A minimal quantitative RNAseq Pipeline

Dan MacLean

2020-01-22

Contents

1 About this course **5**
1.1 Prerequisites 5

2 Counting Aligned Reads in Genomic Regions **9**
2.1 About this chapter 9
2.2 Counting the number of reads that have aligned to gene regions . 9
2.3 Preparing the input 10
2.4 Running `make_counts()` 12
2.5 Summaries and Diagnostic plots 13
2.6 Extracting and saving the count matrix 16

3 Running `minimap2` **19**
3.1 The `minimap2` command and options 19
3.2 Further Reading 20

4 Filtering Badly Aligned Reads **21**
4.1 SAM Format 21
4.2 `samtools` 22
4.3 The `samtools` command and options 22
4.4 Checking the filtering 22
4.5 Are we done? 23
4.6 Further Reading 23

5 Connecting Programs and Compressing output **25**
5.1 BAM Files 26
5.2 Connecting Program Input and Output With Pipes 26
5.3 From reads to filtered alignments in one step 27
5.4 Sorting BAM files 27
5.5 Indexing the sorted BAM 28
5.6 Further Reading 28

6 Automating The Process **29**
6.1 Shell scripts 29
6.2 Creating a script that automates our alignment pipeline. 30

6.3 Running the script 30
6.4 Running on different input files 30

7 Running an alignment on the HPC **33**
7.1 An HPC is a group of slave computers under control of a master computer 33
7.2 Logging into the submission node 35
7.3 Preparing a job 35
7.4 Submitting with `sbatch` 37
7.5 Checkout tasks 38
7.6 Further Reading 38

Chapter 1

About this course

In this short course we'll look at a method for getting quantitative estimates of gene expression from RNAseq data. The course assumes that you will already have performed a read alignment so is *not* a 'read to results' course. The course is very brief and will show you how to use a perform a common pipeline centered around `DESeq` in `R` and `RStudio`

I acknowledge that there are lots of other programs and methods - this course is *not* meant to be comprehensive, it is meant to get you being productive. Seek out further advice if you need to run other programs or systems. Do be encouraged though, lots of what you learn here will be applicable to other pipelines for the same job (they all run in a similar manner with similar objects) so this is a good place to start.

The course is intended to run on your 'local' machine, that is to say, your laptop or desktop computer. In general these machines will be powerful enough for most datasets though the pipeline we will learn can be easily adapted for a high performance computing environment if you need greater computational power.

1.1 Prerequisites

This course assumes that you are a little familiar with the basics of running R and R commands from the R console. You'll need to know the basics of typing in commands and getting output, not much more.

1.1.1 R and RStudio

1.1.1.1 Installing R

Follow this link and install the right version for your operating system https://www.stats.bris.ac.uk/R/

1.1.1.2 Installing RStudio

Follow this link and install the right version for your operating system https://www.rstudio.com/products/rstudio/download/

1.1.1.3 Installing R packages in RStudio.

You'll need the following R packages:

1. devtools
2. atacR
3. DESeq

For simplicity, install them in that order.

To install `devtools`:

Start RStudio and use the `Packages` tab in lower right panel. Click the install button (top left of the panel) and enter the package name `devtools`, then click install as in this picture

To install `atacR`:

Type the following into the RStudio console, `devtools::install_github("TeamMacLean/atacr")`

To install `DESeq`:

Type the following into the RStudio console, `BiocManager::install("DESeq")`

Now you are done! Everything is installed ready for you to work with. Next we need to get the sample data

1.1.2 Sample reference genome and reads

You'll need this zip file of data: sample_data.zip which contains a reference genome and a set of paired end reads. Download it, extract the files and put them into a folder on your machine. I suggest something like `Desktop/align_tut`. This will be the directory we'll work from in the rest of the course.

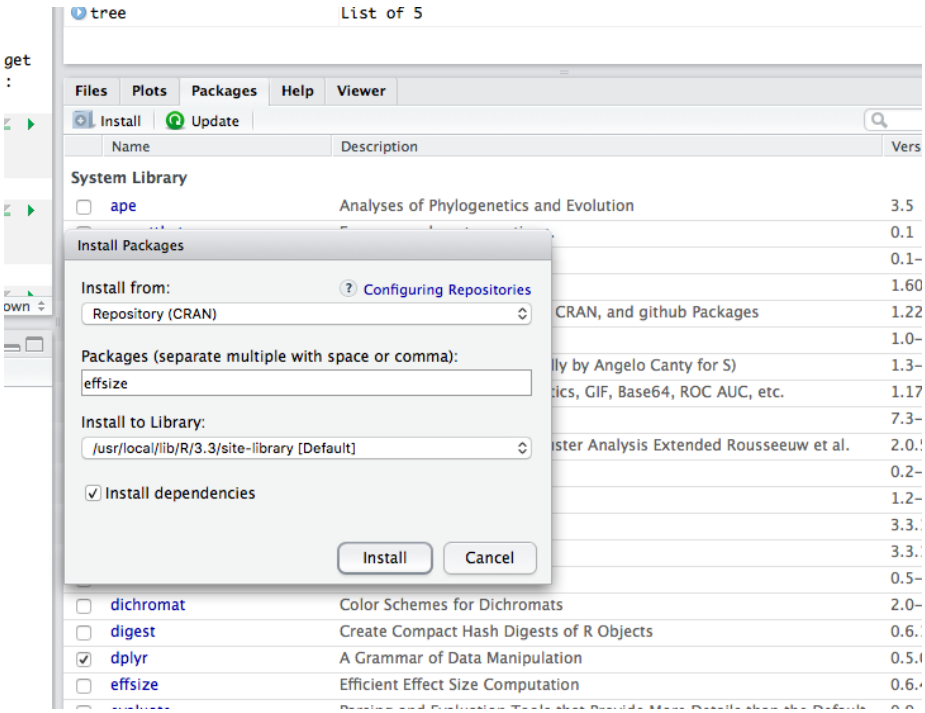


Figure 1.1: Installing Packages

That's all you need to do the lesson. If you have any problems getting this going, then ask someone in the Bioinformatics Team and we'll help.

Chapter 2

Counting Aligned Reads in Genomic Regions

2.1 About this chapter

1. Questions

- How do I calculate counts of reads at genes from my alignments?

