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)

August 25, 2015

Contents

i

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 has been done before: FreeSurfer incorporates make into their pipeline "behind the scenes." We have developed many examples of neuroimaging pipelines in multiple domains that are written using make.

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. However, make allows expression of neuroimaging workflow with only the programming concepts of parallelism and variables. Being able to incorporate programs from many different packages without wrapping them saves time and effort. T. M. has taught several

undergraduates, graduates and staff to use it to develop, debug, and execute pipelines. These students have contributed examples to this manual. 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 commands from Makefiles are shown in an outlined box with no special prompt character:

```
This is 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).

Data for the lab practicals and examples documented in this manual are available from NITRC. We will assume that you have downloaded them into some location on your own machines (e.g. `makepipelines`) and will refer to these files directly. To run the example makefiles, you must set an environment variable, `MAKEPIPELINES` to the location of this directory (see Makefile Examples). If you are at IBIC you may find these examples on `/project_space/makepipelines`.

Throughout this manual we assume a UNIX style environment (e.g., Linux or MacOS) with common neuroimaging packages installed. All examples have been tested on Debian Wheezy using GNU Make 3.81. 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

! You do not type the $ to execute bash commands; it stands for the prompt.

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

3

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 those in Figure 1.1.

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

Figure 1.1: Basic `make` syntax.

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 file.

```
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.

In Figure 1.2 the target file is `s001_T1_skstrip.nii.gz`, which "depends" on `s001_T1.nii.gz`. More specifically, to "depend on" something means that *a)* 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 older 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. If you are not convinced, see **??** for a more detailed explanation of how `make` works.

Additional Resources For Learning About make

The focus of this manual is on structuring neuroimaging projects using make. These additional books and manuals will be extremely helpful for learning about make more generally.

GNU Make Manual, Free Software Foundation. Last updated Oct 5, 2014. Link to manual. Although this manual is for version 4.1, which is a somewhat newer version than we use in our examples, most of the information here is the same. This is an excellent reference for the syntax of make and its functionality.

Managing Projects with GNU Make, Third Edition by Robert Mecklenburg. Nov 2004. Link to open book content. This O'Reilly book covers a lot of basics but in a readable form for someone trying to manage a large scale project. Information about directory structures is probably less relevant for neuroimaging applications.

The GNU Make Book by John Graham-Cumming. April 2015. Link to purchase. This book has an enormous amount of useful advanced information, including details describing differences between different versions of make, many approaches to debugging, and the help system that we use in our examples.

Chapter 2

Running make in Context

In the previous chapter we presented some toy examples. However, in real neuroimaging life one deals with hundreds of subjects with multiple types of scans and many planned analyses. To keep this straight requires some conventions, whether you use make or not. Here we describe some conventions that are useful for managing realistic projects.

File Naming Conventions

File naming conventions are 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, we might include the subject identifier in the final preprocessed resting state data, 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, they 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. Do you really need capital letters?

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 that sounded like fake English words, 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[1] 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`	B_0 Field map phase image for reading session
`RPI001_read_fMRIb0_mag.nii.gz`	B_0 Field map magnitude image for reading session
`RPI001_read_fMRIb0_mag_brain.nii.gz`	Skull stripped B_0 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`	B_0 Field map phase image for writing session
`RPI001_write_fMRIb0_mag.nii.gz`	B_0 Field map magnitude image for writing session
`RPI001_write_fMRIb0_mag_brain.nii.gz`	Skull stripped B_0 magnitude image
`RPI001_DTI.nii.gz`	DTI image
`bvecs.txt`	b-vectors for DTI processing
`bvals.txt`	b-values 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

---

[1]REDCap is the Research Electronic Data Capture web application that we use for storing subject-specific information

were processed. The more you can keep things the same between projects, the more Makefiles 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. We have recreated this directory structure in $MAKEPIPELINES/Udall, without any actual MRI data, so that you can follow the structure. You can also look at **??** for a description of how to set up a similar directory structure for an actual data set.

```
/project_space/Udall
├── bin.............................Holds bash 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 analyses
├── 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 setgid 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 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, because 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.

Notice that in the project directory, we keep scripts that we write to process the data in the `bin` directory. Although `bin` historically is where "binary" executable files are kept, the distinction between executable `bash` scripts and compiled executables is not terribly important. In this example we separate `tcl` scripts and R scripts to make things neater.

Other data files and miscellaneous support files for workflow are stored in the `lib/` subdirectory. Again, 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. This makes it much easier to find things.

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 single-subject (and group) 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). We have found it very useful to document, manage, and merge this data very carefully, including the important QA variables that describe whether or not a subject's scan should be included in analyses.

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).

[?] The setgid bit on a directory ensures that files created here assume the group to which the directory belongs, and not the group to which the person creating them belongs.

Setting up an Analysis Directory

Up until this point we have only seen 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). You can also see this recipe implemented in `Udall`. 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 (see **??**).

This recipe relies on the concept of a "symbolic link." This is a file that points to another file or directory. A symbolic link allows you to access a file or directory using a different pathname. 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 symbolic link, use the `ln` command.

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

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

Target paths can either be absolute or relative, but they must be relative to their new location (in `linkname`). For example, using the directory structure in Figure 2.2, the command to create the link `session1/RC4101` would look like this (run from `Udall/session1`:

```
$ ln -s ../subjects/RC4101/session1 RC4101
```

Defining the basic directory structure

Let us assume the project home (`PROJHOME`) is called `$MAKEPIPELINES/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

```
/project_space/Udall
  ├── subjects ............... All subject data collected in neuroimaging sessions
  │   ├── RC4101
  │   │   ├── session1. .................... Subject-level processing happens here.
  │   │   ├── session2
  │   │   └── session3
  │   └── RC4103
  │       ├── session1
  │       ├── session2
  │       └── session3
  ├── session1/RC4103 .............. Symbolic link to subjects/RC4103/session1.
  ├── session2/RC4101 .............. Symbolic link to subjects/RC4101/session2.
  ├── session2/RC4103.............................................and so on.
  ├── session3/RC4101
  └── session3/RC4103
```

Figure 2.3: A longitudinal analysis directory.

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 2.3 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/session1, PROJHOME/session2, and PROJHOME/session3 whose only purpose is to run make within all the subject-level directories beneath it. Figure 2.4 shows the example session-level makefile (Udall/subjects/makefile_session.mk).

> **?** #: the make comment character.

This Makefile obtains the list of subjects using a wildcard (so it expects to be in a directory where each subject has its own subdirectory). We will create symbolic links to this Makefile later.

The name of the subject directory is declared to be a phony target, so even if it exists, make will try to rebuild it. The recipe to do this is the very last line, which

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

SUBJECTS=$(wildcard RC4???)

.PHONY: all $(SUBJECTS)

all: $(SUBJECTS)

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

Figure 2.4: Session-level Makefile

calls make recursively within each subject directory. It also passes along a TARGET variable. If unspecified, this would be the first target in the subject specific makefile.

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. However, we expect that the only thing that is different about each session is session-specific processing, which can be controlled by a SESSION variable. We can create a single subject-specific makefile and set up symbolic links to it just like we intend to do with the session-level makefile. Figure 2.5 is an example of a subject-level makefile that can be seen at Udall/subjects/makefile_subject.mk.

