GAWorkshop Documentation

Release 1

Schiffels S, Peltzer A, Clayton S

Mar 03, 2017

Contents

This is the central documentation / handout for a workshop held May 9 - May 11 2016 by Dr. Stephan Schiffels, Alexander Peltzer and Stephen Clayton at the Max-Planck-Institute for the Science of Human History, Jena/Germany.

Contents:

CHAPTER 1

Introduction

This session will introduce some basics of the command line environment and connecting to the compute cluster at google.

Basic software

There are some softwares that you will want to install for this session.

Mac OSX

Direct access to cluster storage:

- http://osxfuse.github.io/
  - You need to install FUSE and SSHFS

An editor for text files and scripts:

- https://atom.io

Some things that you may want to have on your mac but that you don't need right now:

- https://www.macports.org
  - Software packaged for your mac
- https://developer.apple.com/downloads/
  - Apple developer command line tools

Ubuntu

```
apt-get sshfs
```

The command line environment

The command line environment allows you to interact with the compute system using text. The standard mode of interaction is as follows:

- **R**ead
- **E**valuate
- **P**rint
- **L**oop

The commands that you type are read and evaluated by a program called the **shell**. The window that displays the shell program is usually called the **terminal**.

- Terminal - displays the shell program
- Shell - the **REPL**

  We interact with the shell program using text. The shell typically reads input one line at a time. Therefore we commonly call the interaction with the shell via a terminal the 'command line'.

Using the bash shell

There are a few different shell programs available but we will use 'bash'.

```
# To find what shell your using
ps -ef | grep $$ | grep -v "grep\|ps -ef"
 502 44414 44413   0 21Apr16 ttys003    0:03.96 -bash
```

Bash reads your input and evaluates it. Some words are recognised by bash and interpreted as commands. Some words are part of bash, others depend on settings that you can modify. Bash is in fact a programming language. You can find a manual here:

- https://www.gnu.org/software/bash/manual/bashref.html

The environment

Many programs (including bash) need to be able to find things or know about the system. A universal way to supply this information is via environment variables.

- The environment is the set of variables and their values that is currently visible to you.
- A variable is simply a value that we can refer to by its name.

You can set environment variables using the 'export' command.

```
# Here we prepend to the environment variable called 'PATH'
export PATH="/Users/clayton/SoftwareLocal/bin:$PATH"
```

The PATH variable is important because it supplies a list of directories that will be searched for commands. This means that words that are not built in bash commands will be recognised as commands if the word matches an **executable** file in a directory from the list.

An example

```
is_this_my_environment
Yes, but we can make it better! Repeat after me:
export PATH="/projects1/tools/workshop/2016/GenomicAnalysis/one/bin:$PATH"
is_this_my_environment
```

```
# Good advice
export PATH="/projects1/tools/workshop/2016/GenomicAnalysis/one/bin:$PATH"
is_this_my_environment
Yes, and now you know how to make it better!
```

We ran the command 'is_this_my_environment' twice but got different results??

- Actually this is very useful - We say what should be done - The system takes care of how

Connecting to the cluster

1. Command line

```
ssh google.co.uk
# To make this easier you can add the following to your ssh config
cat ~/.ssh/config
Host google
HostName google.co.uk
User clayton

# This will let you connect to the cluster without as much typing
ssh google
```

2. Storage

```
mkdir -p /Users/jane/Mount/google_home
# If you are on Ubuntu then you should omit the ovolname option
sshfs jane@google: /Users/jane/Mount/google_home -ovolname=google
```

The environment on the cluster

On the google cluster you have the option of using a pre-configured environment. In your home directory on the cluster you can add the following to your bash profile.

```
cat ~/.bash_profile
source /projects/profiles/etc/google_default
# logout and login again for this to take effect
exit
ssh google
# You now have the latest versions of software installed at google on your path
# We can find the location of a program using which
which bash
/bin/bash
```

If you have mounted your google home folder then you can do this:

```
touch ~/Mount/google_home/.bash_profile
open -a Atom ~/Mount/google_home/.bash_profile
```

And now add the following line and save:

```
source /projects/profiles/etc/google_default
```

Next time you login in to google using ssh this setting will take effect.

Doing things with bash

Here are some examples to get you started.

