Reduced representation sequencing exercises

Mark Ravinet

mark.ravinet@ibv.uio.no

Contents

Reference-mapped RAD analysis

Introduction

For the first session today, we will try our hand at analysing RAD data when we have reference genome available. First we will map our data to the reference genome, then we will call SNPs, perform some simple filtering and quality checking steps before plotting genome-wide FST estimates.

We are going to be working on double-digested RAD seq data from a Canadian population of three-spined sticklebacks (*Gasterosteus aculeatus*). The Little Campbell stream is famous in stickleback evolutionary biology because it is one of the first places that marine and freshwater stickleback morphs were explained in detail (Hagen & McPhail 1967).

---

1. Getting started

As you will no doubt be aware, running NGS analyses locally on your computer is often not feasible! As with the majority of all tutorials throughout this course, we will be working on the EVOP server.

Logging in and setting up

To login, use `ssh` or a similar shell client, like so:

```
ssh k00319250@evop2018login.imp.fu-berlin.de
```

Once you are logged in, familiarise yourself with your surroundings. You should have already learned some Unix commands by now but use `ls` to see everything present in your home directory. We are going to do all our work in a special directory which we need to create. Use the following commands to do so:

```
cd ~
mkdir ref_rad
cd ref_rad
```

Get access to some useful scripts

For most of our work today, we will follow my lead during the class. However, at some point, we will also use some scripts I have previously prepared for you. I have hosted these on a Github repository[1] specifically for the course.

Getting them is very straightforward. Just use the following command:

```
git clone https://github.com/markravinet/EVOP_2018 mark_scripts
```

This will create a `mark_scripts` directory in your home folder with some useful scripts inside.

Why have I chosen to use a git repository for this purpose? Well, first of all, it means that you can download everything you need with a single command. Secondly, it is very easy for me to update this repository if I want to make something new available to you or if I want to update the scripts.

Your scripts directory is a clone of the one I have online - if you want to make sure it is up-to-date, simply enter it and do the following:

[1] https://github.com/markravinet/EVOP_2018

```
cd mark_scripts
git pull
```

You should either see a dialogue saying new files are being downloaded **OR** that the directory is up-to-date.

Making your life easier

Before we proceed, we are going to set an environmental variable that will make things easier throughout the rest of this tutorial. There will be times when I ask you to copy things from a directory and this will mean less typing and hopefully, less confusion.

```
TDIR=/opt/evop/public/NGS_NonModelOrganisms/ref/
```

`TDIR` is our directory for this tutorial. Confirm it worked by doing the following:

```
ls $TDIR
```

You will soon see why this is helpful!

What are fastq files?

Before we can actually do any analysis, we are going to need to get some data. Typically for an analysis like this, you will recieve sequencing data as a large fastq[2] file which you need to demultiplex (i.e. breakdown into files for separate individuals). For the sake of convenience today, all have already been demultiplexed and quality checked, so we can get started straight away. Let's get at the data. We will copy it into our current directory

```
cp $TDIR/raw_reads.tar.gz ./
```

This is a gzipped tarfile - in other words it is a compressed archive of data. Before we can look at the data, we need to extract it. Do that like so:

```
tar -vxf raw_reads.tar.gz
```

This will create a directory in your `ref_rad` directory. Move in to the newly created `raw_10K` directory.

```
cd raw_10K
```

You should see a list of files that all end with **fastq.gz**. These are gzipped fastq files - raw sequencing data for each individual in the study. Use the following command to take a closer look at the first individual:

```
zcat LCM10_1.fastq.gz | head -4
```

You should see the first full lines of the first fastq file (**note:** here the file is truncuated - you will see more bases).

[2] https://en.wikipedia.org/wiki/FASTQ_format

```
@MG00HS09:622:C695WACXX:7:1101:7183:1964 1:N:0:CTGGTT
TTTCAAATGGTTTTCCCCTCGTCAAGTAAGAGAACACAAAGGAATGTTGAGGA
+
CCCFFFEFHHFHHJJIJJJJJIJJJJHIIJJJJJJJJJJIJJGIJIJJJJII
```

What do these lines mean?

- **Line 1:** Sequence id: this is usually the Illumina machine identifier and cell coordinate. The string after the space here tells us this is the first (forward) read of a paired-end sequencing project. The final 6 basepairs are the library code
- **Line 2:** This should be familiar! This is the raw sequence
- **Line 3:** A spacer line - contains only '+'
- **Line 4:** Quality scores - here they are encoded as Phred33

Manipulating fastq files

An important task for any budding bioinformatician is to calculate how many reads are present for each individual. Use the following code to do this:

```
zcat LCM10_1.fastq.gz | wc -l
```

Here we have piped the file (using `zcat`) to `wc`which with the `-l`flag will count the number of lines in the field. Remember that there are four lines per read from above so it should be easy to calculate the number of reads present here. Alternatively if you are lazy (like me!) you could use another tool like `awk` to calculate the exact number of sequences

```
zcat LCM10_1.fastq.gz  | wc -l | awk '{print $1/4}'
```

Usually, a single fastq file will contain millions of reads. To make our practical today more manageable, I have subsampled the dataset to include only the first 10k reads for each individual. If you are running through a pipeline for the first time, it is often worth doing this to make sure everything works properly and then repeating it with the full size data.

Use `ls -lah` to have look at the contents of the directory again. You will now see the size of the files; without subsampling, these would be >500 mb each and be much more difficult to manage in a short period of time.

You should also note the name of the files. We have two types of sample here, those starting with `LCM` and those with `LCS` which are Little Campbell marine and stream fish respectively. Also note how there are two files for each individual, this is because of paired-end sequencing. Those with `_1` are forward reads, those with `_2` are reverse reads.

---

2. Aligning to the reference genome

Finding the reference genome

We are going to align our data to an updated version of the stickleback reference genome, described in Roesti *et al.* (2013).

You will find a copy of this reference genome in my lecture files directory. Copy it your own directory using the following code.

```
cd ~/ref_rad
cp $TDIR/Gac_ac_rm_Roesti.fa ./
```

In order to align to the genome, we are going to use `bowtie2` which is a nice and very fast aligner. See here[3] for more details on bowtie2.

Indexing the reference genome

Before we can actually perform an alignment, we need to index the reference genome we just copied to our home directories. This essentially produces an index for rapid searching and aligning. Use the following command:

```
bowtie2-build Gac_ac_rm_Roesti.fa gac
```

For our purposes today, it takes too long to run this command so once you've initated the command, press `ctrl + c` to kill it. Note that the second part of this command, `gac` is the name for index of our reference genome. Use `ls` to see that you have created a number of files with `gac` as the prefix.