2. Objectives

- Understand the basis for the gene region and read counting technique
- Understand what the count matrix represents
- Use the `make_counts()` function to make a count matrix

3. Keypoints

- Gene regions are designated by coordinates in GFF files
- A count matrix is a table-like object of reads that are found in a given genomic region
- The count matrix is the main object in a DESeq analysis

In this chapter we'll look at the fundamentals of read counting from a BAM file of aligned reads.

2.2 Counting the number of reads that have aligned to gene regions

The basis of quantitative RNAseq is working out how many of our sequence reads have aligned to each gene. In broad terms this is done by taking the

genomic coordinates of all the aligned reads (the start and end positions of the read's alignment on the reference genome) and cross-referencing them with the positions of the genes from a gene file. The resulting table is called a count matrix. See the figure below for a representation.

It is our aim in this section to create a count matrix from BAM files.

2.2.1 atacR

`atacR` was initially designed to help with the analysis of ATAC-Cap-seq data, a quite differen sort of data to RNAseq, but as with many bioinformatics pipelines, the first steps are quite common so we can make use of the neat way `atacR` handles the count matrix creation in the helpful function `make_counts()`

2.3 Preparing the input

We needs three things to work: the BAM files, a GFF file and a file of sample information.

2.3.1 The GFF file

GFF files are one way among many of describing the positions of genes on a genome. Here's a quick look at one.

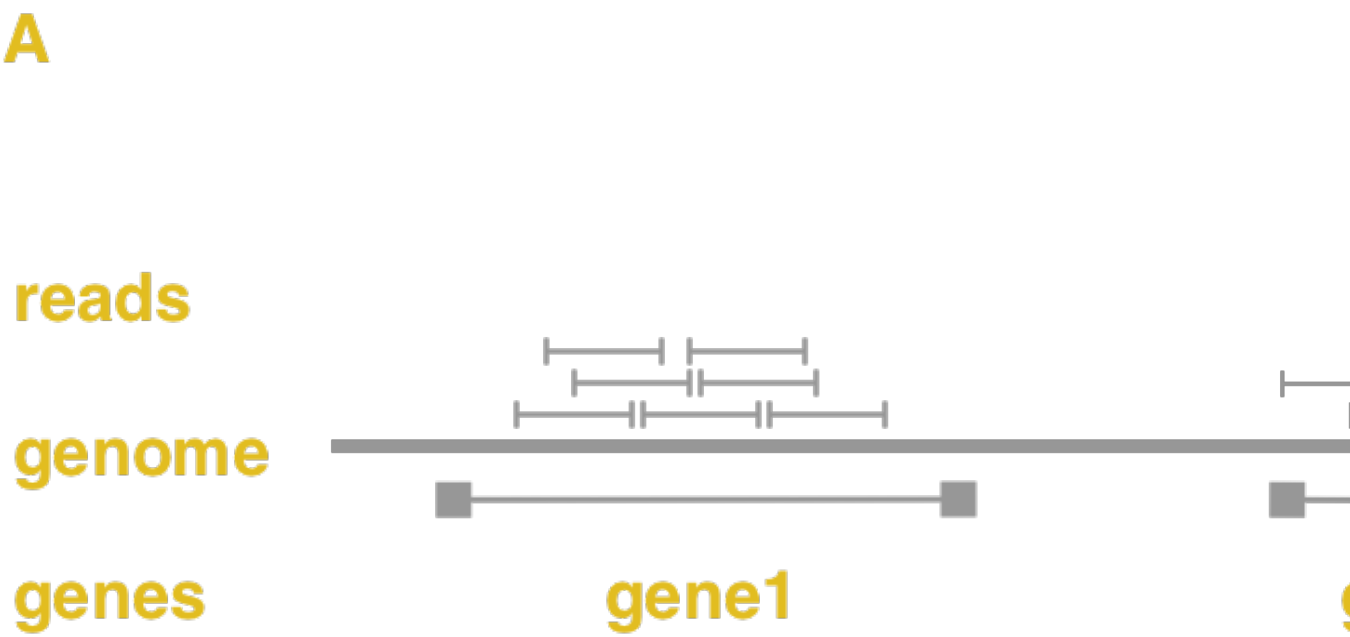
```
chr123  .  gene  1300  1500  .  +  .  ID=gene1
chr123  .  gene  1050  1500  .  +  .  ID=gene2
```

As you can see, it's a simple file with a gene represented on each line, by its chromosome (`chr123`), its start and end and its strand. The best thing about GFF files is that usually we can just download them from the relevant genome website. They tend to be freely available.

2.3.2 The Sample Information file

This file is a really simple file that references the BAM file of the alignment with the sample and replicate information. It has three columns: `sample_name`, `bam_file_path` and `treatment`. Here is an example.

```
## Parsed with column specification:
## cols(
##   treatment = col_character(),
##   sample_name = col_character(),
##   bam_file_path = col_character()
## )
```



B

	read count
gene1	7
gene2	4

Figure 2.1: A) Graphic of read alignment and gene position showing reads within genes. B) The equivalent count matrix that comes from this alignment

treatment	sample_name	bam_file_path
control	control_rep1	sample_data/control1/alignedSorted.bam
control	control_rep2	sample_data/control2/alignedSorted.bam
control	control_rep3	sample_data/control3/alignedSorted.bam
treatment	treatment_rep1	sample_data/treatment1/alignedSorted.bam
treatment	treatment_rep2	sample_data/treatment2/alignedSorted.bam
treatment	treatment_rep3	sample_data/treatment3/alignedSorted.bam

The `sample_name` column describes the treatment and replicate performed, the `bam_file_path` describes the place in which the BAM file for that sample is saved and `treatment` is the general name for the treatment that was used; this column is usually not unique when you have replicates.

2.3.3 The BAM files

The BAM files all come from a previously done alignment. The sample information file describes the place where they are kept and the sample they represent.

2.3.4 Sample files for this chapter

All the files are provided for you in the sample data you downloaded as `50_genes.gff` and `sample_information.csv` and in the folders containing BAM files. Feel free to examine them and look at how they relate to each other.

