Using snakemake to create robust and reproducible bioinformatic pipelines

Dan MacLean

10/25/2022

Table of contents

Preface

Motivation

Do you ever feel like the large pipeline or large number of steps in your bioinformatic analysis is a pressure? That somehow it is in charge and you are just there to sit and tell the computer what to do, over and over? Do you fear having missed a step or mis-specified a file and lose sleep from the horror of having to re-do something over again because of a reviewer request? Fear not, these terrors are exactly what `snakemake` is designed to help slay. `snakemake` can help you build robust (in the sense that it can be stopped by an unexpected hiccup and can restart from where it left off once that hiccup is cleared) and reproducible pipelines in a quick and easy fashion.

`snakemake` is one of a number of tools that allows you to chain together multiple processes into a pipeline. These tools are sometimes called workflow managers and they tie processes together by having some model of the dependency structure between the inputs and outputs of the steps of a pipeline.

Things like `bash` scripts *can* do this job, but they're bad ways if we want to be reproducible and robust to failure without resorting to heavily engineering scripts to recognise when inputs/outputs change. Dependency based tools like `makefiles` and their derivatives have been around for decades, doing similar jobs but recently more pipeline specific tools like `snakemake`, `Nextflow Common Workflow Language` and even the graphical `Galaxy` workflows have appeared specifically for scalable reproducible analysis pipelines.

`snakemake` is a good choice as it has a lightweight Python based syntax that will be familiar to many users.

In this short tutorial we'll look at how to create `snakemake` pipelines for use on a `slurm` cluster like the one in use at TSL.

Setup and Prerequisites

This tutorial presumes you are at least a little familiar with bash scripts and Python (but not much) and that you have experience submitting and running jobs on a `slurm` cluster.

To replicate the examples, you'll need the data here sample_data.zip and the programs, `minimap2`, `samtools` and `snakemake` which you can install from conda and bioconda.

If you need any help with this please see the bioinformatics team.

1 First Look

In this section we'll look at making and running a first pipeline and understanding why `snakemake` is better for you than `bash` scripts.

1.1 Making a `snakefile`

`snakemake` is intended to replace the mess of `bash` scripts you use to run your workflows. So let's look at converting a simple `bash` script to `snakemake`

```
minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq | \
samtools view -S -h -b -q 25 -f 3 > aln.bam
samtools sort aln.bam -o aln.sorted.bam
```

The two commands convert into the following `snakemake` rules

```
rule sort:
  input: 'aln.bam'
  output: 'aln.sorted.bam'
  shell: "samtools sort aln.bam -o aln.sorted.bam"

rule align_and_bam:
  input:
    fq1="ecoli_left_R1.fq",
    fq2="ecoli_right_R2.fq",
    ref="ecoli_genome.fa"
  output: "aln.bam"
  shell: "minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq | samtools vi

rule final_alignments:
    input: 'aln.sorted.bam'
```

Here are the key points

1. A `snakemake` file is composed of `rules`

2. Each rule has (at least) one input file, (at least) one output file and a command for making the output file from the input
3. Rules are therefore linked into a chain or tree by the files that go in and come out
4. There is an extra final rule that specifies the final result of the pipeline. This rule has no output or command, only inputs

That is the basis of it!

1.2 Running the pipeline

We run the rules by putting them in a file. Usually this is suffixed with `.snakefile` to give something like `my_pipeline.snakefile`. When we come to run the pipeline `snakemake` needs us to tell it the name of the rule to run (recall our base rule is `final_alignments`), and the snakefile and the cores the pipeline is allowed to use with the `-c flag`.

1.2.1 The dry-run

Usually we don't want to run the pipeline without doing some sort of checking first. This is the purpose of the dry run feature. This allows us to see the jobs that the file specifies, without actually doing them, the snakemake `-n` flag creates a dry-run. Put together that makes something like

```
snakemake final_alignments --snakefile my_pipeline.snakefile -c1 -n
```

Default file and rules

Actually, `snakemake` looks for the default file `Snakefile` and the main rule `all` so if you use them you can have a command like `snakemake -c1 -n` which is less verbose but also less explicit.