```
SESSION=$(shell pwd|egrep -o 'session[0-9]'|egrep -o '[0-9]')

subject=$(shell pwd|egrep -o 'RC4[0-9][0-9][0-9]')
test:
        @echo Testing that we are making $(subject) from session
$(SESSION)
```

Figure 2.5: Subject-level makefile

The subject-level makefile defines critical variables, such as the SESSION, which

16

it obtains from the directory path, and the `subject` variable, which it also obtains from the directory path and a regular expression that matches the expected subject name. To set these variables we use the program `egrep`, which allows us to extract a specific pattern from the current working directory.

The only rule is a little dummy rule (because we have no actual data in this test directory) that ensures we have set these variables directly.

Creating links to the session-level makefile

Recall that the session-level makefile needs to be located within the directories `Udall/session1`,`Udall/session2` and `Udall/session3`. We could just copy it there, but then if we modified it in one place we would have to remember to change it everywhere. This would probably cause inconsistencies at some point.

Instead, we create a symbolic link to `makefile_session.mk` from each session directory as follows:

```
cd $PROJHOME/Udall/session1
ln -s ../subjects/makefile_session.mk Makefile
```

Figure 2.6: Creating symbolic links to the session-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 `subjects/RC4101/session1/`, we can type

```
$ ln -s ../../makefile_subject.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/`.

This should generate output similar to that shown in Figure 2.8, showing that `make` goes in to each of the subject directories and creates the `test` target.

```
cd $PROJHOME/Udall/subjects/session2
make TARGET=test
```

Figure 2.7: Making `test`

Figure 2.8: Output of recursive make

Setting Important Variables

`make` uses variables to control a lot of aspects of its own behavior. We describe only a few here; see the GNU Make 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.

Note that `SHELL` is accessed `$(SHELL)`, as are all `make` variables.

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

Figure 2.9: Setting $(SHELL)

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 variable 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 2.10: Running DTIPrep in make

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

A practical way to control for this is to specify the location of the program (if that conveys version information) as in Figure 2.11.

```
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 2.11: 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 a 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`.

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

Figure 2.12: 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. See **??** for an example of how this is done in practice

all

This is the default, the first target in the file. Nothing in this target should require human intervention.

help

21

We use a help system described by John Graham-Cumming, described in chapter 4, to document makefiles. You can approach any makefile by typing make and get a list of documented targets and their line numbers. From the programmer's perspective, it is easy to add documentation to a target, because the call to print help and the target are located right next to each other.

clean

Typically this target is used to remove all generated files (e.g., .o, dependency lists) and clean up the directory to its original state so that one can type "make" again and regenerate. So the idea is that "make clean" will bring you back to square one - starting the pipeline from scratch. Good for disaster recovery.

mostlyclean (or archive)

Same as clean, but does not remove things that are a pain to recreate (e.g., involve hand checking, or time-consuming analysis) or are critical results for publication. We use this because we are perpetually short of disk space, and this helps to clean up.

.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. See **??** for a lesson on secondary targets.

.INTERMEDIATE

This target allows you to specify files that can be automatically deleted after the final targets are created. For example, during resting state preprocessing (**??**) you create many intermediate files during the process (e.g., the output of motion correction,

despiking). These are useful to check for QA purposes but in the end you may not want to keep them. Specifying them as intermediate targets will delete them after completion of the pipeline.

expand this section

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, because 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 for Writing Parallelizable Makefiles

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 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 as intended, not matter what, because the first line of this recipe is not remembered by the second, and brain.volume.txt will be empty.

Instead, write the script as shown in Figure 3.2. Note the ;\ in red connects the two lines.

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

Figure 3.1: A non-functional multi-line makefile

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

Figure 3.2: A now-functional multi-line makefile

You can also use `&&`, a `bash` operator that executes the next command only if the previous command was successful[1]. For example:

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

Figure 3.3: A multi-line makefile using "&&\"

In this instance, `bash` will not attempt to `cat` the file if `fslmaths` or `awk` failed.

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 3.4: 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 processors 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

---

[1] In other words, has exited with an exit status ($?) of "0."

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. Another approach is to create unique temporary names using the `mktemp` program and delete them when you are finished.

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 seperate 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. Examine the rule in Figure 3.5.

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

Figure 3.5: The wrong way to run two long tasks

This sort of 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. A worse problem is that you have specified two targets, `a.out` and `b.out`. This is like reproducing this rule twice. If you run in parallel this rule will fire twice, once to create `a.out` and once to create`b.out`, and you will spend twice as much effort as you need to (but in parallel). So it is better for many reasons to write the rule in two separate lines, as in Figure 3.6.

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

b.out: a.out
        run_b
```

Figure 3.6: The right way to run two long tasks

Executing 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`. You can either specify the number of jobs after `-j` or you can omit the argument and let it use the maximum number of cores available.

It is good 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. For this reason, you may want to specify that `make` should use fewer cores than available so that you can still get good response time on the machine.

Recall that `make` will die when it obtains an error. When running jobs in parallel, `make` can encounter an error that much faster. If one job dies, all the rest will be stopped. This is rarely the behavior you want, because typically each job is independent of the others. To tell make not to give up when the going gets tough, use the `-k` (keep going!) flag.

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

Using the grid engine

Four cores (or even eight or 12 or 24) is nice, but a cluster is even nicer. A cluster gangs together several multi-core machines to act like one. Our cluster environment is the Sun Grid Engine (SGE) which is a batch queuing system. This software layer allows you to submit jobs, queue them up, and farm them out to one of the many machines in the cluster.

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

! You can submit jobs from any machine in a grid engine. In IBIC, broca and pons are the easiest to type.

27

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 (which is not parallelized), 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. Note that we do not use this environment in IBIC.

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) require 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. This way, it is easy to hit ↑ and delete `-n` without having to muck about arrowing over to the flag placed at the beginning of the command. By doing this you could save seconds a day!

```
$ 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.

Use the trace option in `make` 4.1

Debugging is such an issue that the latest version of GNU `make`, version 4.1, has tracing functionality to show you what is going on. Even if your system does not have this installed, you can download it and run it with the `--trace` option. Note that there are some subtle differences between `make` versions. For a thorough desciption of version differences and more debugging options, see .

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 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.

4. 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 3.7 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 3.7: 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.

Chapter 4

Recipes

This section contains recipes for different kinds of small tasks you may wish to accomplish with `make`. Some of these recipes can be seen "in action" in real makefiles in the example data supplement to this manual. These real example makefiles are documented in Makefile Examples. When we refer to these files, we will assume that the path is relative to where you have unpacked these examples in your environment.

Obtaining a List of Subjects

A common thing you may want to do in a Makefile is to set a variable that includes the names of all subjects that you will be processing. Here are some examples of code to set the `SUBJECTS` variable. This first approach (Figure 4.1) keeps a list of subjects in a file called `subject-list.txt`. This is extremely handy for analyses that use only a subset of the subjects: for example, only a treatment group, or a selected subset of subjects from the entire study. For an example of this approach in use, see **??**.

```
SUBJECTS=$(shell cat subject-list.txt)
```

Figure 4.1: Obtaining a list of subjects from a file.

This second approach (Figure 4.2) simply uses the project directory as reference, finding all the subject directories it can, and then assumes if a subject directory exists, so does the data to run this analysis. For an example of this approach in use, see `oasis-longitudinal-sample-small/visit1/Makefile`, described in practical 2.

Both approaches are appropriate for different circumstances. For example, we have used the first approach for a pilot study where several subjects in the pilot had some technical problems with their ASL scans. They were perfectly usable for other purposes but not for the ASL analysis. We have used the second approach for

33

```
SUBJECTS = $(wildcard OAS2_????)
```

Figure 4.2: Obtaining a list of subjects using a wildcard.

keeping up with FreeSurfer analyses of cortical thickness, running FreeSurfer on new subjects as soon as their dicoms are converted to nifti files and their T1 image is placed in their subject directories.

Setting a variable that contains the subject id

It is useful to set a variable that contains the subject identifier, to be used when naming or accessing files that incorporate the subject identifier. We assume that this is in the context of subject-specific processing within a directory. One way to do this robustly (so that it works whether or not you are accessing the subject directory via a symbolic link or from the directory itself) is as shown in Figure 4.3. For an example of this approach in use, see `oasis-longitudinal-sample-small/visit1/Makefile.subdir`, described in Practical 2.