Once we have these files prepared, we can go on to use the `atacR` package to make the count matrix.

2.4 Running `make_counts()`

First we must load in `atacR`. Type the following into the R console.

```
library(atacr)
```

Now we can do the counting with `make_counts()`. Here's how to do it. Remember to properly describe the path to the files. The paths given here are correct if the files are in a folder called `sample_data` in the current working directory.

```
count_information <- make_counts("sample_data/50_genes.gff",
                                 "sample_data/sample_information.csv",
                                 is_rnaseq = TRUE
                                 )
```

The function should run and give no output. Note that it is important to set `is_rnaseq` to `TRUE` to tell the function to count appropriately. The results are saved in the `count_information` project.

2.5 Summaries and Diagnostic plots

With the counts computed we can do some diagnosis on the quality of the experiment.

We can see summary information with the `summary()` function

```
summary(count_information)
```

```
## ATAC-seq experiment of 2 treatments in 6 samples
##  Treatments: control,treatment
##  Samples: control_rep1,control_rep2,control_rep3,treatment_rep1,treatment_rep2,treatment_rep3
##  Bait regions used: 50
##  Total Windows: 99
##
##  On/Off target read counts:
##            sample off_target on_target percent_on_target
## 1   control_rep1          0     57733               100
## 2   control_rep2          0     66155               100
## 3   control_rep3          0     66122               100
## 4 treatment_rep1          0    100547               100
## 5 treatment_rep2          0    120325               100
## 6 treatment_rep3          0    107611               100
##  Quantiles:
##  $bait_windows
##     control_rep1 control_rep2 control_rep3 treatment_rep1 treatment_rep2
## 1%        149.48       294.60       241.12         228.70         102.98
## 5%        386.35       437.75       340.50         328.30         193.90
## 95%      2335.20      2438.20      2927.10        4445.90        6940.20
## 99%      3054.18      2752.19      3291.34        5234.33        9423.95
##     treatment_rep3
## 1%          116.50
## 5%          324.00
## 95%        4438.75
## 99%        6948.15
##
## $non_bait_windows
##     control_rep1 control_rep2 control_rep3 treatment_rep1 treatment_rep2
## 1%             0            0            0              0              0
## 5%             0            0            0              0              0
## 95%            0            0            0              0              0
```

```
## 99%            0            0            0              0              0
##     treatment_rep3
## 1%               0
## 5%               0
## 95%              0
## 99%              0
##
##  Read depths:
##             sample off_target on_target
## 1   control_rep1          0   1154.66
## 2   control_rep2          0   1323.10
## 3   control_rep3          0   1322.44
## 4 treatment_rep1          0   2010.94
## 5 treatment_rep2          0   2406.50
## 6 treatment_rep3          0   2152.22
```

It is long, but actually quite helpful. The first thing to note is that the words relate to ATAC-Cap-Seq, but in our context 'bait regions' just mean gene regions and non-bait just means intergenic regions. The 'on_targets' are read hits to genes, the 'off_targets' are read hits to intergenic regions.

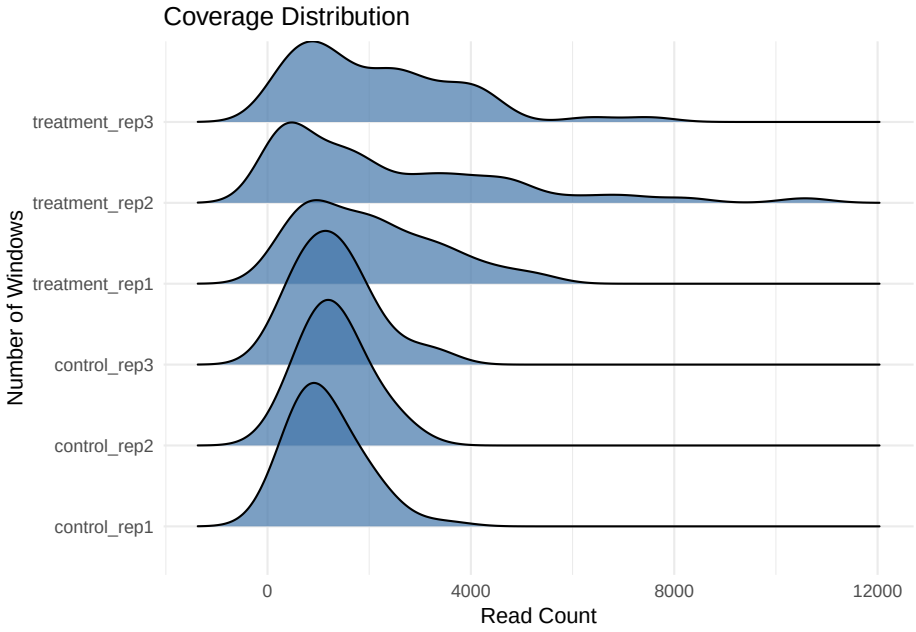
We can see that all the reads have hit in gene regions; that the read depth distribution of genes from the quantiles section give depths in the 1000 - 2000 range. This sort of summary is helpful when you're trying to work out whether the RNAseq is useful, lots of reads 'off target' is bad, as is low depth.

2.5.1 Gene Count Plots

We can see the distribution of depths over genes as a plot using the `plot_counts()` function

```
plot_counts(count_information, log10 = FALSE)
```

```
## Picking joint bandwidth of 488
```