We get a lot of output. It can be broken down into a few bits,

1. The Summary
2. The Jobs
3. The to-be-done list

1.2.1.1 The Summary

At the top of the file we are given a summary of the number of times each rule will be run and the resources specified (here just the defaults)

```
Building DAG of jobs...
Job stats:
job                 count    min threads    max threads
----------------  -------  -------------  -------------
align_and_bam           1              1              1
final_alignments        1              1              1
sort                    1              1              1
total                   3              1              1
```

1.2.1.2 The Jobs

Next we get a much more granular view, each job is presented with the expected input and output and a reason why it needs running. Usually this will be either `missing output files` IE the output hasn't been created so the job still needs to run or `Input files updated by another job` meaning that an input file is newer than an output file (or when it is created it will be) so this file needs updating.

```
[Wed Oct 26 16:51:18 2022]
rule align_and_bam:
    input: ecoli_left_R1.fq, ecoli_right_R2.fq, ecoli_genome.fa
    output: aln.bam
    jobid: 2
    reason: Missing output files: aln.bam
    resources: tmpdir=/var/folders/22/kjdvv_k14cj1m6hq5hl527qw0006zc/T


[Wed Oct 26 16:51:18 2022]
rule sort:
    input: aln.bam
    output: aln.sorted.bam
    jobid: 1
    reason: Missing output files: aln.sorted.bam; Input files updated by another job: aln.bar
    resources: tmpdir=/var/folders/22/kjdvv_k14cj1m6hq5hl527qw0006zc/T


[Wed Oct 26 16:51:18 2022]
```

```
localrule final_alignments:
    input: aln.sorted.bam
    jobid: 0
    reason: Input files updated by another job: aln.sorted.bam
    resources: tmpdir=/var/folders/22/kjdvv_k14cj1m6hq5hl527qw0006zc/T
```

1.2.1.3 The to-be-done list

This is a version of the summary outlining the bits of the pipeline that need to complete for everything to be in order.

```
Job stats:
job                 count    min threads    max threads
----------------  -------  -------------  -------------
align_and_bam           1              1              1
final_alignments        1              1              1
sort                    1              1              1
total                   3              1              1

Reasons:
    (check individual jobs above for details)
    input files updated by another job:
        final_alignments, sort
    missing output files:
        align_and_bam, sort

This was a dry-run (flag -n). The order of jobs does not reflect the order of execution.
```

Using a log

As your list of jobs grows and the output from `snakemake` becomes large, its best to use a log file. Do that with `-o somename.log` e.g `snakemake --snakefile final_alignments my_pipeline.snakefile -n -o my_pipeline.log`

1.2.2 The Run Proper

Everything looks good in the dry-run so let's go ahead and run. Although we haven't made explicit point of it, this run will happen in the current directory with all files expected to be in and going to the current directory. That looks like this at the moment.

```
$ ls -l
total 85136
-rw-r--r--@ 1 macleand  2006   5205449  2 Jul  2019 ecoli_genome.fa
-rw-r--r--@ 1 macleand  2006  19186649 28 Nov  2019 ecoli_left_R1.fq
-rw-r--r--@ 1 macleand  2006  19186649 28 Nov  2019 ecoli_right_R2.fq
-rw-r--r--  1 macleand  2006       420 26 Oct 16:50 my_pipeline.snakefile
```

Run the pipeline with

```
snakemake final_alignments --snakefile my_pipeline.snakefile -c1
```

We get a lot of output to the screen (or the log if we specified that). Hopefully at the end we see

```
Finished job 0.
3 of 3 steps (100%) done
```

an indication that it has completed everything (if not we're into some debugging - more on that later). And the working directory looks like this now

```
-rw-r--r--  1 macleand  2006   9397482 28 Oct 10:13 aln.bam
-rw-r--r--  1 macleand  2006   7929255 28 Oct 10:13 aln.sorted.bam
-rw-r--r--@ 1 macleand  2006   5205449  2 Jul  2019 ecoli_genome.fa
-rw-r--r--@ 1 macleand  2006  19186649 28 Nov  2019 ecoli_left_R1.fq
-rw-r--r--@ 1 macleand  2006  19186649 28 Nov  2019 ecoli_right_R2.fq
-rw-r--r--  1 macleand  2006       420 26 Oct 16:50 my_pipeline.snakefile
```

All the files we expected to be created have been and are sitting nicely in the directory. Hurray!

1.3 The first awesome thing about `snakemake`

So far this has all been very much like a bash script. The `snakemake` file seems to be just an elaborate reproduction. Now lets have a look at a killer feature that makes `snakemake` very much more useful than bash scripts - its ability to work out whether all parts of the pipeline are up to date and whether anything needs redoing.

