the macro without the braces, `\maken`.
      ii. Never uppercase.
   (b) `bash`
      i. Use the macro `\bashn{}`.
   (c) Any `bash` or `make` command. This is done automatically in the one-liner and multi-line environments. Be sure to monospace commands written in-line.
   (d) Explicit references to a file: e.g., "... open `subjects.txt`."
   (e) Commands run from the command line: e.g., `ls`, `bet`, `flirt`, `qmake`, etc ...
   (f) References to directories: e.g., `/bin/`, `oasis-multisubject-sample/`.
      i. Use the trailing slash for clarity in all cases.
   (g) References to named variables: e.g., `$(SUBJECTS)`, `$<`, `%`, `$PWD`.
   (h) Command-line flags written in-line: e.g., `-n`, `-lart`.
2. Don't monospace:
   (a) **Don't double up on quotes and monospacing.** If both would be used, prefer quotes. This is to avoid a complete monospacing overload.

(b) Abstractions or filetypes: e.g., “Makefiles are usually located . . . ” or “The nifti files . . . ”

(c) Programs typically used through a GUI: e.g. OpenOffice, gimp.

(d) Tool suites, like FSL or AFNI.

(e) The names of operating systems.

(f) References to the grid engine in any form.

i. Use the article with the abbreviation: “the SGE.”

ii. “Grid engine,” not “gridengine.” This is Oracle/SG’s usage, although they capitalize it, which we won’t do.

(g) Subject identifiers.

3. Oxford comma = yes: e.g. “X, Y, and Z,” not “X, Y and Z.”

4. Don’t use em dashes (–) for emphasis. They should appear in pairs the majority of the time.

5. Margin notes and footnotes:

(a) Use margin notes to call out things a new reader would find useful:

i. Define important terms with a [?].

ii. Call out important notes with a [!].

(b) Use footnotes for things that could be completely ignored: advanced features, humorous commentary, etc. . . .

6. LaTeX swallows up two dashes (--). Use `\dd` to get a double dash. You can use a space after this command and LaTeXwill ignore it.

7. **Skull-strip:** Two words, always hyphenated.

8. Headers:

(a) Chapters and Sections Are in Title Case

(b) Subsections in sentence case

(c) Figure captions in sentence case

9. Numerals:

(a) Spell out numbers less than 10, except the ones LaTeX generates.

(b) Always spell out a number at the beginning of a sentence. However, it usually better to recast the sentence to avoid this.

(c) If you must refer to multiple number things, try:

i. Using A, B, C . . . instead: “Process A will overwrite foo.out just in time for process B to use it.”

10. Plurals
    (a) Since we may have to pluralize some weird things: No apostrophe between the stem and the "s," no matter what. Use the same formatting (e.g. monospacing). If possible, recast the sentence.
11. M/makefile
    (a) "a makefile" is an abstract noun referring to any make script (cf. "a shell script"). This should be capitalized when appropriate.
    (b) "the Makefile" is the top-level makefile in a directory, usually named `Makefile` by IBIC conventions. This reference to the file with the same name does not need to be monospaced.
12. Backticks and single quotes in monospaced font:
    (a) Use `\`{}` to get the correct backtick symbol.
    (b) There is currently no support for a straight single quote in the defaulty `\ttfamily` font or Inconsolata at present.
13. Ellipses
    (a) Spaces before and after: "this ... and that."
    (b) Use `\ldots`
14. FreeSurfer