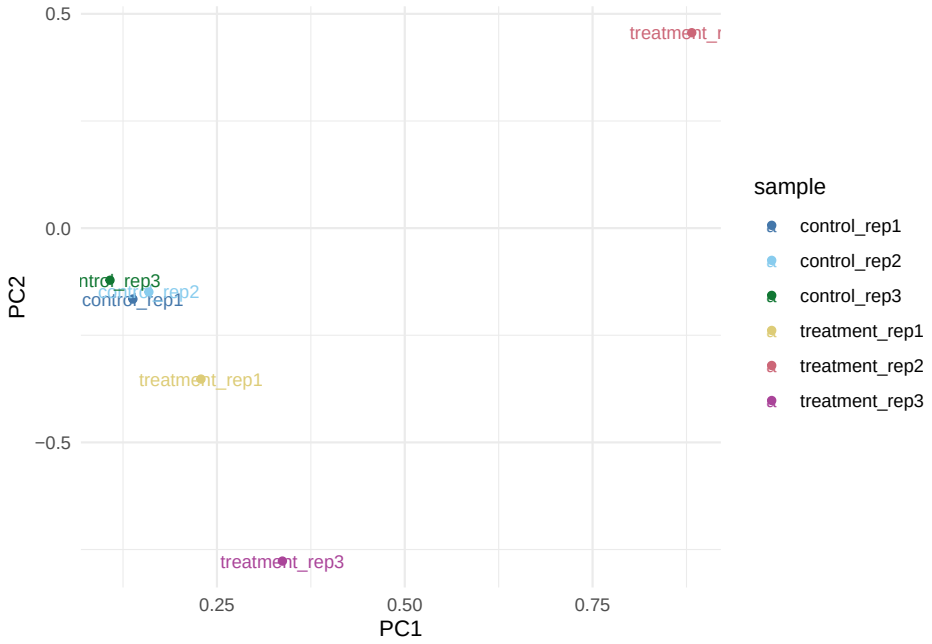
We can see that the mean count per gene (windows in `atacR`) is about 1000. The distributions in the treatment are bit more skewed than the controls.

2.5.2 Comparing Samples with PCA

It is common to examine the similarity of the samples to each other before moving on with analysis, ideally the similar samples will cluster together.

With `atacR` it is easy to perform a quick PCA analysis.

```
sample_pca_plot(count_information)
```



Here we can see that the control samples all cluster together, but the treatment samples are a bit more variable. We might want to normalise these counts later as a consequence.

2.6 Extracting and saving the count matrix

We now want to extract out the actual counts hiding inside the `count_information` object, we can do this with the `assay()` extractor function from the `SummarizedExperiment` package.

```
library(SummarizedExperiment)
raw_counts <- assay(count_information$bait_windows)

head(raw_counts)
```

```
##                          control_rep1 control_rep2 control_rep3
## Chr1:245989-249141                670          784          548
## Chr2:2195797-2200134             1104         1266          976
## Chr3:2454387-2458244              703          922          198
## Chr4:6650421-6657260             1865         1654         3207
## Chr5:11798344-11805414           1482         1266         1646
## Chr1:12893748-12901885           1186         1416         1458
##                          treatment_rep1 treatment_rep2 treatment_rep3
## Chr1:245989-249141                 1784           2558            368
```

```
## Chr2:2195797-2200134              358          1186          4436
## Chr3:2454387-2458244             1373          1167          1726
## Chr4:6650421-6657260             3533           703          2427
## Chr5:11798344-11805414           1258          1690          1864
## Chr1:12893748-12901885            834           594          2684
```

We can see the counts for each gene in each sample. Because `atacR` works on windows, the gene coordinates are given. We can replace the coordinates with gene names if we wish as follows

```
gene_names <- readr::read_csv("sample_data/gene_names.txt", col_names = FALSE )$X1
```

```
## Parsed with column specification:
## cols(
##   X1 = col_character()
## )
```

```
rownames(raw_counts) <- gene_names
head(raw_counts)
```

```
##           control_rep1 control_rep2 control_rep3 treatment_rep1
## AT1G01680          670          784          548           1784
## AT1G07160         1104         1266          976            358
## AT1G07920          703          922          198           1373
## AT1G19250         1865         1654         3207           3533
## AT1G32640         1482         1266         1646           1258
## AT1G35210         1186         1416         1458            834
##           treatment_rep2 treatment_rep3
## AT1G01680           2558            368
## AT1G07160           1186           4436
## AT1G07920           1167           1726
## AT1G19250            703           2427
## AT1G32640           1690           1864
## AT1G35210            594           2684
```

In this code chunk we load in the gene names from a file `gene_names.txt` using the `readr` package. Then we use the `rownames()` function to set the row names of `raw_counts`. This *is* a little cumbersome. Often you'll come across fiddly little things like this in bioinformatics analysis. If you ever get stuck feel free to come and chat to us in the bioinformatics team.

Now we can save the matrix to a file for re-use and importing into other programs. We'll do it in two ways 1) to a native R binary file that we can load straight in, 2) to a CSV file we can examine in programs including Excel.

2.6.1 Saving to an R RDS file

To save as a native R object, use `saveRDS()`, passing the filename you wish to save to.

```
saveRDS(raw_counts, "sample_data/raw_counts.RDS")
```

To save as a csv file use `write.table()`, again passing the filename you wish to save to.

```
write.csv( raw_counts, "sample_data/raw_counts.csv")
```

Now we can move on to using `DESeq`.

Chapter 3

Running `minimap2`

Running `minimap2` takes only one step. Assuming we've already `cd`'d into the directory with the reads and reference we can use this command

```
minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq > aln.sam
```

Try running that and see what happens… You should get an output file in the working directory called `aln.sam`. On my machine this takes just a few seconds to run.

Let's look at the command in detail.

3.1 The `minimap2` command and options

First we get this

```
minimap2
```

which is the name of the actual program we intend to run, so it isn't surprising that it comes first. The rest of the command are options (sometimes called arguments) telling the program how to behave and what it needs to know. Next up is this

```
-ax sr
```

which gives option `a` meaning print out SAM format data. And option `x` meaningwe wish to use a preset parameter set. The preset we wish to use comes after `x` and is `sr`, which stands for `short reads` and tells `minimap2` to use settings for short reads against a long genome. Next is this

```
ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq
```

which are the input files in the 'reference' 'left read' 'right read' order. Finally, we have

```
> aln.sam
```

which is the `>` output redirect operator and the name of an output file to write to. This bit specifies where the output goes.

So the structure of the `minimap2` command (like many other commands) is simply `program_name options input output`.

And this one command is all we need for a basic alignment with `minimap2`. We can now move on to the next step in the pipeline.