Let's look at the dry-run output from the pipeline we just ran.

```
$ snakemake final_alignments --snakefile my_pipeline.snakefile -c1 -n
Building DAG of jobs...
Nothing to be done (all requested files are present and up to date).
```

Well, that's reassuring. Nothing need be done. What if a component file changed. Lets look at what happens if an input file is updated. Using `touch` to update the timestamp on the reference file

```
$ sample_data touch ecoli_genome.fa
$ sample_data ls -l
total 119480
-rw-r--r--  1 macleand  2006   9397482 28 Oct 10:13 aln.bam
-rw-r--r--  1 macleand  2006   7929255 28 Oct 10:13 aln.sorted.bam
-rw-r--r--@ 1 macleand  2006   5205449 28 Oct 10:32 ecoli_genome.fa
-rw-r--r--@ 1 macleand  2006  19186649 28 Nov  2019 ecoli_left_R1.fq
-rw-r--r--@ 1 macleand  2006  19186649 28 Nov  2019 ecoli_right_R2.fq
-rw-r--r--  1 macleand  2006       420 26 Oct 16:50 my_pipeline.snakefile
```

One of the source files is now newer than the outputs. What does `snakemake` now think needs to be done

```
$ snakemake final_alignments --snakefile my_pipeline.snakefile -c1 -n

Building DAG of jobs...
Job stats:
job                 count    min threads    max threads
----------------  -------  -------------  -------------
align_and_bam           1              1              1
final_alignments        1              1              1
sort                    1              1              1
total                   3              1              1
```

it thinks that the whole pipeline needs to be redone! This is the first awesome thing about `snakemake` - if one of the upstream files is updated (input or output files, it doesn't matter), the relevant parts of the pipeline will be slated to run again (which in this small pipeline will be everything). `snakemake` will work out which bits need doing again automatically from the rule descriptions. In large pipelines this is a major time saver and increases reproducibility massively. Bash scripts must be manually managed which leads to more manual errors.

Note

I find it really hard to overstate how useful this ability to pick-up-from-where-it-left-off is. It saves an immense amount of checking and redoing and re-issuing of the same commands when something went wrong - especially something catastrophic or hard to detect at the end. Or when you unexpectedly get a new sample and need to add it in, or when your boss wants to change one tiny thing. `snakemake` insane reproducibility is a huge win for research which is naturally iterative. Its true that there's a slight learning curve and requires more investment in time at the start of the project, but that is won back in spades later.

1.4 The second awesome thing about `snakemake`

The next life improving thing about `snakemake` is how it handles files. Up to now we've hardcoded the file names into the rules. Thats not scalable. `snakemake` provides a clever pattern match facility between the input and output files to match them up between the rules, it cross-references these with actual filenames and fills in the patterns. It provides access to them using special objects called `wildcards`, `input` and `output`. All we need to do is specify the patterns in our rules and the inputs to the master rule (ie `final_alignments`). In essence, we start by describing the files we want to get out of the pipeline and `snakemake` works back from there according to our pattern.

This is massively easier to see in the rules themselves. Here's our original set of rules.

```
rule sort:
  input: 'aln.bam'
  output: 'aln.sorted.bam'
  shell: "samtools sort aln.bam -o aln.sorted.bam"

rule align_and_bam:
  input:
    fq1="ecoli_left_R1.fq",
    fq2="ecoli_right_R2.fq",
    ref="ecoli_genome.fa"
  output: "aln.bam"
  shell: "minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq | samtools vi

rule final_alignments:
  input: 'aln.sorted.bam'
```

And here's our new set of rules

```
rule final_alignments:
  input:
    ecoli='ecoli.sorted.bam',
    pputida='pputida.sorted.bam'

rule sort:
  input: "{sample}_aln.bam"
  output: "{sample}.sorted.bam"
  shell: "samtools sort {input} -o {output}"

rule align_and_bam:
  input:
    fq1="{sample}_left_R1.fq",
    fq2="{sample}_right_R2.fq",
    ref="{sample}_genome.fa"
  output: "{sample}_aln.bam"
  shell: "minimap2 -ax sr {input.ref} {input.fq1} {input.fq2} | samtools view -S -h -b -q
```

Things to note:

1. The final rule is at the top - it doesn't actually matter to `snakemake` what order the rules are in. In many cases for us it's easier to have the last rule at the top. Note the common name scheme between the input file names.
2. The `{sample}` is the wildcard. The `{}` is a replacement operator, the value of the wildcard will be put in there at runtime.
3. The special `{input}` and `{output}` objects can have more than one attribute, so we can have more than one input or output file into a rule.
4. We can thread the replacements into the `shell` commands to make them generic across samples too.

So we can hopefully see how the rules link up to each other.

What does the dry-run say in the following folder with multiple samples in there?