- http://www.tldp.org/LDP/abs/html/

The basics

There are some basic operators that you should be familiar with:

```
| pipe
> less than
& ampersand
```

There are some variables that are set by bash. These are useful for seeing if your commands worked.

```
# The exit code of the last command you ran
echo $?
```

A key concept in bash is chaining commands to create a pipeline. Each command does something to your data and passes it to the next command. We can use the <b>pipe</p> operator to pass data to the next command (instead of printing it on the screen).

```
# Echo produces the string 'cat'.
# Tr replaces the letter 'c' with 'b'
echo "cat" | tr "c" "b"
bat
# What if our pipeline is much more complicated?
# What happens if a step in the pipeline fails?
echo "not working" | broken_command | sed -e 's/not //'
echo "$?"
0
# The exit code of the last command 'sed' was 0 (success) and yet our pipeline failed
# We have to tell bash to give us the exit code of failing commands instead
# We do this by using the set builtin and the pipefail option
set -o pipefail
echo "not working" | broken_command | sed -e 's/not //'
echo "$?"
127
```

You can run your commands in different ways. This is useful if you want to run things in parallel or want to use the results in your program.

```
# Run the command in a sub shell
result=$(echo "the cat on the mat")
# Run the command without waiting for the result
echo "the cat sat on the mat" &
```

Useful things

If you want to repeat the same command for different inputs then looping is useful. There are some different ways to write loops, depending on the data you have.

```
#result=$(cat words.txt)
result=$(echo "the cat was black")
echo "${result}"

for i in $result;
do
```

```
  if [[ $i == "black" ]]; then
    echo "white"
  else
    echo "${i}"
  fi
done

if [[ -e words.txt ]]; then
  echo "I found the words"
fi
```

The spaces (especially around the [[ ]]) are important.

A very useful program is **xargs**. This is an incredibly useful command because it lets you create new commands from your input text.

```
echo "This is about cats" > words.txt
echo "This is about dogs" > other_words.txt
echo -e "words.txt\nother_words.txt" | xargs -I '{}' cat {}
This is about cats
This is about dogs
# If you want to see what your command will be before you run it then
# you can run the echo program to produce your command as text
echo -e "words.txt\nother_words.txt" | xargs -I '{}' echo "cat {}"
cat words.txt
cat other_words.txt
# To run these lines you can pipe them into bash
echo -e "words.txt\nother_words.txt" | xargs -I '{}' echo "cat {}" | bash
This is about cats
This is about dogs
# By piping commands into bash as text it is easy to achieve complex tasks
# e.g. Creating copy or move commands using awk to build the destination
#      file path using components of the source path
```

CHAPTER 2

Schmutzi Workshop

This is intended to provide a little hands on experience with schmutzi by G. Renaud et al and is solely designed to provide some more detailed step-by-step information on how to determine contamination using the tool. If you find bugs in this, I'm happy to fix them; if you find bugs in the tool itself, please use the projects GitHub repository to open issues for them here . This is _not_ my tool, but I happen to be one of the more frequent users of the method, thus this was basically writing up things I found out myself or with help from the developer(s).

Detection of contamination on mitochondrial data

schmutzi can be used to detect human contamination in mitochondrial data. In case you have enough reads mapping on to the mitochondrion reference genome, you can utilize the methods provided by *schmutzi* to automatically detect present contamination from other human sources automatically. This procedure is split into two parts, one performing a damage-based contamination estimation and a second one utilizing a database of mitochondrial allele frequencies. During this workshop, we will work on two human endogenous samples of undisclosed origin, that you will have to analyze yourselves: *Sample_A* and *Sample_B*. Of course, you can also transfer your own test cases to our system and then apply the methods taught in this course on these instead.

Warning: The provided test BAM files are only for testing purposes and should not be distributed further.

Note: You can run these methods and single steps manually but you could also run this in a more concise way instead by creating a simple bash script for your convenience. In case you'd like to do this, follow the simple template provided below:

```
#!/bin/bash
command1
command2
...
```

Sample preparation

Subsampling

In order to make these use cases computationally feasible, please do not use samples with more than 50x mitochondrial coverage or subsample the samples if you have more coverage than this.

Note: In case you use our provided sample datasets, you don't need to do any subsampling, as the samples we provide as use cases are small in size anyways.

For those of you who would like to use their own datasets, please apply *samtools view* and produce a subsampled version of your input data if the input file is too large for our course.

```
samtools view -b -s 0.2 input.bam -o output.bam
```

This should produce a rough 20% of your input file, which is taking randomly reads from the sample instead of taking an order set of reads from the input.

MD-Tagging

In order for *schmutzi* being able to access the data properly, we need to add **MD** tags to the BAM files. **MD** tags can be used by programs to perform e.g. SNP detection without a reference genome, as the tag contains information on which positions of the corresponding read there are matches, mismatches or indels. To add MD tags to your data, use the *samtools calmd* command:

```
samtools calmd -b Sample_A.bam ../ref/human_MT.fa > Sample_A.MD.bam
```

Warning: In order for this to work, you need to ensure to use the **same reference** that you used for mapping/creating your BAM file(s)!