To save time, we will copy some already built indexes to our home directory

```
cd ~/ref_rad
cp $TDIR/gac* ./
```

Performing a paired end alignment

Now we are ready to align our sequences! To simplify matters, we will first try this on a single individual. First create a directory to hold our aligned data:

```
cd ~/ref_rad
mkdir align
```

As a side note, it is good practice to keep your data well organised like this, otherwise things can become confusing and difficult in the future.

To align our individual we will use `bowtie2`. You might want to first have a look at the options available for it by calling `bowtie2 --help`

We align paired-end reads like so:

[3]http://bowtie-bio.sourceforge.net/bowtie2/index.shtml

```
bowtie2 -x ./gac -1 ./raw_10k/LCM10_1.fastq.gz \
-2 ./raw_10k/LCM10_2.fastq.gz \
-N 0 -p 4 \
-S ./align/LCM10.sam
```

Be prepared, this will take a a minute or two. In the meantime, what have we done here?

- `-x` specifies the reference genome index to use
- `-1` tells bowtie2 where the forward reads are
- `-2` tells bowtie2 where the reverse reads are
- `-N` is the number of mismatches allowed in the read seed
- `-p` is the number of threads to run the command on
- `-S` is the output - here we are writing it as a SAM file.

Once your alignment has ended, you will see some alignment statistcs written to the screen. We will come back to these later.

SAM files

Lets take a closer a look at the output. To do this we will use `samtools`. More details on `samtools` can be found here[4]

```
cd align
samtools view -h LCM10.sam | head
samtools view LCM10.sam | head
```

This is a SAM file - or sequence alignment/map format. It is basically a text format for storing an alignment. The format is made up of a header where each line begins with `@`, which we saw when we used the `-h` flag and an alignment section.

The header gives us details on what has been done to the SAM, where it has been aligned and so on. We will not focus on it too much here but there is a very detailed SAM format specification here[5].

The alignment section is more informative at this stage and it has at least 11 fields. The most important for now are the first four. Take a closer look at them.

```
samtools view LCM10.sam | head | cut -f 1-4
```

Here we have:

- The sequence ID
- The flag - these have various meanings, 0 = mapped, 4 = unmapped
- Reference name - reference scaffold the read maps to
- Position - the mapping position/coordinate for the read

[4]http://www.htslib.org/doc/samtools.html
[5]https://samtools.github.io/hts-specs/SAMv1.pdf

Note!! The other fields are important, but for our purposes we will not examine them in detail here.

Now, lets look at the mapping statistics again:

```
samtools flagstat LCM10.sam
```

This shows us that a total of 200k reads were read in (forward and reverse), that around 79% mapped successfully, 64% mapped with their mate pair, 10% were singletons and the rest did not map.

BAM files

One problem with SAM files is that for large amounts of sequence data, they can rapidly become HUGE in size. As such, you will often see them stored as BAM files (Binary Aligment/Mapping format). A lot of downstream analyses require the BAM format so our next task is to convert our SAM to a BAM.

```
samtools view LCM10.sam -b -o LCM10.bam
```

Here the `-b` flag tells `samtools` to output a BAM and `-o` identifies the output path.

You can view bamfiles in the same way as before.

```
samtools view LCM10.bam | head
```

Note that the following will not work (although it does for SAM) because it is a binary format

```
head LCM10.bam
```

Before we can proceed to more exciting analyses, there is one more step we need to perform - sorting the bamfile. Most downstream analyses require this in order to function efficiently

```
samtools sort LCM10.bam -o LCM10_sort.bam
```

Once this is run, we will have a sorted bam for downstream analysis! However as you may have noticed, we have only performed this on a single SAM output so far... what if we want to do it on multiple individuals? Do we need to type all this everytime?

Using bash scripts to work on multiple files

Rather than typing out each command, we can use a bash script. For ease today, I have provided one for you and you downloaded it earlier from my github repository. Copy it to your current directory and take a look:

```
cd ~/ref_rad
cp ~/mark_scripts/reference_mapping/align_sort.sh ./
less align_sort.sh
```

This looks complex but it isn't really. The first line is:

```
#!/bin/sh
```

Which just declares the shell - it tells the script to run in bash. Note that # indicates this line is a comment.

We declare an array like so:

```
INDS=($(for i in ./raw_10k/*.fastq.gz; do echo $(basename   ${i%_*}); done | uniq))
```

This basically produces a list of file names to use inside the bash for loop.

Next is the for loop - it is identical to the commands we ran previously except that it takes each individual in the list of individuals and runs it for each of them.

Explaining this bash file in depth is outside of the scope of this exercise but it is worth learning how to write a script like this yourself as it will make your life much much easier.

So now that you have a rough idea of what the bash script does, you can run it like so:

```
bash align_sort.sh
```

Once all the reads are aligned, sorted and converted to BAMs, we are ready to move on to the next step.

Note! If this is taking too long or you are running out of time, you can copy bams I have already aligned and sorted into your home directory. Just do the following

```
cd ~/ref_rad/align/
cp $TDIR/align/*_sort.bam ./
```

---

3. Calling variants

If everything has worked correctly up to this point, we now have a set of sequence reads that are aligned to our reference genome. What we want to do now is to call variants from these alignments.

Disclaimer: There are many ways to perform variant calling for RAD-seq data - bcftools, GATK, Stacks - this is just an example. You should evaluate which tool will be best for your purposes with your own data.

To call variants today, we will use `bcftools` which is designed by the same team behind `samtools` - they are part of the same pipeline. `bcftools` is itself a comprehensive pipeline and produces a variant call format (vcf) that is used in many downstream analyses.

Indexing the reference... again

The first thing we need to do is index our reference genome again. This actually needs to be done with `samtools`. Return to the home directory and perform the following actions

```
cd ~/ref_rad/
samtools faidx Gac_ac_rm_Roesti.fa
```

Let's take a closer look at that index.

```
head Gac_ac_rm_Roesti.fa.fai
```

You don't need to worry about this in great deal but for clarity, the columns are: chromsome name, chromosome length, offset of the first base, fasta line length, fasta line length + 1.

Setting environmental variables

With this step done, we are nearly ready to call variants. To keep the command line clear, we are going to set an important environmental variable like so:

```
REF=~/ref_rad/Gac_ac_rm_Roesti.fa
```

What exactly are we doing here? Well we are declaring a variable in the `bash` environment. So from now on `$REF`will refer to the path of the reference genome. This means rather than typing the whole path, we can use `$REF` to access it. To see a variable, type `echo $REF` (or whatever the variable is called). This is a really useful thing to incorporate in to bash scripts to make them easy to use for different projects.