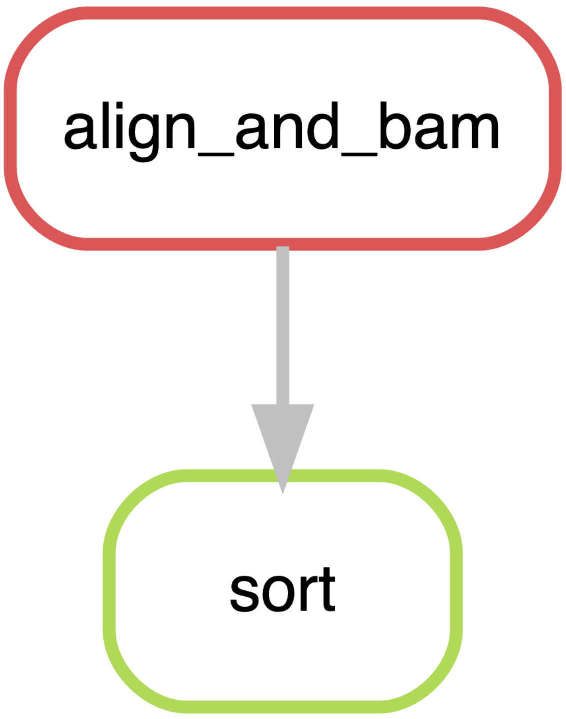
```
$ ls -l
total 170264
-rw-r--r--@ 1 macleand  2006   5205449 28 Oct 11:04 ecoli_genome.fa
-rw-r--r--@ 1 macleand  2006  19186649 28 Oct 11:04 ecoli_left_R1.fq
-rw-r--r--@ 1 macleand  2006  19186649 28 Oct 11:04 ecoli_right_R2.fq
-rw-r--r--  1 macleand  2006       479 28 Oct 11:28 multi.snakefile
-rw-r--r--@ 1 macleand  2006   5205449 28 Oct 11:04 pputida_genome.fa
-rw-r--r--@ 1 macleand  2006  19186649 28 Oct 11:04 pputida_left_R1.fq
-rw-r--r--@ 1 macleand  2006  19186649 28 Oct 11:04 pputida_right_R2.fq
```

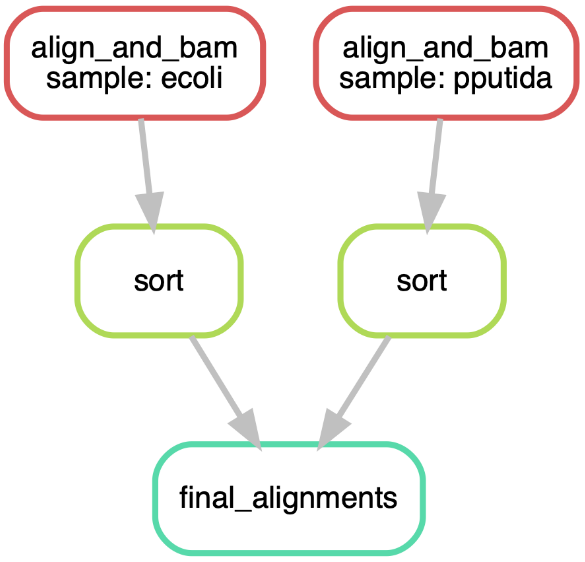
```
$ snakemake final_alignments --snakefile multi.snakefile -c1 -n
Job stats:
job                 count    min threads    max threads
----------------  -------  -------------  -------------
align_and_bam           2              1              1
final_alignments        1              1              1
sort                    2              1              1
total                   5              1              1

Reasons:
    (check individual jobs above for details)
    input files updated by another job:
        final_alignments, sort
    missing output files:
        align_and_bam, sort
```

Looks great! `snakemake` has found all our new samples and increased the number of jobs needed accordingly. The figures below show a graphical version of the pipelines. See how snakemake has made it very easy to jump up in scale.

1.5 Summary

We've seen how the fundamentals of `snakemake` allow us to build efficient and reproducible pipelines. In the next section we'll look at features of `snakemake` that help to power larger and more complex pipelines.


align_and_bam
sort


align_and_bam
sample: ecoli
align_and_bam
sample: pputida
sort
sort
final_alignments

2 Useful `snakemake` features

2.1 The `expand()` function

`snakemake` requires a list of files in it's rule inputs. These are just standard Python lists and can be made using functions. A helper function called `expand()` does some wildcard expansion of its own. You can see its use in our `final_alignments` rule here.

```
samples = ['ecoli', 'pputida']

rule final_alignments:
  input: expand( "{sample}.sorted.bam", sample=samples)
```

We can create all the input files programatically using a list of names `samples` and using the `expand()` function which just slots each of the values into its proper place to create a list, saving us a lot of definitions on large sample sets. This will work the same if we give it more than one list and wildcard to expand, like this

```
samples = ['ecoli', 'pputida']
timepoints = ['0h', '2h']
treatments = ['test', 'control']

rule final_alignments:
  input: expand( "{sample}_{time}_{treatment}.sorted.bam", sample=samples, time=timepoints
```

which will create all the combinations of those lists.