```
subject=$(shell pwd | egrep -o `OAS2_[0-9]*')
```

Figure 4.3: Determining the subject name from the current working directory and a pattern.

By using the command `grep` and a pattern to identify the subject identifier from the current working directory, this method will work no matter where in the path the subject identifier is located.

Using Conditional Statements

Setting a conditional flag

Sometimes we wish to process data only if we have the correct file (or files). By design or by accident, it may be that a certain scan was not acquired or is not of sufficient quality to process. Here, we have created a file `00_NOFLAIR` for each subject data directory that is missing a Fluid Attenuated Inversion Recovery (FLAIR) scan sequence. We set a variable called `HAVEFLAIR` that we can later use to "comment out" commands that would fail miserably if the basis files do not exist (Figure 4.4). To see this in use, see `testsubject/test001/Makefile`, described in practical 3.

We could also just look for the FLAIR image and decide that if it isn't there, we shouldn't try to process it (not throwing an error). With hundreds of subjects, however, it takes a lot of time to cross reference the master spreadsheet to check

```
HAVEFLAIR = $(shell if [ -f 00_NOFLAIR ] ; then echo false; else echo
true; fi)
```

Figure 4.4: Setting a variable to determine whether a FLAIR image has been acquired.

whether the file is missing by design or whether it did not get copied correctly. Making little marker files such as `00_NOFLAIR` (which can contain missing data codes and reasons for the missing data) helps a lot to avoid re-checking.

Another example is a multisite study that occurs at several sites, each with different scanners. We may want to set a site variable that we can use to perform conditional processing (for example, adjusting for different sequence flags). Here, we create a site file called `00_XX`, where the XX is a two letter code for the specific site. The site variable is then set to `00_XX` (Figure 4.5).

```
SITE = $(shell echo 00_??)
```

Figure 4.5: Setting a variable to indicate the study site.

Finally, **??** illustrates how to perform conditional processing based on the existence of specific files. In that example, the presence of certain files triggers different streams of diffusion tensor imaging (DTI) processing. This is a useful strategy for programming a makefile to be portable across different common acquisitions.

Using a conditional flag

Once set, it is possible to test a conditional flag and perform different processing. Having set a conditional flag, it can be used to select rules in your makefile. Here, for example, we use the `SITE` variable to reorient and rename the correct T1 image (which has a different name depending on the different sequence/scanner used at each site).

Conditional execution based on the environment

By default, environment variables set in the shell are passed to `make`. These variables may be overridden within a makefile. In this example (Figure 4.7), used in many of the example data makefiles, we check whether users have set an environment variable `MAKEPIPELINES`. This is used to refer to files within the example directories. If it is undefined it is set to the default IBIC location (so that IBIC users can try out the makefiles without worrying about setting the variable).

```
ifeq ($(SITE),OO_BN)
$(subject)_T1.nii.gz: $(wildcard *SCALP.nii.gz)
        fslreorient2std $< $@
endif

ifeq ($(SITE),OO_MR)
$(subject)_T1.nii.gz: $(wildcard *t1_fl3d_SAG.nii.gz)
        fslreorient2std $< $@
endif
```

Figure 4.6: Testing the site variable to determine the name of the T1 image.

```
ifeq "$(origin MAKEPIPELINES)" "undefined"
MAKEPIPELINES=/project_space/makepipelines
endif
```

Figure 4.7: Testing to see whether an environment variable is undefined

Conditional execution based on the Linux version

In our environment we strive to run the exact same image of the Linux operating system, with all the same software packages installed, on all computers. However, we often migrate workstations to the latest version of the operating system over a period of several months. Different groups time their upgrades to minimally disturb image processing. In one case, there was a particular AFNI program necessary for our resting state analysis that was only installed in the latest OS. So, we check the version of the OS in order to conditionally set the targets that the makefile will build (Figure 4.8).

The command `lsb_release -sc` produces the codeword corresponding to the Linux version installed (here, precise), and this is what we check. The `@` before echo in the second all target indicates not to print the commands themselves to the screen, only the output of the commands, which improves readability.

```
RELEASE=$(shell lsb_release -sc)

ifeq ($(RELEASE),precise)
all: processedrest_medn.nii.gz
else
all:
        @echo "The software to run this pipeline is only installed on"
        @echo "the newest version of the OS, 12.04"
endif
```

Figure 4.8: Testing the Linux version to determine whether to proceed

Part II

Practicals

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 practical designed to be completed as you read the chapter.

Practical 1

Overview of make

Learning objectives: Obtain a conceptual understanding of make, understand the basic syntax of makefiles and pattern rules.

Lecture: The Conceptual Example of 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 (Figure 1.1).

> **!** **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

---

[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

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.

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).

Note that there are no circular references in this graph! (Hence, it is called an "acyclic" graph.) Also note that every arrow points in one direction (to what it depends upon). This is what makes this graph "directed". A graph like this is a very good way of describing a partially ordered sequence of steps. However, drawing 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.
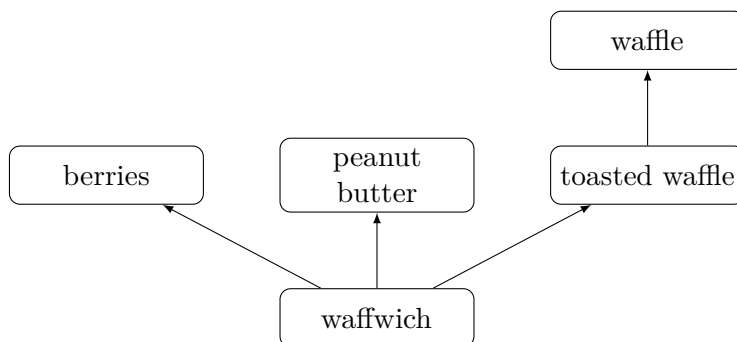
Figure 1.2: How to make a waffwich

```
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

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 search for the targets that it needs to create in whatever order it needs to find them.

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 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 $MAKEPIPELINES/oasis-multisubject-sample/ to your home directory. You can do

this with the following command:

```
$ cp -r $MAKEPIPELINES/oasis-multisubject-sample ~
```

This is a tiny sample of subjects from the OASIS data set (`http://www.oasis-brains.org`). Your task is to skull strip all of these brains. 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
```

The first command reorients the image and the second performs the skull stripping. By default, `make` reads files called "Makefile." According to the manual, GNU looks for `GNUmakefile`, `makefile`, and `Makefile` (in that order) but if you stick to just one convention, you are unlikely to get confused.

Open the makefile (`Makefile`)that came with the directory with an editor. You should see 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 (so that the `fslreorient2std` rule is first). 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, as you can see in Figure 1.5.

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

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

%_T1.nii.gz: %_raw.nii.gz
        fslreorient2std $*_raw.nii.gz $*_T1.nii.gz
```

Figure 1.5: Pattern-matched Makefile

Using the `$*` to stand in for patterns can get visually ugly. One common shortcut is to use the automatic variable `$@` to replace the target, and `$<` for the first dependency.