Calling variants

To call variants, we will first use the `samtools mpileup` tool to pileup our BAM files. What does this mean exactly? Well we will take all reads at a given position and call variants from the reads covering that position. We need to do this for all individuals.

After performing the pileup, we than pass the output to `bcftools call` which will actually call variants. To see the options available to each part of the pipeline, just type their names into the command line.

Rather than perform each step previously as we did before, here we will use pipes - `|` - to pass data from each part of the pipeline to the next. Rather than explaining every point now, it's best we just perform the calling and break it down later.

```
cd ~/ref_rad/
samtools mpileup -Rug -t DP,AD -C 50 -f $REF \
./align/*_sort.bam | bcftools call -f GQ,GP \
-vmO z -o ./stickleback.vcf.gz
```

While this is running, let's go through the options and get an idea of what we did.

For `samtools mpileup`:

- `-R` - ignores read group data - so each BAM is an individual
- `-u` - Do not compress the output
- `-g` - Generate genotype likelihoods in BCF format (don't worry too much about this)
- `-t` - output tags - here we are telling `mpileup` to output depth and depth per variant data (this will be clearer later)
- `-f` - the location of the reference sequence

For `bcftools call`:

- `-f` - format fields for the vcf - here they are genotype quality and genotype probability
- `-v` - output variant sites only - i.e. ignore non-variant parts of the reads
- `-m`- use bcftools multiallelic caller
- `-O`- specify the output type, here it is `z` - i.e. gzipped (compressed) vcf
- `-o` output path

In essence, we have piled up each read against each genome position and then called variants where there are enough alternative reads to support the presence of a variant.

If the previous step worked, you should now have a nice vcf file available to view. If not, you can copy one I made for you earlier like so:

```
cd ~/ref_rad/
cp $TDIR/vcf/stickleback.vcf.gz ./
```

Exploring vcf files

First, a bit of housekeeping. Make a vcf directory and move the vcf into it.

```
cd ~/ref_rad
mkdir vcf
mv stickleback.vcf.gz ./vcf
cd vcf
```

vcf stands for 'variant call format' and is a standard format used for variant calling and in population genomics. Again a detailed specification can be found online[6]. It can take a bit of getting used to but it is widely supported and very useful. Let's have a look at the vcf.

```
bcftools view -h stickleback.vcf.gz
```

A lot of information will flash by - this is the vcf header. Like the SAM file, the header contains information on what has been done to the vcf. The last line is particularly important as it shows what each field in the main body of the vcf is and it also gives the individual names. Try the following:

```
bcftools view -h stickleback.vcf.gz | head
bcftools view -h stickleback.vcf.gz | tail
```

Here `-h` means show me only the header. You can also see `-H` to see only the raw calls. For some more information on the vcf format, see here.

A nice feature of vcf files is that you can access almost any part of the genome you are interested in. To do this though, you need to index the vcf first. Simply do the following

```
bcftools index -t stickleback.vcf.gz
```

Let's see what variants are present at the start of chrIV:

```
bcftools view -H -r chrIV stickleback.vcf.gz | head
```

Here the `-r` flag specfies the region of the genome to examine. We can achieve the same effect by specifying the base pair coordinates

```
bcftools view -H -r chrIV:100000-200000 stickleback.vcf.gz
```

Now you're probably wondering what exactly all this data actually means. Let's explore in a bit more detail.

```
bcftools view -h stickleback.vcf.gz | tail -1 | cut -f 1-9
bcftools view -H stickleback.vcf.gz | head -1 | cut -f 1-9
```

These are the first 11 fields of the vcf and they are always present. What do they mean?

- `CHROM` - chromosome or scaffold id from the reference genome
- `POS` - base pair reference position
- `ID` - SNP id - blank in this case
- `REF`- Reference base - A,C,G,T or N. More than one indicates an indel
- `ALT` - Alternaate base - the alternate base called as a variant
- `QUAL` - Phred quality score for alternate base call (site not individual)
- `FILTER` - filter status - if PASS then all filters passed
- `INFO` - additional info on each site - explanation stored in header
- `FORMAT`- the format for the genotype fields

[6]https://samtools.github.io/hts-specs/VCFv4.2.pdf

There is a lot of information here! Don't worry too much if it doesn't make perfect sense immediately - familiarity with this format will come with time and experience.

Let's look at the the first site again in detail:

```
bcftools view stickleback.vcf.gz | grep -m 1 -A 1 "#CHROM" | cut -f 1-7
```

You should see this:

```
#CHROM  POS ID  REF ALT QUAL    FILTER
chrI    187111  .   C   A   159 .
```

Which means we have a variant at 187111 on chr1. The reference base is C, alternate base is A. There are no filters.

Have a look at the info field with the following code

```
bcftools view stickleback.vcf.gz | grep -m 1 -A 1 "#CHROM" | cut -f 8
```

You can find out what these mean by grepping the header.

```
bcftools view -h stickleback.vcf.gz | grep "INFO"
```

So for example, DP here means the raw read depth for the entire site.

What about the actual genotype information? Firstly it is wise to look at the format here.

```
bcftools view stickleback.vcf.gz | grep -m 1 -A 1 "#CHROM" | cut -f 9
```

To investigate what these mean, grep the header again.

```
bcftools view -h stickleback.vcf.gz | grep "FORMAT"
```

Now let's take a look at the call for the first individual.

```
bcftools view stickleback.vcf.gz | grep -m 1 -A 1 "#CHROM" | cut -f 10
```

This will return:

```
./align/LCM10_sort.bam
1/1:36,3,0:1:0,1:41,3,2:3
```

The first line is obviously the name of the sample, then we have the genotype. Here `0` always denotes the reference, `1` the alternate base so we can see this individual is heterozygote for the `REF` and `ALT` bases. We will ignore the genotype likelhood for now. You can see there is 1 read in total covering this site and no reads for the variant - meaning this genotype call is obviously wrong.

Is there an easier way to view genotypes in this way? Yes there is - using the `bcftools query` utility.

This allows you to look at the genotypes like so:

```
bcftools query -f '%CHROM\t%POS\t%REF\t%ALT[\t%GT]\n' stickleback.vcf.gz | head
```

You can also translate them into actual basecalls. Try this:

```
bcftools query -f '%CHROM\t%POS\t%REF\t%ALT[\t%TGT]\n' stickleback.vcf.gz | head
```

The `bcftools query` utility is very powerful and a useful tool to know about for file conversion in your own work.