2.2 The `config.yml` file

We won't often have all our files in the current directory, nor want our results and intermediate files to go there, they'll usually be spread about the filesystem. Which means we will have to start dealing with varied paths. Recall that `snakemake` *is* Python. This means that we can create paths using standard Python string operations like `+`. This is most useful when combined with a `config.yml` file which looks something like this

```
scratch: "/path/to/a/scratch/folder/"
databases: "/path/to/a/database/folder/"
results: "/path/to/a/results/folder"
```

These paths make up a base set of paths that we may want to write or read from in our rules. When loaded into the snakefile a Python `dict` object called `config` is created that we can access using the keys named in `config.yml`. Here's an example

```
samples = ['ecoli', 'pputida']
timepoints = ['0h', '2h']
treatments = ['test', 'control']

configfile: "config.yml"

rule final_alignments:
  input: expand( config['results'] + "{sample}_{time}_{treatment}.sorted.bam", sample=samp

rule sort:
  input: config['scratch'] + "{sample}_{time}_{treatment}_aln.bam"
  output: config['results'] + "{sample}_{time}_{treatment}.sorted.bam"
  shell: "samtools sort {input} -o {output}"

rule align_and_bam:
  input:
    fq1="{sample}_{time}_{treatment}_left_R1.fq",
    fq2="{sample}_{time}_{treatment}_right_R2.fq",
    ref=config['databases'] + "{sample}_genome.fa"
  output: config['scratch'] + "{sample}_{time}_{treatment}_aln.bam"
  shell: "minimap2 -ax sr {input.ref} {input.fq1} {input.fq2} | samtools view -S -h -b -q
```

It should be easy to see how to load the config file and inject the values into our paths nicely.

2.3 `lambda` functions

In the `config` example above it may have been conspicuous that the fastq files were not graced with the information from the config file. This gives us opportunity to explore how to use the wildcard information to get a path using custom functions. For input files, `snakemake` allows us to use a Python `lambda` function. These are one line functions that don't get a name. You can pass them the `wildcards` object and get them to call a second function that uses that information to generate the pathname for the file. Have a look at this snippet

```
rule align_and_bam:
  input:
    fq1=lambda wildcards: my_function(wildcards, "fq1")
    fq2=lambda wildcards: my_function(wildcards, "fq2")
```

The function `my_function()` *must* return a single pathname as a string, as it is *just* Python the function can be defined in the top of the `snakemake` file or imported. We'll look at these in more depth later.

2.4 Rerunning a specific step

If we really want to micro-manage our pipeline we can run individual steps at will. We have up to now been running the whole thing from the final rule. But any rule can be taken as the end point. Just use its name in the invocation,

```
snakemake <any rule> --snakefile my.snakefile
```

2.5 Deleting intermediate files

Quite often there's no need to keep anything but the final result file(s). Since we can regenerate intermediate files easily using its rule in the `snakefile` we can usually just tell `snakemake` to remove output files when they're no longer needed by wrapping the path in the `temp()` function, like this

```
rule align_and_bam:
  input:
    fq1=lambda wildcards: my_function(wildcards, "fq1")
    fq2=lambda wildcards: my_function(wildcards, "fq2")
    ref=config['databases'] + "{sample}_genome.fa"
  output: temp(config['scratch'] + "{sample}_{time}_{treatment}_aln.bam")
  shell: "minimap2 -ax sr {input.ref} {input.fq1} {input.fq2} | samtools view -S -h -b -q
```

This saves as lot of space during runtime for big pipelines *and* saves a lot of clean up.

2.6 More `shell`

In all our examples we've used a `shell` line to hold the command. We can make the `shell` command multi-line by wrapping it in Python triple quotes, enabling us to have longer commands/chains in the snakefile

```
rule align_and_bam:
  input:
    fq1=lambda wildcards: my_function(wildcards, "fq1")
    fq2=lambda wildcards: my_function(wildcards, "fq2")
    ref=config['databases'] + "{sample}_genome.fa"
  output: temp(config['scratch'] + "{sample}_{time}_{treatment}_aln.bam")
  shell:"""
minimap2 -ax sr {input.ref} {input.fq1} {input.fq2} | \
samtools view -S -h -b -q 25 -f 3 > {output}
"""
```

A common alternative that prevents the snakefile from getting gummed up with job specifics is just to put the commands in a bash script and call that. Any script that can be run on the command line can be run this way, including Python, R etc

```
rule align_and_bam:
  input:
    fq1=lambda wildcards: my_function(wildcards, "fq1")
    fq2=lambda wildcards: my_function(wildcards, "fq2")
    ref=config['databases'] + "{sample}_genome.fa"
  output: temp(config['scratch'] + "{sample}_{time}_{treatment}_aln.bam")
  shell:"bash scripts/do_alignments.sh {input.ref} {input.fq1} {input.fq2}"
```

The `shell:` can also be replaced with `run:` which allows you to use Python directly in the snakefile.