Using these new variables, our code can now be written as in Figure 1.6.

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

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

Figure 1.6: Pattern-matched Makefile

But try executing `make` now, with just those rules. (If you are completely lost now, look at the file `Makefile.1` to see what your makefile should look like). 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.

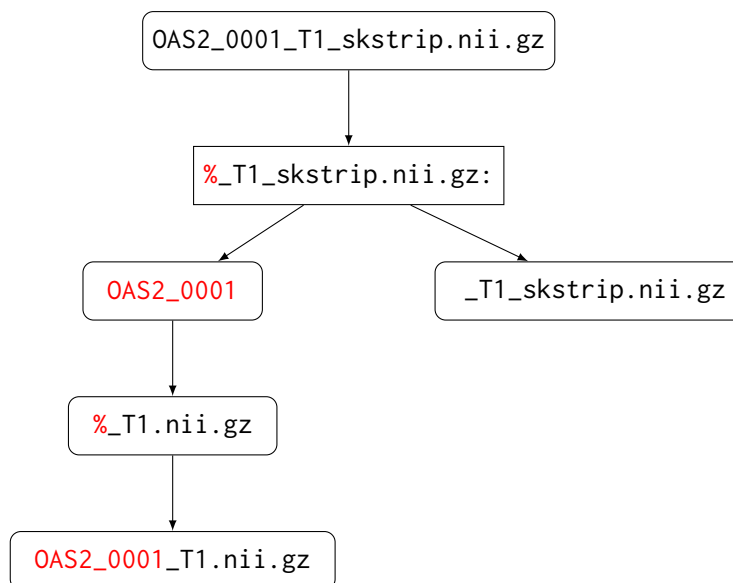Figure 1.7: Pattern-matching as a sieve

Figure 1.7 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.

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)
```

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 original state.

---

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

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

Figure 1.8: make clean

Note that in the recipe, it's totally fine to use wildcards. Depending on how far along you got with your processing before you decide to clean and restart, not every file might have been created. For this reason we typically use the `-f` flag for `rm`. This flag hides error messages from attempting to erase nonexistent files.

The convention of putting a target for `clean` is just a convention; you need to muster the discipline 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, we like the looser convention of `mostlyclean` or `archive` 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).

Do not forget to add `clean`, `mostlyclean` and any other phony targets that you find useful to your list of phony targets:

```
.PHONY: skstrip clean
```

Compare your final makefile to the finished version in `Makefile.2` and you are done with the first practical.

Practical 2

`make` in Context

Learning objectives: Organize a subject directory, run make in parallel, convert a shell script to a makefile.

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. See Running make in Context for examples of good file naming conventions and typical project directory structures.

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`.

These basic issues and the goal of minimizing complexity drive the directory structure described in this practical.

Practical Example: A More Realistic Longitudinal Directory Structure

Organizing Longitudinal Data

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

```
$ cp -r $MAKEPIPELINES/oasis-longitudinal-sample-small/
```

This is a very small subset of the OASIS data set (`http://www.oasis-brains.org`), 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. I have reduced the size of this directory by taking only one of the T1 images for each person, for each visit, and only of 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 as a subdirectory within 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
```

This command symbolically links the directory `../subjects/visit1/OAS2_0001` to the original directory `../subjects/OAS2_0001/visit1`. Now, if you enter `ls -l` into the command line while remaining in the `visit1` directory, you will notice that the subdirectory `OAS2_0001` is symbolically linked, as indicated by an arrow, to the original target directory. You can also check whether a file or directory is linked by using the `ls -F` shorthand command, which indicates symbolic links with an symbol. The command `ls -L` dereferences the symbolic links so that you can view information for the original target file or directory itself.

You can also create symbolic links in bulk. To do so, remove the link that you have just created and use the program in the `visit1` directory (`makelinks`) to create all the subject links for visit1.

Take a look at the `makelinks` script, reprinted below. This script loops through all of the subject directories that have a first visit (visit1). It extracts the subject identifier from the directory name (`$i`) by using the `egrep` command to find the bit of the name that matches the subject identifier pattern (`OAS2_[0-9]`). With this information, it can link the subject name to the directory.

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

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

```
#!/bin/bash
for i in ../subjects/*/visit1
do
        subject=`echo $i| egrep -o 'OAS2_[0-9]*'`
        ln -s $i $subject
done
```

Figure 2.1: Script to create symbolic links within a longitudinal directory structure

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:

```
$ make Makefile
```

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 (`Makefile.subdir`). Go into a subject directory and run `make`.

By now, you might be getting really tired of seeing the same `fslreorient2std` and `bet` commands. `make`, by default, 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 2.2: 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 2.3: 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 explicit 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 perform 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 familiar 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.[1] 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 (see Desikan et al., 2006; Fischl et al., 2004b,a) 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 **??**.

---

[1]They are: `amygdala`, `broca`, `dentate`, `homunculus`, `pole`, `pons`, `precuneus`, and `triangularis`.

Practical 3

Getting Down and Dirty with `make`

Practical Example: Running QMON

As was mentioned in the previous practicum, we can determine how many processing units we have on our computer by calling up the system monitor. To do so, type `gkrellm` into the command line. As you will see, this software also allows us to monitor the status of CPUs, memory usage, and disk space.

   If the number of jobs that you need to run exceeds the number of cores that you have, submitting jobs to the grid engine (explained in chapter 2) might be a good idea. But how do we check whether the machines in the cluster we are interested in are available for batch processing? The `qmon` monitor is a graphical user interface that comes with the Sun Grid Engine software. It gives us the ability to track the status of machines in a cluster (and more besides, but you do not have to worry about that for now).

   In order to run `qmon`, you need to be logged in to one of the cluster workstations. This can be done by initiating a secure shell session, as follows:

```
$ ssh -X <username>@broca.ibic.washington.edu
```

   Note that the `-X` flag is necessary to allow you to log in to the remote machine (e.g., `broca`) but forward graphical output to the computer that is managing your graphical display. Once you have answered the password prompt, type `qmon` into the command line. Of interest to us is the first button in the panel: "Job Control." When you click on this, you will be able to see the number of jobs running on the cluster and the number of jobs in the queue, along with the username of the person who submitted those jobs. Ideally, you want to submit your jobs to the cluster when

! A list of the clusters and machines available for use at IBIC can be found on the wiki.

55

the cluster is free.

However, this scenario does not necessarily always present itself. If you use `qmake`, you will get an error by default if the grid engine is busy. You can get around this by using the `-now no` options to `qmake`, which will cause it to wait until the grid engine is free. For more information on how to use the grid engine to run a parallel `make` workflow, refer to the manual, or Practical 2 where we covered the use of `qmake`.

If you are primarily employing FSL tools with `make`, you can use `FSLPARALLEL` to submit jobs for parallel processing. Before running `make`, set up the following environment variable in your shell:

```
$ export FSLPARALLEL=true
```

Once you have done this, you may go ahead and use `make` *without parallelization* to submit jobs to the grid engine. This calls up a script called `fsl_sub` that FSL uses to schedule and submit jobs in parallel in the way that some of its cluster aware tools, such as MELODIC, are capable of doing. Jobs parallelized using `FSLPARALLEL` will automatically be queued under "Pending Jobs" if the cluster workstations are busy processing other jobs.

Practical Example: A Test Subject

If you have not already, copy `testsubject` to your home directory as follows:

```
$ cp -r $MAKEPIPELINES/testsubject ~
```

This copy will take a few minutes, depending on what kind of file systems you are copying from and to, because this directory is fairly large.