Looking at the genotypes, you will see they are mostly missing. One of the reasons for this is the fact we used such a tiny dataset. So in the next section we will look at a proper set of variant calls from multiple individuals.

---

4. Filtering variants

Get some real data

At this point our subsetted dataset kind of breaks down as we do not have enough reads or coverage to call a decent variant set

To rememdy this, we are going to use a vcf I have prepared using a **full dataset of 40 individuals**. To get the full dataset into your directory, use the following commands:

```
cd ~/ref_rad/vcf
cp $TDIR/vcf/stickleback_full_unfiltered.vcf.gz* ./
```

This will copy both the vcf and its index to your directory.

How many unfiltered variants?

One thing we didn't check yet is how many variants we actually have. Each line in the main output of a vcf represents a single call so we can use the following code to work it out:

```
bcftools view -H stickleback_full_unfiltered.vcf.gz | wc -l
```

That's a substantial number of calls! But chances are many of them are not useful for our analysis because they occur in too few individuals, their minor allele frequency is too low or they are covered by insufficient depth.

Up until now, we have not filtered any sites at all. I personally prefer to keep as many variants in the analysis as possible and then only filter at the final step. I also prefer to do a bit of initial analysis to get an idea of how to set the filters. I will give one example of that here - for sequencing depth.

To perform filtering and to generally get an idea of what sort of information our vcf contains, we are going to use `vcftools`, a very useful and fast program for handling vcf files[7].

[7] https://vcftools.github.io/examples.html

Getting an idea of how to set filters

How do you set filters for your dataset? This isn't actually an easy question to answer but there are a few basics you should always filter on - depth, genotype quality and minor allele frequency.

Ideally you should use `vcftools`to get an idea for yor data - plot the distribution of allele frequencies or genotype depth for example. We don´t have time for that today so instead we will use some approximate rules that should provide decent results.

- **Depth:** You should always include a minimum depth filter and ideally also a maximum depth one too. Minimum will prevent false positive calls (like the one we saw earlier) and will ensure higher quality calls too. A maximum cut off is important because regions with very, very high read depths are likely repetitive ones mapping to multiple parts of the genome.
- **Quality** Genotype quality is also an important filter - essentially you should not trust any genotype with a Phred score below 20 which suggests 99% accuracy.
- **Minor allele frequency** MAF can cause big problems with SNP calls - and also inflate statistical estimates downstream. Ideally you want an idea of the distribution of your allelic frequencies but 0.05 to 0.10 is a reasonable cut-off.
- **Missing data** How much missing data are you willing to tolerate? It will depend on the study but any site with >25% missing data should be dropped.

Filtering the vcf

Now we have an idea of how to filter our vcf, we will do just that. Run the following `vcftools` command on the data to produce a filtered vcf. We will investigate the options after the filtering is run.

```
vcftools --gzvcf stickleback_full_unfiltered.vcf.gz \
--remove-indels --maf 0.05 --max-missing 0.75 \
--minDP 10 --maxDP 200 --recode --stdout | gzip -c > \
stickleback_full_filtered.vcf.gz
```

What have we done here?

- `--gvcf` - input path – denotes a gzipped vcf file
- `--remove-indels` - remove all indels (SNPs only)
- `--maf` - set minor allele frequency - 0.05 here
- `--max-missing` - set minimum missing data. A little counterintuitive - 0 is totally missing, 1 is none missing. Here 0.75 means we will tolerate 25% missing data.
- `--recode` - recode the output - necessary to output a vcf
- `--stdout` - pipe the vcf out to the stdout (easier for file handling)

Now, how many variants remain? There are two ways to examine this - look at the vcftools log or the same way we tried before.

```
cat out.log
bcftools view -H stickleback_full_filtered.vcf.gz | wc -l
```

You can see we have substantially filtered our dataset!

Have a closer look at the vcf before moving on to the next step to get an idea of what a proper variant call with multiple individuals looks like

NB. Because the full vcf was generated for a proper dataset, it was not created in exactly the same way as the one we made today - so be aware there will be some differences - i.e. filenames, pipeline etc

---

5. Estimating population genomic statistics

Now we are finally at the stage that many of you will want to reach with your own organisms - performing population genomic analyses. Here we will carry out a simple analysis - estimating genome-wide *F*ST from our RAD_seq data.

Getting sample lists

Once again we will use `vcftools`. Before we can perform this analysis though we need to get a list of the samples in the vcf file. Luckily I have two prepared for you.

```
cd ~/rad_ref/vcf/
cp $TDIR/other/L* ./
cat L*
```

You should now see a list of samples. One of these files `LCS_samples.txt` is a list of individuals from the Little Campbell stream, the other, `LCM_samples.txt` is for Little Campbell marine fish.

Estimating per-site *F*ST

Now we have a list of samples for each population, we can perform F_{ST} estimates. First we will do this on a site by site basis.

```
vcftools --gzvcf stickleback_full_filtered.vcf.gz \
--weir-fst-pop LCM_samples.txt \
--weir-fst-pop LCS_samples.txt \
--out full_site
```

Here the `--weir-fst-pop` option is specified twice, to let `vcftools` know we want F_{ST} calculations between these two populations. `--out` specifies an output prefix.

This analysis is extremely quick. Let's look at the output.

```
head full_site.weir.fst
```

You can see now that there are *F*ST estimates for each site in our vcf file.

Estimating sliding window *F*ST

One big feature of working with genomic data is the **huge** number of sites you need to deal with. Plotting >4000 SNPs can be very noisy (as you will see shortly) so it is often easier to perform sliding window estimates of *F*ST - i.e. *F*ST averaged over a set genome-window. Here we will use 250 kb non-overlapping windows.

```
vcftools --gzvcf stickleback_full_filtered.vcf.gz \
--weir-fst-pop LCM_samples.txt \
--weir-fst-pop LCS_samples.txt \
--fst-window-size 250000 --fst-window-step 250000 \
--out full_site
```

This is almost identical to the command we ran previously except that now we have specified a window size with `--fst-window-size` and a window step with `--fst-window-step`. If we wanted a smoothed sliding average, we could have set the step to lower than window size, but for now this will do.

Let's look at the output:

```
head full_site.windowed.weir.fst
```

Now we have mean estimates for 250 kb windows!

Note: this is only one of many, many analyses you could perform. Pipelines such as `vcftools` have many options to compute population genomic statistics on vcf files. You might also want to consider `PopGenome`, an `R` package, the `Python` libraries `egg-lib` and `pyvcf` amongst others. Of course if you're feeling particularly daring, you can write your own… _____

6. Visualising the data

Now we have reached the final step of this exercise. There are lots of different options for visualising the data. Personally, I prefer `R`. Since you will not be learning `R` properly until later in the course, I will not go into much detail here.