2.7 Drawing the pipeline

It is possible to get `snakemake` to generate a picture of your pipeline, which is great for understanding when things get complicated or showing your boss how involved these things are. We use the `--dag` option in conjunction with `graphviz` (which will need installing separately). Here's the magic spell

```
snakemake --snakefile my.snakefile --dag | dot -Tpng -Gsize=9,15\! -Gdpi=100 > pipeline.png
```

If you don't want the whole set of files in there, and just want to see the 'core' rules then you can use `--rulegraph` instead of `--dag`

```
snakemake --snakefile my.snakefile --rulegraph | dot -Tpng -Gsize=9,15\! -Gdpi=100 > pipelin
```

2.8 Summary

These are all helpful `snakemake` features that will help your snakefile work more easily in a real setting. Most pipelines you develop will use most of these features.

3 Working on a slurm cluster

In this section we'll look at how to adapt your snakefile to run well across many nodes of a cluster. We'll look at

1. The `params` object
2. The command-line options for `snakemake` on the cluster
3. Custom functions for filenames

3.1 How snakemake expects to run on a slurm cluster

Briefly, `snakemake` expects each job to run individually on different machines on the cluster under the management of one core job that runs for the duration of the pipeline. That means that each job can have its own parameters and jobs will dispatch more quickly if they get the correct parameters for their needs. We will learn how to set the jobs parameters through the `params` object in the rule.

`params` is a rule attribute, like `input` and `output` that can take parameters to be passed through to the `shell` command run by that rule. It also can be referenced in the command-line invocation of snakemake. This makes it perfect for setting extra job options. Lets look at three examples of use, first just passing an option to a command

3.2 params

3.2.1 Keeping the rule clean

This is mostly a way to be explicit and make params easy to see and rules clean. Use the new params block like the wildcards

```
rule align_and_bam:
  input:
    fq1=lambda wildcards: my_function(wildcards, "fq1")
    fq2=lambda wildcards: my_function(wildcards, "fq2")
    ref=config['databases'] + "{sample}_genome.fa"
  output: temp(config['scratch'] + "{sample}_{time}_{treatment}_aln.bam")
  params:
    quality=25,
    flags=3
  shell:"""
minimap2 -ax sr {input.ref} {input.fq1} {input.fq2} | \
samtools view -S -h -b -q {params.quality} -f {params.flags} > {output}
"""
```

3.2.2 Dynamic parameter setting

We can use `lambda` functions to generate values for parameters if we need to based on the values of wildcards, here we guess the memory needed for a job

```
rule align_and_bam:
  input:
    fq1=lambda wildcards: my_function(wildcards, "fq1")
    fq2=lambda wildcards: my_function(wildcards, "fq2")
    ref=config['databases'] + "{sample}_genome.fa"
  output: temp(config['scratch'] + "{sample}_{time}_{treatment}_aln.bam")
  params:
    mem=lambda: wildcards: guess_parameter(wildcards)
    quality=25,
    flags=3
  shell:"""
minimap2 -ax sr {input.ref} {input.fq1} {input.fq2} | \
samtools view -S -h -b -q {params.quality} -f {params.flags} > {output}
"""
```

Note how we don't use this in the actual shell block. What's going on there? This links us to our third use, using the `params` info to set `slurm` options for this job.

3.2.3 Using `params` to set `slurm` job options

The `snakemake` command has an option called `--cluster` that specifies a template for the submission for each job. It takes a string that will resolve wildcards and pass them as options for `slurm`. Look at this

```
snakemake --snakefile my.snakefile --cluster 'sbatch --mem={params.mem}'
```

The `--cluster` option is hijacked for each job and values from the snakefile pushed in when the job is submitted. In practice this means that each job will get its own specific value of `mem` from its `params` block, allowing us to specify the value as needed.