Go into this directory and look at its contents.

```
$ cd testsubject; ls
```

You will see three subdirectories: `bin/`, `lib/` and `test001/`. As I am sure you have noticed, this is a much simplified structure as compared to the longitudinal directory structure that we looked at last week. For this example, I have eliminated the individual subject subdirectories (because there is only one) and the symbolic links (because there is only one timepoint). This will allow us to focus on the makefile structure for multiple types of analyses.

While there is no strict convention dictating the manner in which you should organize your project directory, it is good practice to set up a directory tree structure that clearly separates the various parts of your projects from one another. In most

cases, a `bin/` directory stores executables and scripts that have been written for the project. The `lib/` directory, on the other hand, typically holds makefiles used for the project (although in some IBIC project directories, a `templates/` directory is used for this purpose instead). The `testsubject/` directory in this example stores a single subject's processed data.

Now, `cd` into the `test001/` directory and have a look at the top-level makefile. In the first section of the makefile (prior to the listing of the various targets and dependencies), I have set two conditionals to check whether we have a FLAIR and T1 image. While this may not useful when we are dealing with a single subject, you will find that it makes processing a large number of subjects much easier. What the two conditionals are doing is that they are asking shell to go into a subject directory and look for files called '`00_NOFLAIR`' and '`00_NOT1`'. For this to work, you will need to create these files beforehand in the subject directories of subjects whom you know do not have these images. This is relatively easy to do and makes processing less of a pain. It will also help you to debug `make`, which is likely to break when it finds that it is missing dependencies that do not exist.

If a file '`00_NOT1`' is found, a FALSE boolean is returned, and `make` will not make targets that require a T1 image as a dependency. This is specified in the next conditional:

```
❶  ifeq ($(HAVET1),true)
   all: $(call print-help,all,Do skull stripping, etiv, HC volume
       calculation) T1_skstrip.nii.gz first etiv
❷  else
   all:  $(call print-help,all,Do skull stripping, etiv, HC volume
       calculation)
           @echo "Subject␣is␣missing␣T1␣-␣nothing␣to␣do␣here."
   endif
```

❶ If the variable HAVET1 returns TRUE, `make` will make the targets listed in `all`.

❷ If the variable HAVE1 returns FALSE, `make` will echo to your screen that the subject is missing a T1 image.

Test out this conditional by making a 00_NOT1 file in the `testsubject` directory like so:

```
$ touch 00_NOT1; make -n
```

You will find that `make` will tell you that the subject is missing a T1 image and that there is nothing to do.

Remove `00_NOT1` and try running `make` again. You can see that `make` is now running our edited version of the first script, to conform to the Enigma protocol (see

Stein et al., 2012). The protocol may be viewed online at: http://enigma.ini.usc.edu/protocols/imaging-protocols/hippocampal-segmentation-through-first/

Since running FIRST will take a little time, start it now and place it in the background:

```
$ make first &
```

Estimated total intracranial volume

Let's now turn our attention to the calculation of estimated total intracranial volume (etiv). This can be estimated by running FreeSurfer, but FreeSurfer takes a long time to run and had some difficulties with automatic processing of many of the subjects in the study that this mMkefile was modified from.

One technique for estimating the total intracranial volume is simply to linearly register the brain to standard space (see http://enigma.ini.usc.edu). The transformation matrix describes the scaling that is necessary to align the brain to the standard brain. The inverse of the determinant of this transformation matrix is a scaling factor that can be multiplied by the size of the template (e.g., the volume of the standard space brain) to obtain an actual volume.

If you type make -n etiv, the first command that will be executed is to skull-strip the brain. This uses the -B flag, which does bias field correction and is rather slow. However, you can see that the T1_skstrip file is actually sitting in that directory. So why does it want to recreate it?

Look at the dates at which the files in your testsubject/ directory were last modified. Now look at the dates in the master directory project_space. Copying, using cp -r, did not preserve the dates! Everything got the new date that it was created. Up until now, we have created everything in the directories in which we are working, from scratch.

You can avoid this by copying the files in some way that preserves the dates. For example, you can type:

```
$ cp -r --preserve=time-stamps
```

Alternatively, you can use tar to create an archive, copy the archive, and unarchive it. For example, to tar up a directory called mydirectory into an archive called archive.tar in your home directory, type:

```
$ tar cvf ~/archive.tar mydirectory
```

To untar this archive that you have just created, type:

```
$ tar xvf ~/archive.tar
```

How can we "trick" make into not recreating this skull-stripped file? An easy way here is simply to touch the skull-stripped file, like so:

```
$ touch T1_skstrip.nii.gz
```

Once you have done this, go ahead and make etiv:

```
$ make etiv
```

This time, make will not try to recreate the skull-stripped file, because touching it makes it newer than the image from which it is created. Instead, it will run FLIRT on the T1 skull-stripped image. FLIRT is a FSL tool for linear registration, and is used to align a brain to standard space by transforming the skull-stripped brain image using an affine matrix. Following this, make will extract the intracranial volme from the output matrix generated by FLIRT and print this to a comma-separated value (csv) file.

To look at the contents of the file eTIV.csv, type:

```
$ cat eTIV.csv
```

Often, there are statistics that need to be collected for each subject. One way to go about this is to write a program that gathers the correct statistics for each individual, and then decided whether or not you want to call the program from within make. I find, however, that often I want to know what those numbers are while I am examining the output for a single individual, such as now. Thus, I normally create a little comma separated value (csv) file that contains the subject ID and whatever statistics I am interested in looking at. This way, I do not need to remember the commands to obtain the statistics I want (e.g. what NIfTI file should I be looking at? What command do I need to use to extract the right value?). This will also allow me to go to the "top level" directory that contains all the subjects within it and use cat to gather all the files into a single file for data analysis later on.

As it is, I happen to know that this subject has a big head. And yet, the etiv file tells us that the scaling factor is approximately 0.6. This is, of course, a contrived example. But have a look at the skull-stripped image to see what went wrong. You will see that too much of the brain was removed, which may have occured because there is a lot of neck in the T1 image. As such, you will need to correct the skull-strip.

One way to do this is by specifying the center of the brain:

```
$ bet T1.nii.gz T1_skstrip.nii.gz -c 79,120,156
```

This improves the situation but it is still not great. Moreover, in a large study (suppose you are looking at 1,000 brains), even a modest 10% failure in running a skull-stripping program like FSL's BET would mean having to handle 100 brains by hand.

Look back at the makefile. You can see that there are two alternative rules for obtaining the skull-stripped brain. My favorite way is to obtain a brain mask from FreeSurfer. It takes many hours to run, but if you have run it, why not just use it?

But for now, consider the program robex. This is a robust skull stripping program (Iglesias et al., 2011) that we can run.

```
$ make robex
```

Now check to see that the skull-stripping is better. Use whatever skull-strip you think best represents the brain volume (so that the scaling will be improved, at least – even if it is not perfect) and type:

```
$ make etiv
```

You should see that the scaling factor is higher than it was. Note that it is still less than one. This is because the MNI template has "drifted" and is larger than most actual brains.

Hippocampal volumes

By now, you should have a file called hippo.csv that contains the hippocampal volumes from the Enigma protocol. However, we have not done the quality checking step that is recommended by the protocol. In this example, we will add the step to create a webpage that displays the registrations to the makefile.

This command is as follows:

```
$ ${FSLDIR}/bin/slicesdir -p ${FSLDIR}/data/standard/MNI152_T1_1mm.nii.gz *_to_-
std_sub.nii.gz
```

Run it and you can see the output. Then, delete the directory, slicesdir, that you just created. Now, as an optional exercise, create a rule called firstqa that will execute this command and make sure that it is part of the target to make first.