To make things run more smoothly, I have written an `R` script that will create a plot for you. You are more than welcome to look at this script in detail and feel free to ask me questions about it. If you are already sufficient in `R` (or any other plotting utility for that matter), you are also more than welcome to attempt your own plot.

You downloaded this script when you cloned my git repository. So you can move it into the vcf directory like so:

```
cd ~/ref_rad/vcf
cp ~/mark_scripts/reference_mapping/fst_plotter.R ./
```

Now we will run the `R` script like a program:

```
Rscript fst_plotter.R -s full_site.weir.fst \
-w full_site.windowed.weir.fst -o fst_plot.pdf
```

If you want to know more about the options, run `Rscript fst_plotter.R -h`. This should produce a nice genome-wide FST plot as a pdf for you to look at. Again, feel free to look at the mechanics of the `fst_plotter.R` script using `less` or `nano`. I'm happy to answer questions on it for those interested.

So that's it for the first exercise! If you would like to learn more about the study system we looked at here, you can find out more in the two papers by Kusukabe *et al.* (2016)[8] and Ishikawa *et al.* (2017)[9] where this data was actually used!

---

denovo assembly RAD analysis

Introduction

As you are no doubt aware by now, the morning session was lengthy and had a lot to take on board! Now that we are all more familiar with how referenced-based RAD-seq works, hopefully this next exercise will be a bit more straightforward.

Our aim for this exercise is to do exactly the same as we did before - call variants from a RAD-seq dataset. Unlike before we will not map to a reference genome. Instead we will perform a *denovo* assembly using the Stacks pipeline before calling SNPs and then outputting the data for genomic analysis. Like before, we will generate FST estimates for our variants and if time allows, we will also use a method to detect selection.

For this analysis, we are going to work on a Nicuagaran cichlid species - *Amphilophus* from a single site in a Nicuagaran crater lake. There are two species in this analysis, the thick lipped *A labiatus* and the ancestral *A citronellus.* Our aim is to find SNPs under divergent selection between them.

---

[8]http://onlinelibrary.wiley.com/doi/10.1111/mec.13875/full
[9]http://onlinelibrary.wiley.com/doi/10.1111/evo.13175/abstract

1. Getting started

Login to the EVOP server like so:

```
ssh k00319250@evop2018login.imp.fu-berlin.de
```

Navigate to your home directory and make a new directory to work in.

```
cd ~
mkdir denovo_rad
cd denovo_rad
```

Like before, we are going to make a variable that will be useful for accessing data when we need it.

```
TDIR=/opt/evop/public/NGS_NonModelOrganisms/denovo/
```

Check it worked like so:

```
ls $TDIR
```

Next make a directory for the raw data and copy the data into it.

```
mkdir raw
cd raw_10k
cp $TDIR/raw_10k/* ./
```

Like before, have a look at the data. How many individuals are there for each species? Can you remember what each line in the fastq format represents?

---

2. Running the stacks pipeline

A bit of background

Unlike with our previous example, we do not have the luxury of a reference genome in this exercise. So we turn to the Stacks pipeline to perform our *denovo* assembly.

Stacks is very well supported and has a **LOT** of information on its website[10]. There are many components to the Stacks pipeline and it can also be used for reference aligned data too.

Typically, I would run Stacks each component at a time as it allows more control over how the pipeline runs. However this is not recommended for beginners so for today's purposes I am going to show you how to run the whole pipeline using the `denovo_map.pl` program.

How does this work? The *denovo pipeline* has a number of steps:

[10]http://catchenlab.life.illinois.edu/stacks/

1. `ustacks` - this identifies read stacks, matches them to form loci and then calls SNPs within individuals
2. `cstacks` - this creates a catalogue of all loci in the dataset and matches amongst individuals
3. `sstacks` - this matches each sample to the catalogue in order to identify loci.

Starting denovo_map.pl

There are many options for `denovo_map.pl` so the easiest way to understand what is going on is to actually run it and then we can break them all down.

Since there are so many options and it is very easy to make a mistake typing them all in, I will let you run a bash script to make the process easier. This was downloaded when you cloned the github repository this morning. Input the following to get the script into your current directory.

```
cd ~/denovo_rad
cp ~/mark_scripts/denovo/denovo_cichlid.sh ./
```

Before running this script, you will also need a 'map' of the population samples - i.e. identifying which population is which. More on this later.

```
cd ~/denovo_rad
cp ~/mark_scripts/denovo/cichlid.popmap ./
```

Take a quick look at this using `cat`. You will see it is a tab-delimited file with the sample name and population assignment (species in this case).

We will also take this opportunity to learn the useful utility, `screen`. This allows you to run multiple analyses in different screens and also preserves the analysis if your shell session is terminated for some reason. Start a new screen and then initiate the script in it.

```
cd ~/denovo_rad
mkdir stacks
screen -S stacks
bash denovo_cichlid.sh
```

You will now see the script is running. Press `ctrl + A + D` to move back to the main terminal. You can see the screens running (and return to your screen - named 'stacks' here) with the following:

```
screen -ls
screen -R stacks
```

Use `ctrl + A + D` to return to the main screen again. Now it's time to understand what the `denovo_cichlid.sh` script actually does. **NB. Don't run the code below, you'll ruin your currently running analysis!**

```
denovo_map.pl -m 3 -M 4 -n 4 -T 4 -b 1 -t \
-S -i 1 \
-O cichlid.popmap \
-o ./stacks \
-s ./raw/citronellus_10.fq.gz \
-s ./raw/citronellus_11.fq.gz \
-s ./raw/citronellus_12.fq.gz \
-s ./raw/citronellus_13.fq.gz \
-s ./raw/citronellus_14.fq.gz \
-s ./raw/citronellus_15.fq.gz \
-s ./raw/citronellus_16.fq.gz \
-s ./raw/citronellus_1.fq.gz \
-s ./raw/citronellus_2.fq.gz \
-s ./raw/citronellus_3.fq.gz \
-s ./raw/citronellus_4.fq.gz \
-s ./raw/citronellus_5.fq.gz \
-s ./raw/citronellus_6.fq.gz \
-s ./raw/citronellus_7.fq.gz \
-s ./raw/citronellus_8.fq.gz \
-s ./raw/citronellus_9.fq.gz \
-s ./raw/labiatus_10.fq.gz \
-s ./raw/labiatus_11.fq.gz \
-s ./raw/labiatus_12.fq.gz \
-s ./raw/labiatus_13.fq.gz \
-s ./raw/labiatus_1.fq.gz \
-s ./raw/labiatus_2.fq.gz \
-s ./raw/labiatus_3.fq.gz \
-s ./raw/labiatus_4.fq.gz \
-s ./raw/labiatus_5.fq.gz \
-s ./raw/labiatus_6.fq.gz \
-s ./raw/labiatus_7.fq.gz \
-s ./raw/labiatus_8.fq.gz \
-s ./raw/labiatus_9.fq.gz
```