We can specify more options arbitrarily. The following allows us to specify the queue as well and for some reason the number of cores (threads), has its own argument, giving us a rule like this ready for the cluster.

```
rule align_and_bam:
  input:
    fq1=lambda wildcards: my_function(wildcards, "fq1")
    fq2=lambda wildcards: my_function(wildcards, "fq2")
    ref=config['databases'] + "{sample}_genome.fa"
  output: temp(config['scratch'] + "{sample}_{time}_{treatment}_aln.bam")
  threads: 8
  params:
    mem="32G",
    queue="tsl-short",
    quality=25,
    flags=3
  shell:"""
minimap2 -ax sr {input.ref} {input.fq1} {input.fq2} | \
samtools view -S -h -b -q {params.quality} -f {params.flags} > {output}
"""
```

Which we would add into the command line as

```
snakemake --snakefile my.snakefile --cluster 'sbatch --mem={params.mem} --partition={params.
```

3.3 `source`

On a cluster like the one TSL uses each job in the snakemake pipeline runs on a machine with its own execution environment, that is its own memory, CPUs and loaded software. This means that things like `source` and `singularity` images have to be loaded in each job and not globally. They wont work if you put them in the command line, instead they have to go in the `shell` block (or script) like this

```
  shell:"""
source minimap2-2.5
source samtools-1.9
minimap2 -ax sr {input.ref} {input.fq1} {input.fq2} | \
samtools view -S -h -b -q {params.quality} -f {params.flags} > {output}
"""
```

3.4 Using a file-of-files as a database for mapping wildcards to filesystem paths

Often we will want to use filenames that have no indication of our sample name or other wildcards in them. This might be because they are raw datafiles from sequencing providers and we don't want to change the filenames or copy the large files across the filesystem from a central storage. Because we are able to use `lambda` functions in `snakemake` and any old Python we can make a database of mappings between the wildcard names and other things like filepaths in a csv file, and read it in for use at any time we like.

Consider the following sample file (name `sample_info.csv`)

```
sample, fq1_path, fq2_path, treatment, time
pputida, /my/seq/db/pputida_1.fq, /my/seq/db/pputida_2.fq, test, 0h
ecoli, /my/seq/db/ecoli_mega_R1.fastq.gz, /my/seq/db/ecol_mega_R2_fastq.gz, control, 6h
...
```

Note that the file names don't have a common pattern, so won't easily be resolved by `snakemake` wildcards. Instead we can build lists of the columns by parsing the file in a usual Python way at the top of the `snakemake` file

```
samples = []
fq1 = []
fq2 = []
times = []
treatments = []
with open("sample_info.csv") as csv:
    for l in csv:
        l = l.rstrip("\n")
        if not l.startswith("sample"):
            els = l.split(",")
            samples.append( els[0] )
            fq1.append( els[1] )
            fq2.append( els[2] )
            times.append( els[3] )
            treatments.append( els[4])
```

We can generate functions that given a `sample` will return the other items e.g `fq`

```
def sample_to_read(sample, samples, fq):
'''given a sample and list of samples and a list of fqs returns the fq with the same index
as the sample'''
    return fq[samples.index(sample)]
```

So now we can use the wildcard to get back the fq file in the `lambda` function in the rule like this

```
rule align_and_bam:
  input:
    fq1=lambda wildcards: sample_to_read(wildcards.sample, samples, fq1)
    fq2=lambda wildcards: saample_to_read(wildcards.sample, samples, fq2)
```

Which returns the full filesytem path for each fq based on the `sample` wildcard.

This is a really useful feature, but it can be tempting to think of it as a solution to everything. Try to use it only for files that come *into* the `snakemake` pipeline at the beginning and not for things that are generated internally or for final outputs.

3.5 Setting the log file and naming the job

It is a good idea to explicitly set the log filename. Otherwise the run log info will go to a generically named slurm output file. This is a problem because every job run on the HPC under `snakemake` generates a slurm output file which is generically named and the main one can get lost. Similarly, the output from the slurm `squeue` command can get busy with many jobs, so it can also be a good idea to set the main job's name. Logfile can be set with the `sbatch` option `-o` and name with `-J` e.g

```
sbatch -J my_jobs -o my_jobs.log ...
```

3.6 Assigning the maximum number of parallel jobs

You can limit the number of jobs that will run concurrently with `-j`. `snakemake` will not allow more than the specified number of jobs into the queue at any one time. It will manage the submission of jobs right until the completion of the pipeline whatever value you choose. It doesn't create any extra work for you, just throttles `snakemake` should you require it. EG

```
snakemake --snakefile my_pipeline.snakefile --j 20
```

