

Practicum 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? With functions similar to that of `gkrellm`, `qmon` 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
```

Once you have answered the password prompt, type `qmon` into the command line. Of interest to us is the first button in the panel that appears – 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 queue, along with the username of the person who submitted those jobs. Ideally, you want to submit your jobs to the cluster when the cluster is free.

However, this scenario does not necessarily always present itself. Your jobs will automatically be queued under ‘Pending Jobs’ if you choose to submit them when the cluster workstations are busy processing other jobs. For more information on how to use the grid engine to run a parallel make workflow, refer to the [manual](#), or

 A list of the clusters and machines available for use at IBIC can be found at: <https://www.ibic.washington.edu/wiki/pages/viewpage.action?pageId=15008331>

Practicum 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
```

This calls up a script called `fs1_sub` that FSL uses to schedule and submit jobs in series according to how some of its cluster aware tools, such as `MELODIC`, operate. Once you have done this, you may go ahead and use `qmake` to submit jobs to the grid engine.

Practical Example: A Test Subject

If you have not already, copy `$MAKEPIPELINES/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 `testsubject`. 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).

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 `testsubject` 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 be 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 failed on many of the brains in the project that this makefile was modeled from.

One technique for estimating the total intracranial volume is simply to linearly register the brain to standard space. 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 a file, type:

```
$ tar cvf /archive.tar
```

To untar a file, 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. 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 volume 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 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 occurred 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 1000 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 can just do it, why not just use it?

Implementing a resting state pipeline: converting a shell script into a makefile

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 bashscript 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.

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.

4. 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 adhere to a general concept – that is, 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.

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.

❗ 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.

Implementing a resting state pipeline: converting a shell script into a makefile

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

```
#!/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 0.16: `resting-script` file

The above would look like such in a makefile:

```
③ PROJECT_HOME=/mnt/home/username/testsubject/testsubject
④ SHELL=/bin/bash

⑤ .PHONY = all clean

⑥ all: xfm_dir/T1_to_MNI.mat rest_mc

⑦ #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

⑧ #Motion Correction
rest_mc.nii.gz: rest.nii.gz
    mcflirt -plots rest_mc.par -in rest.nii.gz -out rest_mc -
        rmsrel -rmsabs
```

! 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.

- ③ When creating your makefile, you must specify your Project home.
- ④ Be sure to tell `make` to use `bash` by setting the default shell.
- ⑤ 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.
- ⑥ Define `all` to include all your targets.
- ⑦ 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 line 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`.
- ⑧ 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`.