This looks daunting but actually it's pretty simple. `denovo_map.pl` options are:

- `-m` - minimum number of identical reads to create a stack within an individual (therefore minimum locus depth is 2x this number)
- `-M` - minimum number of mismatches within an individual - i.e. to match two stacks into a single locus
- `-n`- minimum number of mismatches between loci when matching to build the catalogue
- `-T` - number of threads to run the analysis on
- `-b` - batch number for stacks catalogue - this can be set to 1 - it is necessary to run the pipeline but since we are not using SQL is not important.
- `-t` - break up repetitive stacks in ustacks - any stack with excessive reads will be split or removed to prevent repetitive regions being incorporated

into the analysis.
- `-S` - disable SQL database settings.
- `-i` - initiate individual ids - set to 1.

There are also a number of input/output options

- `-O` - population map - a tab-delimited file denoting population
- `-o` - path to the output directory
- `-s` - path to a fasta file for each individual sample

A brief note on SQL interaction

One of the features of Stacks is that it allows mySQL integration so that you can view your RAD-seq data in an interactive database. Personally I prefer to run Stacks without this option but some users find it useful. If you want to use it for your own projects in the future, you can rerun the above code with the `-S` flag removed. This will make Stacks load the results of the analysis into a mySQL database which is named with the `-D` flag. Type `denovo_map.pl -h` to see more options related to mySQL interaction.

Note that if you run an analysis without SQL interaction (like we have here) you can also later load it into an SQL database using the `load_radtags.pl` program.

What does stacks output?

Check your stacks run using `screen -R stacks` - is it done yet? If you have had problems running the analysis, you can copy the output of the pipeline into your stacks directory like so:

```
cd ~/denovo_rad/stacks
cp $TDIR/stacks/* ./
```

Use `ls -lah` to have a look at what is available here. Firstly you'll see a series of files that start with `batch_1`. These are the batch files produced by the catalogue.

For each individual you will see a series of files named `XXX.alleles.tsv.gz`, `XXX.matches.tsv.gz`, `XXX.snps.tsv.gz` and `XXX.tags.tsv.gz`. The main difference between these and `batch_1` are that the batch files are for the entire catalogue.

Use `zcat`and `head`to have a closer look at these. A quick summary:

- `XXX.alleles.tsv.gz` - alleles for the snp calls
- `XXX.matches.tsv.gz` - matches for the loci into the catalogue
- `XXX.snps.tsv.gz`- SNP calls for each locus
- `XXX.tags.tsv.gz`- denovo RAD tags identified by `ustacks`

You don't need to worry about these files so much as they will be used by the pipeline without your input. However you should have a close look at the `batch_1.catalog.tags.tsv.gz`. This is essentially a file of the consensus sequence for each locus identified in the population and can be useful for downstream analyses (i.e. when BLASTing RAD tags).

Chances are that when the pipeline was run, it failed to effectively run the `populations` module due to some sort of memory error. This usually happnes when there isn't enough memory to deal with this highly demanding step. This isn't fatal - we can run it again with more stringent filtrs to generate some output data using the `populations` module…

Filtering and exporting the data

The `populations` module allows us to filter the dataset based on the samples/loci we want to include. This is similar to the `vcftools` filtering step we carried out for our ref-aligned dataset.

Run populations like so:

```
cd ~/denovo_rad/
populations -b 1 -P ./stacks -M cichlid.popmap -t 4 \
-r 0.5 -p 2 -m 5 --fstats --vcf
```

With the reduced dataset, this should run quite quickly. As you will see from looking at the output, there are very few loci included in the final dataset - like with our reference dataset we will need to switch to some 'real data' for our downstream analyses to make any real sense. For now though, the populations options are:

- `-b` - batch number. Set to 1 - must be set but not important with SQL turned off.
- `-P` - path to Stacks output files
- `-M` - path to population map
- `-t` - thread number for running in parallel
- `-r` - minimum proportion of individuals present in a population to include a locus - 0.75 here.
- `-p` - number of populatiosn a locus must occur in - set to 2 here to ensure only loci present in both populations are included
- `-m` - minimum stack depth to include a locus. This is stack depth and *NOT* locus depth.
- `--fstats` - output F statistics for SNPs and haplotypes
- `--vcf` - output vcf format

NB: Stacks can output a wide-range formats - i.e. Structure, Phylip - for now though we only need these formats.

Another point that is worth making - Stacks is well maintained and is continually updated. A lot of new features for outputing data in various different formats have been added since this tutorial was first put together and new features are being added all the time. Be sure to check the different options on the pipeline website[11]

Now we have run the *denovo* Stacks pipeline in full. The next step is to look at this data in more detail.

---

3. Analysing the data

First things first, let's get the output from a full analysis.

```
cd ~/denovo_rad
mkdir results
cd results
cp $TDIR/\
populations_full/* ./
```

What does the output mean?

Before actually looking into this data in great detail, let's just get a sense of what we have actually produced here. Use `less` or `head` to look at each of the files in turn and examine the contents. The output is described below:

- `cichlid.vcf` - this should be familar after the last exercise - this is a vcf file for the SNP calls from each of the RAD loci. A special feature of vcf files output from Stacks is that the ID field contains the RAD locus identifier.
- `cichlid.sumstats_summary.tsv`- this is a summary of the summary statistics across all loci output by the pipeline. Useful for getting a general idea of what the data is showing. Here we can see there are 4929 private variants within *A. labiatus* for instance.
- `cichlid.sumstats.tsv` - a more detailed version of the previous file. This is the summary statistics per site (**per variant not per locus**). Note that because we required loci to occur in both populations, there should be two rows for each locus in this file.
- `cichlid.hapstats.tsv` - similar to the previous file but now based on RAD loci, not SNPs.
- `cichlid.phistats.tsv` - Phi statistics for haplotype based analyses - these are analogous to FST for the entire locus.
- `cichlid.haplotypes.tsv` - This is a matrix of haplotypes for each individual - this can be used for downstream analyses if necessary.
- `cichlid.genepop` - a genepop file generated from RAD data.