Implementing a Resting State Pipeline: Converting a Shell Script into a Makefile

In the `testsubject/` directory, you will find a bash script called `resting-script`. The goal for this part of the practicum is to convert that shell script into a makefile.

A few things that are worth taking note of:

1. Note that the shell script assumes it will never be re-run again and simply calls `mkdir xfm_dir`. If this directory already exists, however, you will get an error. To avoid this, add a `-p` flag to the command `mkdir`. This allows it to create the specified directory if it does not exist, but to not give an error otherwise. This is especially useful in makefiles that create deep directory structures.

2. Many FSL commands (and other neuroimaging commands) automatically add file extensions such as `.nii.gz` to the input files if you do not append them. However, a makefile will not do this. Therefore, when converting commands from a script, you need to make sure that you don't take such shortcuts in a makefile. A good rule of thumb is to specify all extensions in your targets.

3. The decision whether to make something a multi-line recipe, a shell script of its own, or a short recipe is a little arbitrary. If I think that I want to do something often on a command line, I make it into a short shell script and call it from a makefile. In this example, despiking with AFNI is short enough that it can fit into a recipe.

4. However, anything complicated that constitutes a pipeline in itself is better coded from the beginning as a makefile. Not only does this allow parallelism and fault tolerance at a fine granularity, but you can see that it is easy to ask someone for commands to do registrations or obtain a skull-strip from FreeSurfer. Makefile commands can easily be shared, updated, and replaced with the latest methods. Instead of using FSL FLIRT for registration, try editing your makefile such that registration is done with `epi_reg` instead.

5. Note that `3dDespike` is an AFNI program that has been compiled to use all the cores in a machine. You will want to turn that off because you will do the parallelization yourself; this can be achieved by entering the following line in a makefile: `export OMP_NUM_THREADS=1`

To go about translating this script into a makefile, it may help to do things one step at a time. I recommend that you write and *test* one rule at a time for each step in the workflow in the command line. This will help with debugging. Create a rule, test it with the `-n` flag (which echoes to your shell what `make` intends to do to make the target), then run `make` without the flag to create your specific target. Then delete all the things that you have created and add the next rule. This will allow you to see if things chain together. One of the worst things you can do (if you

⚠ To create a makefile in emacs, you need to ensure that you are in "make" mode. Type esc x makefile_-gmake_mod into the emacs console to change the mode you are operating in.

61

are not very comfortable with `make`) is to write your entire makefile and then try to figure out why it does not work as expected.

For the purposes of this exercise, create a makefile named `Makefile.rest` in order to distinguish it from the top-level makefile that already exists.

To ensure that you are calling the right makefile in the command line, you should do the following:

```
$ make -f Makefile.rest nameoftarget
```

The `-f` flag lets you specify the name of your makefile where the rule to creating your target of interest exists. Recall that by default `make` will look for commands in files called `Makefile` or `makefile`. Because `Makefile.rest` has a different name, you need to tell `make` about it.

Now, let us have a look at the first part of `resting-script`.

If you need help with identifying your target and dependency from the shell commands alone, you should look at the usage of the command in your terminal window. E.g. for FLIRT, simply type `flirt`. You will be able to understand what the flags mean, and from there find out what your input and output is.

```
#!/bin/bash

mkdir xfm_dir

#Create transformation matrices

#Making the 2mm MNI --> ANAT matrix
echo "Registering the T1 brain to MNI"
flirt -in T1_skstrip.nii.gz -ref /usr/share/fsl/data/standard/MNI152_T1_-
2mm_brain.nii.gz -omat xfm_dir/T1_to_MNI.mat
echo "Preprocessing RESTING scans..."

### MOTION CORRECTION
echo "Begin motion correction for rest.nii.gz"
mcflirt -plots rest_mc.par -in rest.nii.gz -out rest_mc -rmsrel -rmsabs
```

Figure 3.1: `resting-script` file

The script from Figure 3.1 would look like the rules below in a makefile:

```
❶   PROJECT_HOME=/mnt/home/username/testsubject/testsubject
❷   SHELL=/bin/bash

❸   .PHONY = all clean

❹   all: xfm_dir/T1_to_MNI.mat rest_mc

❺   #Create transformation matrices and making 2mm MNI --> ANAT matrix
    xfm_dir/T1_to_MNI.mat: T1_skstrip.nii.gz
            mkdir -p xfm_dir ;\
            flirt -in T1_skstrip.nii.gz -ref /usr/share/fsl/data/
                standard/MNI152_T1_2mm_brain.nii.gz -omat xfm_dir/
                T1_to_MNI.mat

❻   #Motion Correction
    rest_mc.nii.gz: rest.nii.gz
            mcflirt -plots rest_mc.par -in rest.nii.gz -out rest_mc -
                rmsrel -rmsabs
```

❶ When creating your makefile, you must specify your project home.

❷ Be sure to tell make to use bash by setting the default shell.

❸ Define your phony variable which is a special target used to tell make which targets are not real files. clean is a common rule to remove unnecessary files, while all is a common rule to make all your other targets.

❹ Define all to include all your targets.

❺ Format your commands to take the form of "rules" to include a target and dependency. Every command following your [target:dependency] definition should be entered on a new line beginning with a TAB character. The target is your output and the dependency is your input (you can, however, have several targets and dependencies listed under a single rule). Remember to also use ;\ at the end of each like in order to tell make to read each line separately, as in the example for creating transformation matrices. Here, we are using FSL FLIRT to create an output matrix called T1_to_MNI.mat.

❻ For motion correction, our expected output (i.e. target) is rest_mc.nii.gz, and our input (i.e. dependency) is rest.nii.gz. The last line tells make to run FSL mcflirt on rest.nii.gz.

Practical 4

Advanced Topics & Quality Assurance with R Markdown

Creating a `make` help system

It is sometimes useful to know what a makefile does without having to look through the entire makefile itself. One way of getting that information at a glance is to create a `make` help system that prints out the targets and a short summary explaining what each target does.

To go about creating this help system, you need to create a separate "help" makefile that tells `make` what to do when you call for help. Before we expand on that, however, let us take a look at the main makefile in the `testsubject` directory copied over from `/project_space/`.

```
$ cd  /testsubject/test001
```

Let the variable $(PROJECT_HOME) represent the pathway to your `testsubject` directory. You can set this variable in top-level makefile.

```
PROJECT_HOME=/yourhomedirectory/testsubject
```

Open up `Makefile` in an editor. You will notice that one of first few lines at the top of the file asks `make` to include a makefile called `help_system.mk`; this is the "help" makefile that we alluded to earlier. The `include` directive tells `make` to read other makefiles before continuing with the execution of other things in the current makefile.

As you scroll down, you will see that some of the targets are followed by a `call` command before their dependencies. For instance, look at the line:

```
robex:  $(call print-help,robex,Alternate skull stripping with ROBEX)
T1.nii.gz
```

This tells `make` to return the target `robex` along with its description when `print-help` is called (`print-help` is defined in `help_system.mk`, as we will see later). Other targets such as `freesurferskstrip` and `etiv` also make calls to `print-help`.