3.2 Further Reading

3.2.1 The > operator

The `>` symbol is actually not part of the `minimap2` command at all, it is a general shortcut that means something like 'catch the output from the process on the left and put it in the file on the right`. Think of the>`' as being a physical funnel catching the datastream! Because it's a general operator and not an option in a program, we can almost always use'>' to make output files. You'll see it pop up quite often

3.2.2 minimap2 further instructions and github

The commands given here for `minimap2` are just a small selection of what are available. You can see the user guide at GitHub

Chapter 4

Filtering Badly Aligned Reads

Once we have an alignment, the next step is often to throw out the reads that align badly or not in pairs as we we expect. To do this we need to look at the alignments and assess them one-by-one. We'll need first to have some understanding of the output from our alignment, in this case `aln.sam` a SAM format file.

4.1 SAM Format

Alignments are generally stored in SAM format, a standard for describing how each read aligned one-by-one. Each line carries the results for a single read. Let's examine a single reads alignment. Recall that we can look at one line in a file called `aln.sam` using `tail -n 1 aln.sam` (this gives the bottom line in the file). Running this prints the following

```
NC_011750.1_1004492_1005000_1:0:0_3:0:0_1869f   147 NC_011750.1 1004931 33  70M =   1004492 -509
```

On close inspection we can see this mess (which is only a single line) contains things like the read name, the position it maps to on the reference sequence, the read sequence, and lots of other strange things like `70M` and `de:f:0.0429`. The important thing to note is that these weird things are encoded quality information for this alignment, so we can - if we know how to manipulate those codes - select read alignment of the proper quality.

Thankfully the program `samtools` makes this easy for us.

4.2 samtools

We can accomplish read filtering with the following command.

```
samtools view -S -h -q 25 -f 3 aln.sam > aln.filtered.sam
```

Try running that and looking at the output file that is generated. You should have another SAM format file called `aln.filtered.sam` in your working directory.

Let's take a look at that command in detail

4.3 The samtools command and options

Straight away, the command seems to fit the familiar `program name options files` pattern. It starts with

```
samtools
```

which is the program name. Then we get the options

```
view -S -h -q 25 -f 3
```

The first option to `samtools` must be the name of the sub-program to run. There are lots of these as `samtools` is a suite of sub-programs. `view` is the option for working with alignments directly. The second option `-S` tells `samtools view` that we are handing it a SAM format file (soon we will hand it a different type) and `-h` tells it to show the header as well (each SAM file has a header that we sometimes don't want). The next two options are the important ones. `-q 25` will remove reads with a mapping quality (a measure of how well a read is aligned) lower than 25 (a reasonable score) and `-f 3` is a 'flag' a really complex way of encoding alignment attributes (see Further Reading for more details). The important thing is that `3` means `keep reads that are paired and whose pair is mapped too`.

At the end of the command is the input and output file information

```
aln.sam > aln.filtered.sam
```

which means the input file is our `aln.sam` and that the output should be redirected to `aln.filtered.sam`

4.4 Checking the filtering

As an exercise to show that we did filter stuff out lets compare the input `aln.sam` file with the output `aln.filtered.sam` file. Recall that `wc -l` will give us the number of lines in a text file. Run it like this, on both files at once

```
wc -l aln.sam aln.filtered.sam
```

I get this as output

```
  200002 aln.sam
  166905 aln.filtered.sam
  366907 total
```

The number of lines (alignments) in the filtered files is less than that in the unfiltered, so we can casually assume the command worked.

And that's all there is to getting the reads filtered. In real-life you have many options for filtering and you may choose to do it at other points (for instance, lots of RNAseq quantification programs will allow you to filter when you use them), but the process will be similar and take advantage of the same mapping quality and flag metrics you've been introduced to here.

4.5 Are we done?

On the face of it then, it looks like we've come to the end of what we intended to do - we did an alignment, and we've filtered out the poor ones. In practice though, we'll be dealing with many millions of reads, many files of many Gb size. This complicates the housekeeping we have to do, not the procedure we've learned *per se*, so before we jump to the HPC we need to look at that. That's the next chapter.

4.6 Further Reading

4.6.1 SAM Format

I only really alluded to the SAM format above, but there's a lot to it. This Wikipedia page gives a lot of detail.

4.6.2 Mapping Quality

A metric that describes how well overall the read aligned, it takes into account not just the alignment, but the nubmer of other possible alignments that were rejected. Consider that a read mapping well equally at a number of places in the genome cannot be said to be mapping well at all. Different aligners make arbitrary decisions about how to score such alignments. See this short summary for information on how it can be calculated.

4.6.3 Flags

The flags option is the most powerful way to describe a filter to `samtools view`, it is also really complicated. The number you pass (e.g `-f 3`) is calculated as a sum of lots of options. The way they're are described in the documentation is a bit more complex than I want to go into, but there are helpful web-apps that can simplify things - try this one

Chapter 5

Connecting Programs and Compressing output

Now that we've been through the whole alignment and filtering pipeline, let's look at the output. Specifically lets compare the sizes of the files we used. Recall that we can do that with `ls -alh`

On my folder I get this (some columns and files removed for clarity)

```
49M 29 Nov 10:46 aln.filtered.sam
59M 28 Nov 16:28 aln.sam
5.0M  2 Jul 15:04 ecoli_genome.fa
18M 28 Nov 15:53 ecoli_left_R1.fq
18M 28 Nov 15:53 ecoli_right_R2.fq
```

The file sizes are in the left-most column. Check out the relative size of the two read files (18M each) and the alignment SAM files (59M and 49M). The output file is much larger than the input. This has implications for storage when the files are really large (many GB) and there are lots of them. The disk space gets used really quickly. Consider also the redundancy we have - that `aln.filtered.sam` is the one we're interested in, not the `aln.sam` so it is taking up unnecesary disk space. It's easy to see that when you are doing a real experiment with lots of samples and hundreds of GB file size, you're going to eat up disk space. Also larger files take longer to process, so you're going to have a long wait. This has implications too when you get to later stages in the analysis

In this chapter we're going to look at a technique for reducing those housekeeping overheads and speeding things up.

5.1 BAM Files

BAM files are a binary compressed version of SAM files. They contain identical information in a more computer friendly way. This means that people can't read it, but it is rare in practice that you'll directly read much of a SAM file with your own eyes. Let's look at the command to do that

```
samtools view -S -b aln.filtered.sam > aln.filtered.bam
```

Again we're using `samtools view` and our options are `-S` which means SAM format input and the new one is `-b` means BAM format output. Our input file is `aln.filtered` and we're sending the output to `aln.filtered.bam`.