[11]http://catchenlab.life.illinois.edu/stacks/comp/populations.php

- `cichlid.fst_summary.tsv` - a summary file giving mean pairwise F_{ST}. Here there is only one value because we are only comparing two populations.
- `cichlid.fst_citrinellus-labiatus.tsv` - pairwise FST for each SNP site. One pairwise file is generated for each population in the analysis, since there are only two populations in this analysis, we have only a single file.
- `cichlid.phistats_citrinellus-labiatus.tsv` - as above but for Phi instead of F_{ST}. In this case only a single file and also Phi values should be identical to those in `cichlid.phistats.tsv`.
- `cichlid.populations.log` - a log of the populations run - useful for understanding why certain loci are dropped from the analysis. For example, the first part of this file shows the distribution of loci with a specific number of samples. You can see clearly that 891189 loci are missing samples (i.e. most probably removed after Stacks validity checks), the majority of loci also occur in only a single individual (249788 in this case).

For more information on all of these file formats and their fields, see the Stacks manual[12].

Assessing the output in more detail

We can get a basic idea of how well our analysis has performed from using some basic Unix commands on the output files. First of all - how many RAD loci did the full analysis produce?

```
cat cichlid.sumstats.tsv | tail -n +4 | cut -f 2 | uniq | wc -l
```

Here we used `cat` to view the file, `tail` to skip the first three lines, `cut` to extract the second column, `uniq` to retain only the unique identifiers and `wc -l` to count the number of lines. You should see we have >17 000 RAD loci… nice!

What about variants? We can use the vcf file for this. Ignore the `bcftools` warning - this is just because of the way Stacks handles vcf files.

```
bcftools view -H cichlid.vcf | wc -l
```

You will get a warning when you do this, but you can ignore it. More importantly, we have over >25 000 SNPs, which means we have ~1.4 SNPs per RAD locus. It is important to remember that RAD loci often have multiple SNPs and that these will be in linkage. This can have some serious repercussions for downstream analyses like detecting selection. One way to overcome this is to use RAD locus haplotypes instead of SNPs or alternatively to randomly select a single SNP from each locus.

We could do this manually in the vcf or we could rerun `populations` with either the `--write_single_snp` or `--write_random_snp` options. The first of these

[12] http://catchenlab.life.illinois.edu/stacks/manual/#pfiles

write the first SNP occurring on a haplotype while the second will randomly select one. For the rest of this exercise though, we will proceed with the data as it is.

Dealing with so much data

One issue you will have no doubt already realised is that it can be very difficult to really get an idea of how to get a handle on this amount of data. This is why it is really very useful to be able to plot your data properly.

If you have time, try plotting some of the data in `R` to get a feel for it. For instance, the distribution of FST or Phi statistics may be useful. The easiest way to do this will be download the files to your local machine (we can show you how to do this in the class).

I have written an `R` script for you to use to get a basic FST histogram. It was one of the scripts downloaded when you cloned my github repo.

Get the script like so:

```
cd ~/denovo_rad/results
cp ~/mark_scripts/denovo/plot_cichlid_fst.R ./
```

Now it should be in your `results` directory. To run it simply type the following:

```
Rscript plot_cichlid_fst.R
```

Download the plot and have a look at it.

---

4. Identifying population structure using PCA

Now that we have generated our population genomic data, one thing we can do is look for any population structuring. For example, we might expect some divergence between the two *Amphilophus* species.

A quick and straightforward way to investigate this is to use Principal Components Analysis[13] on allele frequencies. This will essentially tease apart the main axes of divergence between samples, based on allele frequency differences.

There are lots of ways to perform PCA, but one of the simplest is using the `R` package `adegenet` which is a versatile tool for population genomics[14].

However before we can make use of `adegenet` we need to convert our data into a different format using `plink` which is another really useful tool for genomic

[13] https://en.wikipedia.org/wiki/Principal_component_analysis
[14] http://adegenet.r-forge.r-project.org/

data[15]. **NB:** plink actually has it's own PCA tool but today I want you to use the adegenet version to expose you to as many tools as possible.

Converting to plink raw format

First things first, we need to convert our cichlid.vcf into a binary. To do so, run the following command:

```
plink1.9 --vcf cichlid.vcf --double-id --allow-extra-chr --recodeA
```

Note that since plink1.9 was originally written for human data, it can sometimes be a little fiddly to run. Let's break these command down:

- --vcf - specifies our input file is a vcf
- --double-id - the just tells plink to ignore the sample names in the vcf header; the program tries to assign samples to families and pedigrees and all this does is to tell it to use each name twice for that.
- --allow-extra-chr - tells plink to ignore the chromosome configuration in the vcf. Since this is denovo, we actually don't have any chromosomes (they are listed as Un for unknown) but otherwise, plink would expect 23 chromosomes, as in humans.
- --recodeA - this is the most important command as it tells plink1.9 how to recode the data into a new format. We need a raw output with an additive component (i.e. 0 for homozygote for the first allele, 1 for heterozygote and 2 for homozygote for the second allele)

Take a look in the directory using ls, you should see three files; plink.log, plink.nosex and plink.raw. Move out of the directory and create a pca directory, then move these files into it. Like so:

```
cd ..
mkdir pca
mv results/plink.* pca
cd pca
```

The next step is to perform a PCA on these files! Just in case this step didn't work for any reason, you can get the plink files like so:

```
cp $TDIR/plink/* ./
```

Performing PCA with adegenet

Creating the PCA is quite straightforward and luckily, like most good tools, adegenet is very well documented. In fact, there are a series of tutorials online[16] which show you how to perform most of the major analyses, including PCA.

[15]https://www.cog-genomics.org/plink2/
[16]https://github.com/thibautjombart/adegenet/wiki/Tutorials

Because of that and also because of time constraints we will use a custom R script to make our PCA today. But again, feel free to look at the script in detail. First of all, let's get the script:

```
cp ~/mark_scripts/denovo/make_pca_plot.R ./
```

Now run it. If you want to know what the commands do, use `--help`

```
Rscript make_pca_plot.R -i plink.raw -o pca_plot.pdf
```

Take a look at the PCA plot you've created. Can you see anything odd about it? Firstly you can see there is a good chance that one of the *A. labiatus* individuals has been mislabelled. Secondly, it seems like the main axis of differentiation between the species is well, not really between the species at all. In fact there are two clear outliers - this suggests there may be an error in the data. PCA is quite sensitive to missing data, so this can often be a good way to diagnose issues.