Now, have a look at `$(PROJECT_HOME)/lib/makefiles/help_system.mk`.

```
help:  ; @echo $(if $(need-help),,Type `$(MAKE)$(dash-f) help´ to get help)
```

This line tells `make` to echo the fellowing when you type `make` into the command line.

```
$ make
Type 'make help' to get help
```

Figure 4.1: `make`'s Help System

The variable `need-help` is set such that `make` will filter for the word `help` in your command line entry to decide whether or not you need help. If so, the variable `print-help` will be called upon – and this will result in `make` printing the name of your targets and their descriptions to your shell. Depending on whether or not this will be helpful for you, you may want to copy `help_system.mk` to your own project directory and include it in your top-level Makefile.

We will not reproduce the details of how `help_system.mk` works in this practical. For an excellent description of this simple but useful system, please refer to page 181 of John Graham-Cumming's GNU Make Book, or to his blog posting.

Makefile Directory structure

Oftentimes, there are several people working on processing different types of data from a project and this calls for several makefiles. Or, you may wish to split up a processing pipeline into various parts as we have in our Makefile example. In the main makefile `Makefile` from the `$(PROJECT_HOME)/test001` directory – and this was mentioned earlier – you should have noticed that several other makefiles were included.

```
 include $(PROJECT_HOME)/lib/makefiles/help_system.mk
include $(PROJECT_HOME)/lib/makefiles/resting.mk
include $(PROJECT_HOME)/lib/makefiles/xfm.mk
include $(PROJECT_HOME)/lib/makefiles/fcconnectivity.mk
include $(PROJECT_HOME)/lib/makefiles/QA.mk
```

Here, we see that there are makefiles for:

1. creating the `make help_system`.

2. processing resting-state data called `resting.mk` (this is documented in detail in **??**).

3. obtaining transformation matrices for registrations called `xfm.mk` (see **??** for documentation).

4. running seed-based connectivity analysis called `fcconnectivity.mk` (see **??** for documentation).

5. creating QA reports called `QA.mk` (see **??** for documentation).

For a full description of the makefiles (excluding `help_system.mk`), we encourage you to refer to their documentations in the last section of the manual. The documentations will guide you through each makefile and explain the syntax, scripts and programs used in each of them.

Note that makefiles should be stored in the `lib` directory as was done for this example. This is good practice and helps keep your directories organized and less cluttered. The importance of creating a tidy directory tree structure has been continually emphasized throughout this course. Refer to practical 3 or chapter 2 of the manual for a more thorough discussion of the directory structure common to most IBIC projects.

The `clean` target

In `make`, the `clean` target serves to delete all unnecessary files that were created in the process of running `make`. Remember that storage is limited! Cleaning up your project directory will make space for future projects that you will be working on. The target itself is usually specified last in a typical `make` recipe, and must be included in your list of `.PHONY` targets if you choose to run it.

Now, go to your `$(PROJECT_HOME)/test001/` directory and open `Makefile` in an editor. Scroll all the way down. You will see both an `archive` and `clean` target. `archive` is a target intended to clean up the directory for archiving after a paper has been accepted. The purpose of this target is to retain important results but remove the partial products. Obviously, what you may define as "important results" depend upon the kinds of questions that might come up later.

The `clean` target in this makefile includes `qaclean` and `restingclean` which are themselves targets in other makefiles in the `lib/` directory. Typically, files that are "easy" to make and are no longer necessary can be removed. Other files, such as those generated by FreeSurfer's `recon-all`, are not as easily remade. Other examples of files that you may not want to remove include hand-edited files, and end stage products of quantitative processing pipelines. You should therefore think carefully about what you want to remove and what you want to keep.

Defining Macros

– Kelly's example –

Incorporating R Markdown into `make`

Quality assurance (QA) is an important step in a neuroimaging analyses pipeline. At IBIC, data is typically preprocessed and checked for quality before proceeding with further analyses. This can be done both qualitatively and quantitatively. Some of the common aspects of data that are looked at include motion outliers, quality of brain registration/normalization to a subjects-specific or standard template, and whether brain segmentation has been performed correctly.

While neuroimaging packages usually include QA tools of their own (such as FSL's FEAT report), built-in QA tools have limited application when you are using more than one neuroimaging package to process your data or if you want to view your data in a non-traditional way.

Know that you may certainly create QA measures for each subject in your project and inspect these one-by-one in a previewer program – but this may prove inconvenient and time-consuming. Since there usually are several things that have to be inspected during a quality assurance procedure, it is incredibly helpful to have images and data parameters or statistics displayed on a single page for each subject in a project. Alternatively, you may choose to look at a single aspect of your data and would like to concatenate a QA measure from all of your subjects into one report. This is where R Markdown comes in.

R Markdown is an authoring format that allows us to generate reports from R (http://rmarkdown.rstudio.com). It is extremely versatile in that it can read HTML and bash code to generate HTML and PDF documents, or even interactive web applications.

Although QA measures can be fed into PDF documents or HTML pages, we at IBIC prefer to use the HTML format for a couple of reasons. For one, a HTML report gives us the ability to look at moving GIF images, which are useful when we want to view the collected brain scans as a time series. In addition, it circumvents

the issue of pagination and allows us scroll seamlessly through all our images and statistics in a single page. PDFs, however, tend to be more cumbersome to use as images may be awkwardly split between pages and cause page breaks if they are not properly sized.

Go to the directory `$(PROJECT_HOME)/freesurfer/QA` and open up `QA_check.html` with your internet browser.

```
$ iceweasel QA_check.html
```

Clicking on the link `testsubject` will redirect you to a page that holds the images generated by FreeSurfer. Here, you can see the segmented and parcellated brain of the subject, along with images showing the surface curvature of the brain. This page is automatically created by FreeSurfer's recon-all.

We can also create our own QA images to, say, check whether a T1 brain has been properly registered to a brain in standard space. These images can be specified as targets in your makefile like so:

```
QA/images/T1_to_mni_deformed.png: xfm_dir/T1_to_mni_deformed.nii.gz
$(FSLpath)/data/standard/MNI152_T1_2mm_brain.nii.gz
        mkdir -p QA/images ;\
        $(FSLpath)/bin/slicer $(word 1,$^) $(word 2,$^) -s 2 -x 0.35
sla.png -y 0.35 slb.png -z 0.35 slc.png ;\
        pngappend sla.png + slb.png + slc.png
QA/images/intermediate1.png ;\
        $(FSLpath)/bin/slicer $(word 2,$^) $(word 1,$^) -s 2 -x 0.35
sld.png -y 0.35 sle.png -z 0.35 slf.png ;\
        pngappend sld.png + sle.png + slf.png
QA/images/intermediate2.png ;\
        pngappend QA/images/intermediate1.png -
QA/images/intermediate2.png $@ ;\
        rm -f sl?.png QA/images/intermediate?.png
```

Figure 4.2: Using FSL's slicer to create a registration image

Here, FSL slicer is used to create 2D images from 3D images. To get a better understanding of how slicer is used, simply type slicer into the command line. The FSL page http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/Miscvis also provides a brief description of what slicer does.

In the context of the above example, slicer first overlays the T1_to_mni_deformed image on top of the MNI152 brain. Subsequently, it creates several 2D images named sl?.png that come from sagittal, coronal or axial slices. This is indicated by the -x/y/z flags used followed by the name of the image generated. Note that these files are temporary! Once pngappend has been used to paste these images next to each other to generate a single image, the temporary files are removed. If you are using FSL slicer for more than one instance in a makefile, the temporary files created may cause make to crash if they are given the same names and are put into the same directory when make is run in parallel. Make sure that the names of your outputs from FSL slicer do not overlap each other.