If we check the files with `ls -alh` now we get

```
9.2M 29 Nov 14:05 aln.filtered.bam
49M 29 Nov 10:46 aln.filtered.sam
59M 28 Nov 16:28 aln.sam
5.0M  2 Jul 15:04 ecoli_genome.fa
18M 28 Nov 15:53 ecoli_left_R1.fq
18M 28 Nov 15:53 ecoli_right_R2.fq
```

The BAM file is about a fifth of the size of the SAM file. So we can save space in this way. We have another trick up our sleeve though. We can connect together command lines, so that we don't have to create intermediate files - this reduces the number of files we have to save. We can do this by using something called pipes.

5.2 Connecting Program Input and Output With Pipes

Most command line programs print their results straight out without sending it to a file. This seems strange, but it adds a lot of flexibility. If also set up our programs to read in this output then we can connect them together. We can do this with pipes. The usual way to do this is to use the `|` operator. Let's look at a common example.

Here we'll use the command `ls` and `shuf` to see how this works. We know `ls` will 'list' our directory contents, `shuf` shuffles lines of text sent to it. If we use `|` in between we can connect the output of one to the other. Try running `ls` a couple of times to verify you get the same output both times and then try this a few times

```
ls | shuf
```

you should get different output everytime. The important thing to note is that `shuf` is doing its job on the data sent from `ls`, which sends consistent data every

time. We don't have to create an intermediate file for `shuf` to work from. The `|` character joing two commands is the key.

We can apply this to our `minimap2` and `samtools` commands.

5.3 From reads to filtered alignments in one step

So let's try reducing the original alignment pipeline to one step with pipes. We'll work in the BAM file bit later.

Simply take away the output file names (except the last one!) and replace with pipes. It looks like this

```
minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq | samtools view -S -h -q 25 -f
```

when you do `ls -alh` you should see the new `aln.filtered.from_pipes.sam` file, its size is identical to the file we generated when we created the intermediate `aln.sam` file, but this time we didnt need to, saving that disk space.

5.3.1 From reads to filtered alignments in a BAM file in one step

Let's modify the command to give us BAM not SAM, saving a further step. We already know that `samtools view` can output BAM instead of SAM, so lets add that option (`-b`) in to the `samtools` part.

```
minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq | samtools view -S -h -b -q 25
```

If you check the files with `ls -alh` now you should see that you have the new `aln.filtered.from_pipes.bam` file with no extra intermediate file and the smallest possible output file. Congratulations, you know now the fastest and most optimal way to make alignments and filter them.

5.4 Sorting BAM files

In practice a BAM file of alignments needs to be ordered with the alignments at the start of the first chromosome at the start of the file and the alignments on the end of the last chromosome at the end of the file. This is for computational reasons we don't need to worry about, but it does mean we need to do another sorting step to make our files useful downstream.

Because all the alignments need to be present before we can start we can't use the pipe technique above. So we use an input and output file. The command is `samtools sort` and looks like this.

```
samtools sort aln.filtered.from_pipes.bam -o aln.filtered.from_pipes.sorted.bam
```

Doing `ls -alh` shows a new sorted BAM `aln.filtered.from_pipes.sorted.bam` that contains the same information but is actually a little smaller due to being sorted. We can safely delete the unsorted version of the BAM file.

5.4.1 Automatically deleting the unsorted BAM

If the sorting goes fine, we have two BAM files with essentially the same information and don't need the unsorted file. We can of course remove this with `rm aln.filtered.from_pipes`. A neat space saving trick is to combine the `rm` step with the successful completion of the sort. We can do this by joining the commands with `&&`.

That looks like this

```
samtools sort aln.filtered.from_pipes.bam -o aln.filtered.from_pipes.sorted.bam && rm a
```

The `&&` doesn't connect the data between the two commands, it just doesn't let the second one start until the first one finishes successfully (computers have an internal concept of whether a command finished properly).

This means if the `samtools sort` goes wrong the `rm` part will not run and the input file won't be deleted so you won't have to remake it. This is especially useful later when we wrap all this into an automatic script.

5.5 Indexing the sorted BAM

Many downstream applications need the BAM file to have an index, so they can quickly jump to a particular part of the reference chromosome. This is a tiny file and we usually don't need to worry about it. To generate it use `samtools index`

```
samtools index aln.filtered.from_pipes.sorted.bam
```

Using `ls -lah` we can see a tiny file called `aln.filtered.from_pipes.sorted.bam.bai`, this is the index.

5.6 Further Reading

For a primer on some more aspects of `samtools` see this tutorial

Chapter 6

Automating The Process

We now know everything we need to do an alignment of reads against a reference in an efficient way. What's next is to consider that this process needs to be done for every set of reads you might generate. That's a lot of typing of the same thing over and over, which can get tedious. In this section we'll look at how we can automate the process to make it less repetitive using a script.

6.1 Shell scripts

Scripts that contain commands we usually run in the Terminal are called shell scripts. They're generally just the command we want to do one after another and saved in a file. We can then run that file as if it were a command and all the commands we put in the file are

Shell scripts must be a simple text file, so you can't create them in programs like Word, you'll need a special text editor. On most systems we have one called `nano` built into the Terminal.

6.1.1 Using `nano` to create a shell script

To open a file in `nano` type `nano` and the name of the file, if the file doesn't exist it will be created.

```
nano my_script.sh
```

Will create a file and open it. To save and exit type press `Ctrl` then `X` (thats what `^X` means in the help at the bottom. You can enter your script in here. Remember its not a word processor, its a Terminal text editor, so you have to use the mouse to move round and cutting and pasting is a bit clunky.

6.2 Creating a script that automates our alignment pipeline.

Let's enter our script into `nano`. We'll do it as we did in the earlier chapters, but we'll change file names to make it clear which files are coming from the script.

First, create a script called `do_aln.sh`

```
nano do_aln.sh
```

Once `nano` opens, add the following into it

```
minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq | samtools view -S -
samtools sort aln.script.bam -o aln.script.sorted.bam && rm aln.script.bam
samtools index aln.script.sorted.bam
```

That's all the steps we want to do. Use `Ctrl-X` to save the changes to the file.