---

5. Diagnosing problems with the data

We saw in our PCA plot that there is clearly something contributing to variation along PC1 that is not just population structure. Basically, there is something that is separating those two individuals from all the others.

Finding out what this is a useful exercise in learning how to properly filter our data and to justify our choices for filtering. It is important to remember that there is **no standard way** to achieve this - what follows are guidelines, but it should still be useful.

The easiest way to begin to tackle this problem is to see if there is anything about our data that co-varies with PC1. We can generate some additional data and the plot them to see. To generate some summary statistics about samples, we will use `vcftools`.

First lets setup our environment.

```
mkdir ~/denovo_rad/vcftools
cd ~/denovo_rad/vcftools
cp ../results/cichlid.vcf ./
```

Next we will use `vcftools` to generate per-individual depth, heterozygosity and missingness reports. We will start with depth.

```
vcftools --vcf cichlid.vcf --out cichlid --depth
```

Running this command will create a file called `cichlid.idepth`. Take a look at it with `head`. It is a simple, tab-spaced file with individuals and their mean depth. This is basically the mean depth of coverage across all genotyped sites in our VCF. So our first individual has an average coverage of 41 reads for example.

Next we will calculate individual heterozygosity.

```
vcftools --vcf cichlid.vcf --out cichlid --het
```

Looking at the `cichlid.het` file using `head` or `less` shows another tab delimited file. This time we observed and expected homozygosity and a calculation of the inbreeding coefficient F (also known as a FIS). A value close to 1 indicates an excess of homozygosity, a value close to -1 indicates the opposite - an excess of heterozygotes.

And finally we will look at missingness per individual.

```
vcftools --vcf cichlid.vcf --out cichlid --missing-indv
```

The `cichlid.imiss` file basically tells us the number and fraction of missing sites per individual. High amounts of missing data can cause issues with estimating other statistics.

OK, so now that we have these files, what can we do with them? Since we are dealing with so much data here, it is ofent best to try and visualise the results. For this we turn again to `R`.

Firstly, if the vcftools step didn't work for you, you can copy the data from the teaching directory like so:

```
cd ~/denovo/vcftools
cp $TDIR/vcftools/* ./
```

We will also need the output of our previous PCA. Just to make things easier, we will bring it in to this directory.

```
cp ../pca/pca_data.csv ./
```

Don't worry if you couldn't generate that file either - I included it in the `vcftools` directory in the teaching directory - so you can get it using the command above.

Next, copy the R script I have written for you to plot this data.

```
cp ~/mark_scripts/denovo/plot_pc_covariation.R ./
```

Now run the script like so:

```
 Rscript plot_pc_covariation.R
```

Unfortunately, this doesn't actually explain much! This doesn't mean that this has been an unneccessary exercise - it is actually very worthwhile constructing plots like this because it can tell you about hidden variation in your dataset.

6. Detecting selection using Bayescan (an optional extra)

We saw in the section before last that Stacks produced some population genomic analyses on our behalf. But what if we want to go further than this and identify whether any of the SNP loci we identified are under divergent selection between these two morphs?

There are several different approaches we can take to achieve this but today we are going to use `Bayescan`, a Bayesian method[17] for that decomposes *F*ST into 'local and global effects.

One immediate issue we have is that Stacks does not output the format required to use `Bayescan`. Not to worry - we can produce one! The easiest way to do this is using a SNP matrix generated with the `vcftools perl`utility.

```
cd ~/denovo_rad/results
vcf-to-tab < cichlid.vcf > cichlid.geno
```

Use `head` to take a look at the file - this is a really nice, useful SNP genotype matrix for each individual in our dataset at each SNP position.

We can then convert this to a Bayescan input using a custom R script I have written. Use the following code to copy this to your working directory and then run it on the input.

```
cp ~/mark_scripts/extra/bayescan_convert.R ./
Rscript bayescan_convert.R -i cichlid.geno -o cichlid_bayes.txt
```

As before, feel free to look at this script to see what it is doing. The exact mechanics of it are beyond the scope of this tutorial but it should work with most datasets.

Now we are ready to run Bayescan! Run the following line, as before I'll break it down afterwards.

```
bayescan_2.1 ./cichlid_bayes.txt -o cichlid_bayes_run \
-threads 4 -n 5000 -thin 10 -nbp 20 -pilot 5000 \
-burn 50000 -pr_odds 10
```

What do we have here?

- `-o` - the output prefix
- `-threads` - number of threads to run in parallel
- `-n` - number of samples of the MCMC we want to take
- `-thin` - the thinning interval - basically how often to take a sample, also controls length of MCMC (so 5000*10 = 50000 iterations in this case)
- `-nbp` - the number of pilot runs the program makes to initiate the MCMC - Bayescan does this to optimise the MCMC search
- `-pilot` - length of MCMC for pilot runs

[17]http://cmpg.unibe.ch/software/BayeScan/

- `-burn-` the length of the burnin - here it is %50 of the total MCMC.

Chances are that this will take far too long to run in the class so you can copy some finished results here:

```
cp -r $TDIR/outlier/bayes/ ./
```

Note that here, the `-r` flag for `cp` means copy recursively - so it will copy the entire directory. Enter the directory you just copied and have a look at the output. The main one we are interested in is `cichlid_bayes_run_fst.txt` which is the *F*ST estimate for each locus. The qval in this file is the posterior proability, corrected for false positives that a locus is under selection.

As with most of these analyses, the easiest way to understand the output is to plot it. Again, I have written an `R` script to do this for you. Use the following code:

```
cp ~/mark_scripts/extra/Bayes_plot.R ./

Rscript Bayes_plot.R -i cichlid_bayes_run_fst.txt -o Cichlid_bayes_plot.pdf
```

Have a look at the plot and the output of the `R` script to the screen. How many loci are under selection? **Tip:** The dashed vertical line represents our cut-off threshold for determining whether a locus is under selection.

Concluding remarks

After completeing these two exercises, you should have a good idea of how to go from bare-bones sequencing data to a high quality SNP or RAD locus dataset.

Of course, it's important to keep in mind that these exercises are only meant as a guide and are not a definitive set of instructions. One of the nice things about population genomic analysis is that you can be very flexible and take things in whatever direction best suits your data.

There are many, many resources available out there and programs are always changing. Chances are that by the time you get to use some of these skills on your own data, the pipelines used here will have been updated and altered. The best advice I can give you for your own data is that you should work with it again and again. For many of my projects, I have repeated analyses often more than 3 times.

This might seem like a nightmare to some but it really is the best way to learn these programs, scripting languages and analysis methods. Have fun and don't worry about asking questions - most people are new to bioinformatics!