Given that the code used to generate QA images can be very messy and ugly, you may want to create a bash script to create those images and call that from your QA makefile instead. This also avoids potential problems that may arise when many similarly named temporary files are generated that may cause make to continually overwrite these files when using a tool like FSL slicer in parallel. A wonderful example of such a script can be found at $MAKEPIPELINES/tapping/bin/sliceappend.sh (credit goes to Matt Peverill!).

Once we have created all the QA images we want to include in our HTML report, we need to create a .Rmd file that tells R Markdown where to find these images.

Now let us have a look at a subsection of a `.Rmd` file on the next page. The file is `$(PROJECT_HOME)/lib/Rmd/fMRI.Rmd`.

```
---
title:  Quality Assurance of Preprocessing for TASK for ID SUBJECT
output:
html_document:
keep_md:  no
toc:  yes
force_captions:  TRUE
---
#T1 Skull-Strip

<img src="/project_space/makepipelines/testsubject/test001/QA/images/T1_-
skstrip.gif" width="500px" />


#Parameters

##Time Series Difference Analysis

<img src="/project_space/makepipelines/testsubject/test001/QA/images/TASK_-
tsdiffana.gif" width="750px" />


# MOTION

##Motion Statistics

```{r engine='bash', echo=FALSE}
val=`cat /project_space/makepipelines/testsubject/test001/rest_dir/TASK_-
mean_abs_disp.txt`
echo "Mean Absolute Displacement :  $val mm"
val=`cat /project_space/makepipelines/testsubject/test001/rest_dir/TASK_-
mean_rel_disp.txt`
echo "Mean Relative Displacement :  $val mm"
```
```

Figure 4.3: Example of a R Markdown File

A `.Rmd` file should always begin with a demarcated section listing the title of the report and the type of file to be created. We see that the output here is a HTML document. `toc` refers to table of contents.

In R Markdown, the size of headers depend on the number of #(hash) symbols used before the name of a header. This is akin to how the titles of major sections of a text are rendered in the largest text size, and subsections along with sub-subsections are printed in smaller-size text. A double hash symbol before a title will mean that the title will be treated as a "sub-section" and be printed in smaller size than the main headers. As the number of hash symbols used before a title increases, the text becomes smaller.

Notice that HTML syntax is used to insert images into the report. Heights and widths of images can be specified as well.

Assuming we want to print out some information about the subject into the report, we can use `bash` within a R Markdown file to `grep` for the stuff we need or `cat` a value from a file. In the example above, we open the bash script portion of the file with three backticks and a curly bracket telling R that the following text is written in `bash` code. Here, we are asking R to `cat` the values from two text files that gives us the mean absolute and mean relative displacement in mm (see `cat` help if you are not familiar with its usage). These will then be printed out into our HTML report.

Once we have a `.Rmd` file ready, we want to tell `make` to create the HTML report for us. By doing so, we can parallelize the generation of QA reports. We do this by inserting a target such as the following into our QA makefile (which is fully documented in **??**):

```
QA/rest_Preprocessing.html: $(PROJECT_HOME)/lib/R/fMRI.Rmd TSNR
MotionGraphs SkullstripQA
        sed -e 's/SUBJECT/$(subject)/g' -e 's/TASK/rest/g' $(word
1,$^) > QA/rest_Preprocessing.Rmd ;\
        R -e 'library("rmarkdown");rmarkdown::render("QA/rest_-
Preprocessing.Rmd")'
```
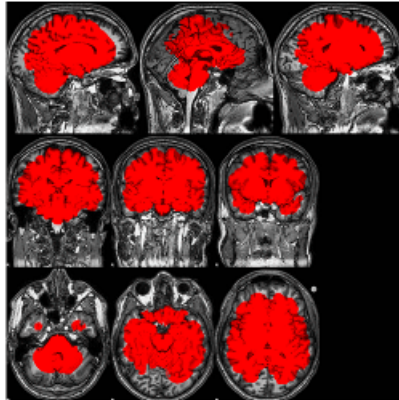
Figure 4.4: Reading a .Rmd file in `make`

The target is your HTML file, and the dependencies are your `.Rmd` file and your images or whatever else you decide to include in your report. The output is dependent on the `.Rmd` file to trigger to regenerate when the source has been changed. We then use `sed` to replace instances of the string "SUBJECT" in the R Markdown file with the actual subject ID. The subject ID variable `$subject` should have been set at the initial portion of your makefile.

`make` will then call R to load the package "rmarkdown" so that it can read R Markdown, and proceed to generate/render your report, which will look something like this:
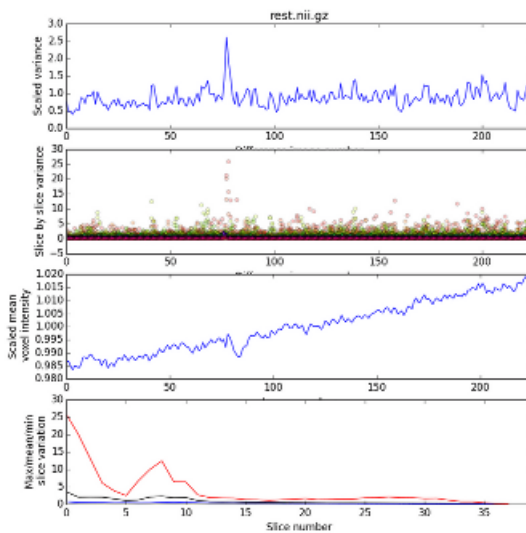
Figure 4.5: QA report in R Markdown

The report in its entirety may be viewed in your Internet browser and can be found at `$(PROJECT_HOME)/test001/QA/rest_Preprocessing_example.html`.

And that's it!

Part III

Examples

Example 1

Makefile Examples

This chapter includes examples of Makefiles included in the practical data. We assume that you have unzipped the practical data into a directory somewhere in your local environment. Once you have done that, you should set an environment variable called `MAKEPIPELINES` to refer to this directory. In the command below we assume you have placed it in your home directory:

```
$ export MAKEPIPELINES=~/makepipelines
```

The examples included below will reference this environment variable as necessary to find the correct files.

Bibliography

Desikan, R. S., Ségonne, F., Fischl, B., Quinn, B. T., Dickerson, B. C., Blacker, D., Buckner, R. L., Dale, A. M., Maguire, R. P., Hyman, B. T., Albert, M. S., and Killiany, R. J. (2006). An automated labeling system for subdividing the human cerebral cortex on mri scans into gyral based regions of interest. *NeuroImage*, 31(3):968 – 980.

Fischl, B., Salat, D. H., van der Kouwe, A. J., Makris, N., Ségonne, F., Quinn, B. T., and Dale, A. M. (2004a). Sequence-independent segmentation of magnetic resonance images. *NeuroImage*, 23(Supplement 1):S69 – S84. Mathematics in Brain Imaging.

Fischl, B., van der Kouwe, A., Destrieux, C., Halgren, E., Ségonne, F., Salat, D. H., Busa, E., Seidman, L. J., Goldstein, J., Kennedy, D., Caviness, V., Makris, N., Rosen, B., and Dale, A. M. (2004b). Automatically Parcellating the Human Cerebral Cortex. *Cerebral Cortex*, 14(1):11–22.

Iglesias, J., Liu, C., Thompson, P., and Zu, T. (2011). Robust brain extraction across datasets and comparison with publicly available methods. *IEEE Transactions on Medical Imaging*, 30(9):1617–1634.

Stein, J. L., Medland, S. E., Vasquez, A. A., Hibar, D. P., Senstad, R. E., Winkler, A. M., Toro, R., Appel, K., et al. (2012). Identification of common variants associated with human hippocampal and intracranial volumes. *Nature Genetics*, 44(5):552–561.