Lecture/Practicum 1. Overview of Make

1. Workflow systems.
    a. Reproducibility. But a shell script will do that.
    b. Parallelism and fault tolerance. To do these, you must have more information than is expressible in a makefile.
2. Waffwich example in pseudocode.

```
For person in a b c
Do
Toast waffle
Spread waffle with peanut butter
Arrange berries neatly in squares on half the waffle
Cut in half and fold
Done
```

3. Note that this "script" tells you very neatly what to do. However, it enforces a linear ordering on the steps. If you had lots of sandwich making resources that you could use, you would not know from this description of the steps that you did not have to do person a's sandwich first, b's sandwich second, and c's third. You wouldn't know that you could in fact toast all the waffles at once before assembling the sandwiches.

    You also would have no way of knowing what ingredients the waffwich depends on. If you execute the steps and realize that 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 PB on person a's waffle, or if it seemed faster, you'd just redo these steps.

    This is just a conceptual example, but in practice, having this information in place saves a whole lot of time. In neuroimaging, because we often process many subject simultaneously, expressing parallelism is key. 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 acquisitions, jobs run longer and longer – so the chance of a computer failure (or a disk filling up, or something) during the time that you run a job is more likely. But this information cannot be automatically determined from a bash script.

4. The representation of this information is a directed acyclic graph (DAG). <insert DAG for waffwich>.
    Describe arrows, dependencies, lack of circularity.
    This is a very good way of describing a partially ordered sequence of steps.
    However, writing a graph is a really nasty way of writing down a program – so we need a better syntax for describing dependencies.

5. Write out the "make" representation of waffwich.

```
Waffwich: toastedwaffles berries PB
      Spread PB on toastedwaffle
      Arrange berries in squares on half waffle
      Cut in half and fold

Toastedwaffle: waffle
      Toast waffle
```

Before we leave this very abstract example – let us note a few things.
Target: dependency and recipe
Order – not linear!
However, we've reached the point where we have to leave this example because I have to explain how make knows whether something exists or not, or whether it has to make it. It does this by assuming that the target and the dependencies are files, and checking the dates on these files. There are many exceptions to how this works, but let's accept this simplification for the moment.

6. A real example. Copy directory `oasis-multisubject-sample` to your home directory.
7. The problem is to skull strip all of these brains for subsequent use by FSL. However, note that first you need to reorient each image. So there are two commands to execute.

```
fslreorient SSS_raw.nii.gz SSS_T1.nii.gz

bet SSS_T1.nii.gz SSS_T1_skstrip.nii.gz
```

P1. Write a makefile to process the first subject in this directory. By default, the program "make" reads commands that are saved in a file called "Makefile" (by convention, normally capitalized). By default, according to the manual, GNU make looks for GNUmakefile, makefile, and Makefile – but if you just use one convention you are unlikely to get confused.

You should have the following:

```
OAS2_0001_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
```

Now play with this. When you type “make”, what happens? What is it really doing? <fill in here>

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 these rules, you get different outcomes – NOT because it is reading them one by one. It’s creating a DAG – but it has to start someplace.

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

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

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

P2. Pattern rules and multiple subjects.

This is great, but so far we’ve only been able to process one subject. You can create rules for each subject by cutting and pasting the two rules that you have and editing them. But that’s a huge pain and since it’s a common thing to want to do, there IS a better way.

You can specify in rules that you want to match patterns. Every file begins with the subject identifier (clever us!) so you can use the symbol % to replace the subject in both the target and dependencies. However, what do you do in the recipe when you don’t know the actual names of the files? The % character won’t work there. However, the symbol $* does. But that can get kind of ugly, visually. Some shortcuts that people like to use that are readily recognized are $@ to substitute for the target, and $< to substitute for the first dependency.

```
%_T1_skstrip.nii.gz: %_T1.nii.gz

      bet $< $@


%_T1.nii.gz: %_raw.nii.gz

      fslreorient2std $< $@
```

But try typing make now, with just those rules. You should get an error that there are no targets. Note that % does not work like a wildcard – if you are used to UNIX, you might be thinking that somehow 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. You need to be explicit about what files you want to create.

For example, try typing

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

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

P3. 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, in this case, at the top) that specifies all of the targets you really want to make. It is called a phony target because it's not a file.

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

Add a few subjects and go, by typing "make skstrip".

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

```
clean:
       rm –f *T1_skstrip.nii.gz *T1.nii.gz
```

Note that in the recipe, it's totally fine to use wildcards. The recipe is executed in a shell. Also note the use of the –f flag. Why is that? What happens if you try to remove a file that doesn't exist? Some other things to observe here are that (a) the only thing that enforces this convention is discipline – you need to add every new file that you create in a makefile to the clean target. More dangerous however is 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 any hand editing. Alternatively, I have often run into situations where conversion from dicoms is not straightforward and I have had to use other nonconventional software packages.

For these reasons, I prefer the looser convention of "mostlyclean" – which means to remove things that most certainly can be easily regenerated but to leave important partial products (e.g., your images converted from dicoms, freesurfer directories, final masks used for quantitative analysis, and other things that are project specific).

But how do you tell make that clean is not a real file? What if you go ahead and touch a file called "clean"?

You can tell make that it should disregard any files with the same names as phony targets using the following statement:

```
.PHONY: skstrip clean
```

P4. But we still have the problem that there are a large number of subjects to specify. Thankfully there are a lot of ways to set variables from 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 from this file by using the following make command.

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

Now we can write a target for skull stripping that

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

The pattern substitution rule for changing subjects to files comes from section 8.2 of the GNU make manual. You can look up the other options.

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

P5. Secondary targets

Now if you type make here it is going to go through and do everything. But it's easier to see the point of secondary targets if you use the "-n" flag for make – which means show me what you will do without doing it. It's puzzling, isn't it, that all of the T1 files are deleted! What's wrong here.

Because we used a pattern to define the rule, and because we didn't specify that we also wanted to create the T1 images, they are deleted! This is a good thing or a weird thing, depending on how you look at it. If you really want those "created on the way" targets, you can specify this to Make as follows. Or, you can explicitly make them dependencies for the skstrip target.

```
.SECONDARY: $(SUBJECTS:=_T1.nii.gz)
```

P6. Running in parallel