6.3 Running the script

To run the script we use the `sh` command and the script name. Try

```
sh do_aln.sh
```

You should see progress from the script as it does each step in turn. When it's done you can `ls -alh` to see the new sorted BAM file from the script.

Congratulations! You just automated an entire analysis pipeline!

6.4 Running on different input files

So our script is great but the input filenames will be the same every time we run it meaning we'd need to go through the whole file and change them which is error prone. Also the output files are the same each time, meaning we could accidentally overwrite any previous work in there, which is frustrating. We can overcome this with a couple of simple changes in our script that make use of variables.

Variables are place holders for values that the script will replace when it runs. Consider these two commands

```
MY_MESSAGE="Hello, world!"
echo $MY_MESSAGE
```

Recall that `echo` just prints whatever follows it. Try running this, you get `Hello, world!` which shows that the process created a variable called `MY_MESSAGE` and stored the message in it. When used by a command the `$`

showed the command that it should use the message stored in the variable and printed `Hello, world!`. We can use this technique in our scripts. Note the command `MY_MESSAGE="Hello, world!"` must not have spaces around the equals sign.

Now we can expand our script to take advantage. Look at this script.

```
LEFT_READS="ecoli_left_R1.fq"
RIGHT_READS="ecoli_right_R2.fq"
REFERENCE_SEQUENCE="ecoli_genome.fa"
SAMPLE_NAME="ecoli"

minimap2 -ax sr $REFERENCE_SEQUENCE $LEFT_READS $RIGHT_READS | samtools view -S -h -b -q 25 -f 3
samtools sort $SAMPLE_NAME.bam -o $SAMPLE_NAME.sorted.bam && rm $SAMPLE_NAME.bam
samtools index $SAMPLE_NAME.sorted.bam
```

Right at the top we create a variable for each of our read files (`LEFT_READS` and `RIGHT_READS`), our reference files (`REFERENCE_SEQUENCE`) and a unique sample name (`ecoli`). These variables get used whenever we need them, saving us from typing the information over and over. The practical upshot of this being that we only need to change the script in one place every time we reuse it for a different sample and set of reads.

Now try this out. Save the new script in a file called `do_aln_variables.sh` and run it as before with `sh do_aln_variables.sh`. When it's run you should see an output called `ecoli.sorted.bam`.

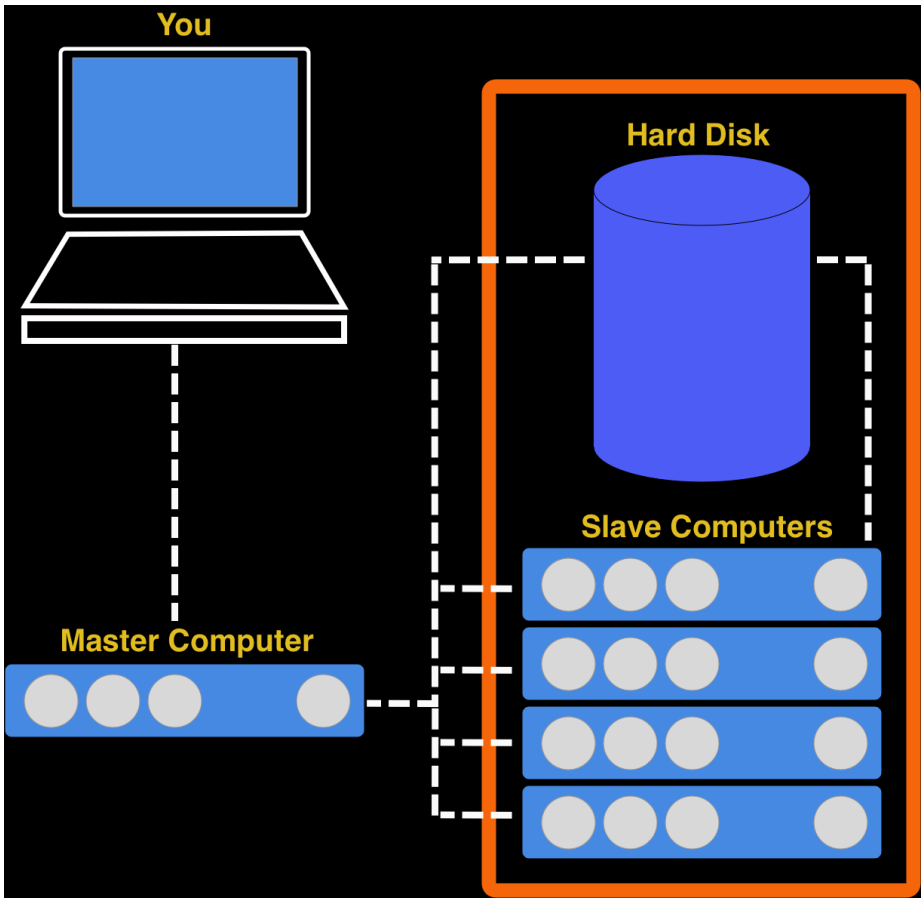
Chapter 7

Running an alignment on the HPC

In this chapter we'll look at how to run the alignment on an HPC cluster. First, we need to know a few things about that HPC before we can begin.

7.1 An HPC is a group of slave computers under control of a master computer

Most of the computers in an HPC cluster are really just there to do what one other computer tells them. They cannot be contacted directly (by the normal user) and they have very little software already on them. As the user you must get a master computer to tell them what to do. A key thing about an HPC cluster is that all the computers share one massive hard-drive. Look at the diagram below.



