Lecture: Making Breakfast

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

A few elements of good workflow systems are:

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

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

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

```
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 0.1: Creating a waffwich

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

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

[1] So I'm told.

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 0.1 can alternatively be represented as a directed acyclic graph (Figure 0.2).

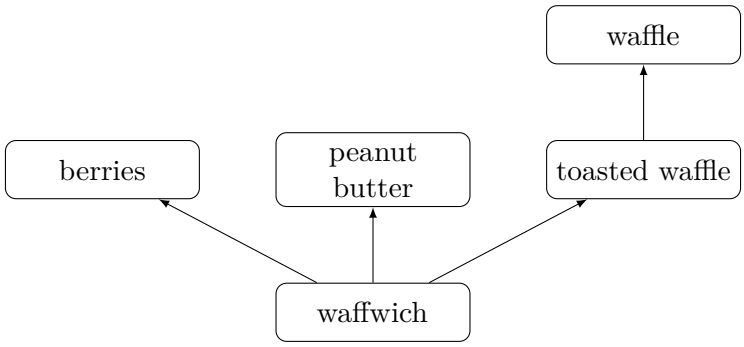

Figure 0.2: How to make a waffwich

A graph like this is a very good way of describing a partially ordered sequence of steps. However, writing a graph is a really nasty way of writing a program, so we need a better syntax for making dependencies. The `make` recipe for a waffwich would look something like Figure 0.3.

! There is no circularity in this graph.

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

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

However, we've reached the point where we have to leave this conceptual example because `make` is aware of whether something exists or not, and whether it has to make it. It does this by assuming that the target and dependencies are files, and by

```
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 0.3: Using `make` to make a waffwich

checking the dates on those files. There are many exceptions to this; but let's accept this simplification for the moment, and move on to a real example with files.

Practical Example: Skull-stripping

Manipulating a single subject

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

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

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

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

and

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

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

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

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

```
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 0.4: Makefile as copied

you change the order of the rules, you get different outcomes, but *not* because it is reading them one by one. It's creating a directed acyclic graph, but it has to start somewhere.

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

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

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

Pattern rules and multiple subjects

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

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

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

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

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

For example, try typing:

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

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

Figure 0.5: Pattern-matched Makefile

```
$ 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 0.6 shows how `make` identifies a pattern from an input and applies it to match the dependencies. Here, you can see that it is important to not include trailing (or leading!) underscores or other characters in your expected pattern.

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

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

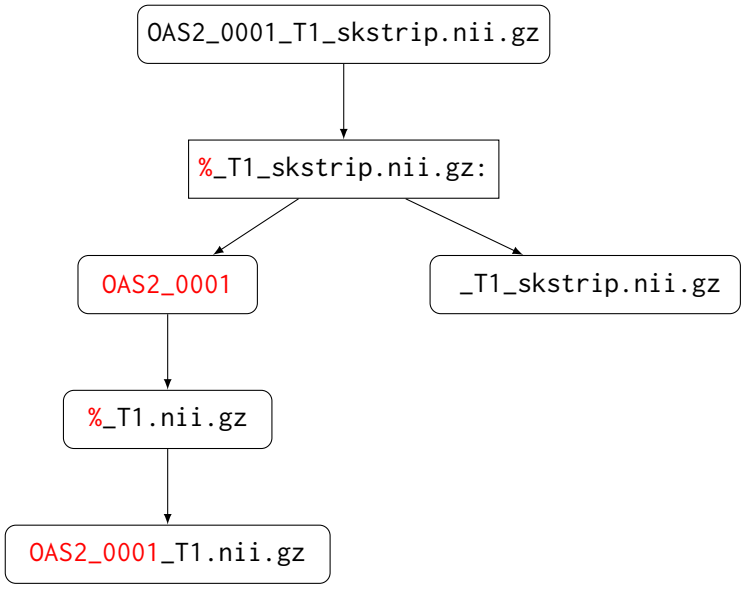

Figure 0.6: Pattern-matching as a sieve

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.

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

`make clean`

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

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

Figure 0.7: `make clean`

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

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

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

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

```
.PHONY: skstrip clean
```