Lecture: Organizing Subject Directories

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

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

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

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

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

Practical Example: A More Realistic Longitudinal Directory Structure

Organizing longitudinal data

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

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

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

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

```
$ find .
```

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

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

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

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

```
$ mkdir visit1
```

Now `cd` into it:

```
$ cd visit1
```

What we want to do is create the symbolic links for each subject's first visit here.

You can do one link by hand:

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

You can also do this in bulk. Remove this link that you created and use the program in that directory (`makelinks`) to create all the subject links for visit1. The current directory won't be in your PATH, so make sure to call it with `./makelinks`.

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

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

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 the subject target is 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 makefiles
```

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

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

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

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

Figure 0.1: The `@` character hides commands

Now go find the same subject via the subject subtree:

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

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

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

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

Figure 0.2: Printing out the value of variables

Now if you type:

```
$ make print-subject
```

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

Running `make` over multiple subject directories

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

From within the `visit1/` directory, type:

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

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

We can do this work in parallel. Bring up a system monitor (`gkrellm` is installed on the IBIC systems). See how many processing units you have and how busy they are. Note these numbers and get 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. Once there, you can use the command:

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

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

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

Running FreeSurfer

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