3.7 Waiting for the filesystem to catch up

In a HPC environment we sometimes have to wait for processes to finish writing to disk. These operation can be considered complete by the operating system but still need writing or the filesystem fully updated. So if a new process in a pipeline can't find the output its expecting from a finished process becauce the filesystem is behind, the whole thing could fall over. To avoid this we can set a latency time in which the `snakemake` process will wait and keep checking for the file to arrive before crashing out. Ususally 60 seconds is fine. Set it as follows

```
snakemake --snakefile my_pipeline.snakefile --latency-wait 60
```

3.8 Unlocking a crashed process

Occasionally the `snakemake` pipeline will crash, often because one of its dependent jobs has failed to complete properly (perhaps it ran out of memory). In this state `snakemake` will become locked, to prevent further corruption of the pipeline. The next step is for you to check the logs to see what went wrong and manually resolve it. Then (and only then) can you unlock the `snakemake` pipeline and restart it. Thankfully, `snakemake` will pick up from where it left off, so no earlier progress will be lost.

You can unlock with the `snakemake` option `--unlock`, e.g

```
snakemake --snakefile my_pipeline.snakefile --unlock
```

3.9 Creating a dispatch script

With all these things to remember for the cluster it can be useful to write a master dispatch script to hold the `snakemake` and cluster options. Here's an example one that allows to call it in one of the following three ways

1. `bash do_snake.sh`
2. `bash do_snake.sh dryrun`
3. `bash do_snake.sh unlock`

`1.` Let's you run the pipeline proper, `2.` does the dryrun, `3` will unlock a crashed process.

Here's what that example script looks like

```
if [ -z "$1" ]
then
    sbatch -J my_job \
    -o my_job.log \
    --wrap="source snakemake_x.x.x; snakemake --snakefile my_pipeline.snakefile my_main_rule
    -j 20 \
    --latency-wait 60"
elif [ $1 = 'unlock' ]
then
    sbatch -J unlock \
        -o my_job.log \
        --wrap="snakemake_x.x.x; snakemake --snakefile my_pipeline.snakefile --unlock" \
        --partition="tsl-short" \
        --mem="16G"
elif [ $1 = "dryrun" ]
then
    sbatch -J dryrun \
    -o my_job.log \
    --wrap="source snakemake_x.x.x; snakemake --snakefile my_pipeline.snakefile -n" \
    --partition="tsl-short" \
    --mem="16G"
fi
```

Note that `snakemake` will need loading with `source`.

3.10 Organising the `snakemake` bits and pieces

If you are going to end up with a pipeline with lots of steps and subscripts and a dispatch script, it can be a good idea to organise into a project structure. Consider putting the scripts and results in separate directories and temp files into scratch as discussed in the `config file` section. Then consider the top level directory as the base for executing everything. Something like this would be good

```
$ pwd
my_pipeline
$ tree
.
  README.txt
  config.yaml
  results
  scripts
```

```
        do_pipeline.sh
        my_pipeline.snakemake
```

so that when you're in the `my_pipeline` directory everything can be run as e.g `bash scripts/do_pipeline.sh dryrun`

4 Summary

4.1 `snakemake` has lots of useful qualities

We've seen how to build `snakemake` pipelines that can handle multiple steps and experimental variables and how to scale it to a cluster or a scattered filesystem. We've also seen how `snakemake` can increase reproducibility and save operator time massively

4.2 But being 'general' over lots of pipelines isn't one of them

Perhaps at this stage you'd be expecting to see something like 'how to generalise `snakemake`' and re-use it over and over in different projects. That's not really something `snakemake` is for. As its really tied to a filesystem then its often a bit fiddly to get to generalise. Instead, take advantage of the fact that `snakemake` is really good at re-doing stuff. Make as many files as you can temporary with `temp()` and remove final results. The resulting 'shell' of the pipeline could be versioned - perhaps in git - but maybe with something as simple as a datestamp on the project folder. That way you won't lose that iteration of the pipeline, and it can be re-run exactly should you need to back track.

4.3 It is worth the effort

Hopefully this quick intro hasn't made you think that all this is too much effort. I like to think of tools like `snakemake` as being things that put you back in charge of the computer. A computer is supposed to be a machine to make your life easier. When we get into a situation where all our work in a task is repetitive then we've missed a chance to do that. The learning curve of `snakemake` is no greater than that you took to learn your first bash script so take the leap and put yourself back in charge.

5 A Checkout Challenge

If you're looking for a task to do in `snakemake` why not try this experiment on the TSL cluster.

On TSL's sequence database we have this *Arabidopsis* project. It contains six samples and two run files per sample. Write a `snakemake` pipeline to

1. QC the reads with e.g `fastqc`
2. Find SNPs with e.g `vcftools`
3. Estimate read counts/TPM with e.g `kallisto`

Happy `snakemake`-ing.