It shows the computer you are working at, the master computer and it's relation to the slave computers and the hard disk. Note that there is no way for you to contact the slaves directly, even though the slaves (or more properly 'worker nodes') are where the actual job runs. So the workflow for running an HPC job goes like this

1. Log into master (more usually called 'submission node')
2. Prepare a task for the submission node to send to the nodes
3. Submit the task to the submission node
4. Wait for the submission node to send the job to the worker nodes
5. Wait for the worker nodes to finish the job

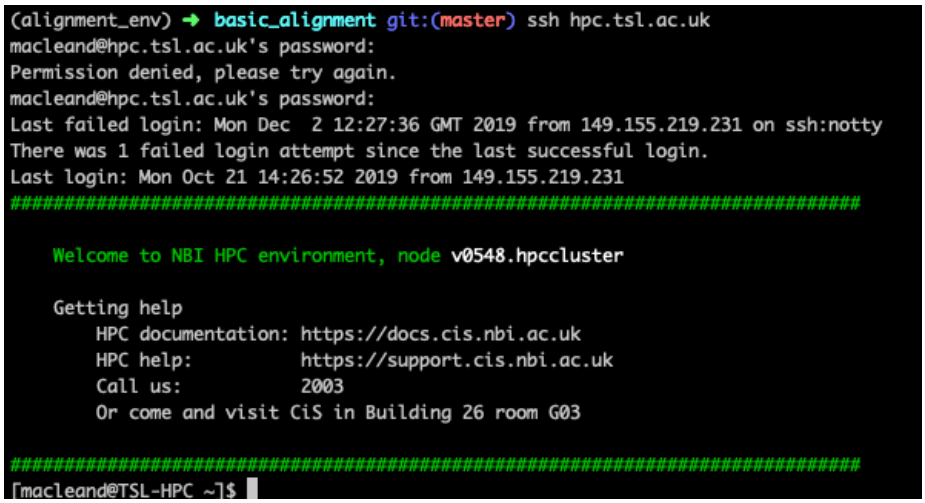
In the rest of this chapter we'll look at how to do these steps

7.2 Logging into the submission node

This is pretty straightforward, you need to use the `ssh` command to make a connection between your computer and the submission node. The TSL submission node has the address `hpc.tsl.ac.uk` so use this command

```
ssh hpc.tsl.ac.uk
```

You'll be asked for a user name and password, it's your usual NBI details. When it's done you should see something like this



This terminal is now working on the submission node (you can tell from the prompt `macleand@TSL-HPC`)

7.3 Preparing a job

To run a job we need to create a submission script. `nano` is available on the submission node, so we can use that. But what goes inside? Here's a typical one.

```
#!/bin/bash

#SBATCH -p tsl-short
#SBATCH --mem=16G
#SBATCH -c 4
#SBATCH -J alignments
#SBATCH --mail-type=begin,end,fail
#SBATCH --mail-user=dan.maclean@tsl.ac.uk
#SBATCH -o alignments.%j.out
```

```
#SBATCH -e slurm.%j.err

source minimap2-2.5
source samtools-1.9

srun minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq | samtools view
```

Not much of this is going to be familiar, but it isn't complicated.

The first line of this file `#!/bin/bash` is one that should always be there. Always put it in and never worry about it again. It just tells the computer this file is a script.

7.3.1 The `#SBATCH` options

The second block of statements, all beginning `#SBATCH` are the resource options for the job. It tells the submission node what resources the job needs to run. These need to go at the top of the script. Let's look at them individually.

7.3.1.1 `#SBATCH -p`

This tells the submission node which queue (or partition in the jargon) the job should run on. We have three basic partitions `tsl-short`, `tsl-medium` and `tsl-long`. The main difference is that jobs that run for a short time shouldn't be held back by jobs that run for ages, so the submission node uses this to run all of its jobs optimally.

7.3.1.2 `#SBATCH -c`

The number here tells the machine how many CPU's (processors) to use. Most tools will be able to make use of more than one and will run faster as a consequence. The job (usually) won't fail if you get this wrong, but it will take longer to start as it waits for more CPU's to come free.

7.3.1.3 `#SBATCH --mem=`

This tells the submission node how much memory your job will need to run. Jobs that exceed their stated memory by too much are killed. REquestiing the lowest possible memory means your job will be executed more quickly. Memory is requested in units of `G` gigabytes, usually.

7.3.1.4 #SBATCH -J

This is a helpful little name for you to identify your jobs with. eg `#SBATCH -J my_jobs`

7.3.1.5 #SBATCH --mail-type=

These are the times during the job that the submission node will email you to let you know of a status change in your job. Always use this option as presented for quickest information.

7.3.1.6 #SBATCH --mail-user

This is simply the address your update emails will be sent to.

7.3.1.7 #SBATCH -o and #SBATCH -e

These are the names of files that output and errors will be sent to. On a long running process the output can get long so it goes to a file, not the email. The weird `%j` is a job ID number that uniquely identifies the job.

7.3.2 The source options

The next lines all begin with the word `source` followed by some software name. No software is loaded into the worker nodes by default, so we need to say which tools we want to use. Do this by using the `source` keyword followed by the software name, e.g `source BLAST-2.2.2`. Many versions of the same tool are available on the HPC, and are differentiated by the version number at the end. You can see which software is available to source by typing `source` then hitting the tab key twice. It should give a very long list of tools.

7.3.3 The srun command

Finally, we get to the actual commands we want to run. This is exactly as we did before but with the command `srun` in front.

7.4 Submitting with sbatch

All of this information should be saved in a single script. You can call it what you want, but use the extension `.sh`. Once you've got this script, you can ask

the submission node to add your job to the queue with `sbatch`. This doesn't go in the script, it goes on the command-line, so if you'd added all the details above to a file called `do_my_alignments.sh` you can submit it by typing `sbatch do_my_alignments.sh`

7.5 Checkout tasks

So that's all you need to know to submit a job. Let's test how that works by creating a simple job and running that. Then we'll try a bigger alignment job. These are

1. Create a job using a submission script that runs this command `date`. Check what the `date` command does on the command line. Note that it runs very quickly (is a short job) and uses very little memory (< 1G) and only needs one CPU.
2. What happened to the output? Check the contents of your directory when the job is done and examine the new files (`less` is useful for this).
3. Explicitly create an output file by running this command through the HPC instead `date > date.txt`. What is the contents of the folder now? What effect did explicitly naming an output file have. What is the `slurm_xxxx.out` file for?
4. Run an alignment job using the information we learned in the earlier chapters. The reference file `ecoli_genome.fa`, `ecoli_left_R1.fq`, `ecoli_right_R2.fq` are available in the HPC filesystem in the folder. `/tsl/data/reads/bioinformatics/tutorial/alignments/`

7.6 Further Reading

You can see more information about the cluster submission system and options at the CiS documentation site