Damage based *contDeam*

This can solely be used to determine contamination based on endogenous deamination. This means, if you use for example UDG treated data, that *contDeam* will tell you that your sample is severely contaminated (as it shows *no deamination* or at least *less contamination*). We are only using the default way of *contDeam*, a more complete documentation can be found on GitHub for your convenience.

First, run *contDeam* on each sample individually. As this produces quite an amount of output files during the iterative process, we should create a folder structure to store our output in a logical way.

```
mkdir -p Results/Sample_A/
mkdir -p Results/Sample_B/
```

This creates two folders in our current folder, making it possible to store all the output created by our methods to be applied in a logical way. Now, we can move on to use *contDeam*:

```
contDeam --library double --out Results/Sample_A/Sample_A --uselength --ref Ref/human_
↪MT.fa RAW_BAMs/Sample_A.MD.bam
contDeam --library double --out Results/Sample_B/Sample_B --uselength --ref Ref/human_
↪MT.fa RAW_BAMs/Sample_B.MD.bam
```

```
```

Note: You should make sure to use the proper commandline here: Specifying *single* for a double stranded library would not produce any meaningful results and thus render your estimation wrong potentially. Make sure to check prior using the command **which** kind of data you have here! Typically you do have double stranded data, but in case you are not certain that you have, you may want to check this with sequencing before.

This should produce something like this on your command line:

```
Reading BAM to set priors for each read ...
..  done
running cmd /projects1/tools/schmutzi/posteriorDeam.R Results/Sample_A/Sample_A.cont.
↪deam Results/Sample_A/Sample_A.cont.pdf  "Posterior probability for contamination\n
amination patterns"
null device
          1
Program finished succesfully
Files created:The plot of the posterior probability is Results/Sample_A/Sample_A.cont.
↪pdf
The contamination estimate is here Results/Sample_A/Sample_A.cont.est
The configuration file is here Results/Sample_A/Sample_A.config
```

You may have a look now at the output of this initial contamination estimation run. How do your samples seem to look like for *Sample_A* and *Sample_B* ? To check this, you can have a look at the output initially generated using e.g. *cat*:

```
cat Results/Sample_A/Sample_A.cont.est
0 0 0.95
cat Results/Sample_B/Sample_B.cont.est
0 0 0.005
```

This means, that based on the deamination patterns both samples look relatively clean with a initial lower estimate of 0 % contamination, an average of 0% and an upper estimate of 95% for the first and 0.05% for the second sample. Relatively means in this case, that *Sample_B* looks clean completely, whereas *Sample_A* shows an initial high contamination of 95%.

However, you can't trust these results individually if:

1. You have less than 500 Million reads (which is very rarely the case)
2. You don't have enough deamination, less than 5% won't work for example (Attention: UDG treatment!)
3. Very little / No deamination of the contaminant fragments
4. (Independence between 5' and 3' deamination rates is required for the Bayesian inference model)

This method could be used for running contamination estimates on both nuclear and mitochondrial data in general, however I would recommend applying DICE for samples with nuclear data in general or perform other tests (X-chromosomal contamation test, looking forward to Stephan Schiffels introduction on this). I will generate a HowTo for DICE in the upcoming weeks, following the *schmutzi* manual here, too.

Mitochondrial based *schmutzi*

Now that we have successfully estimated contamination using *deamination patterns*, we will proceed by using allele frequencies on mitochondrial data, too. *schmutzi* comes with a database of 197 allele frequencies accompanied by an Eurasian subset of allele frequencies, that can be used for our analysis.

Note: If you would like to test e.g. for contamination on other organisms, e.g. some other mammals and you do possess enough datasets to generate such a database, you can also generate these frequencies yourself. For more details, follow Gabriel Renaud's HowTo here .

Now let's run the *schmutzi* application itself. Prior to doing this, we need to index our MD tagged BAM file first:

```
samtools index RAW_BAMs/Sample*.MD.bam
schmutzi --ref Ref/human_MT.fa --t 8 Results/Sample_A/Sample_A /projects1/tools/
↪schmutzi/alleleFreqMT/197/freqs/ RAW_BAMs/Sample_A.MD.bam
schmutzi --ref Ref/human_MT.fa --t 8 Results/Sample_B/Sample_B /projects1/tools/
↪schmutzi/alleleFreqMT/197/freqs/ RAW_BAMs/Sample_B.MD.bam
```

Warning: Make sure to use the correct **freqs** folder, or the tool will crash.

The whole process might run for a couple of minutes, mainly depending on the number of CPU cores `--t 8` you assigned your estimation process.

Warning: Do not use more CPU cores than available, or the whole system might get unstable. *schmutzi* can be pretty heavy in terms of memory / CPU usage, taking up a lot of your systems computational capacities.

In the end, this should produce some output:

```
Reached the maximum number of iterations (3) with stable contamination rate at␣
↪iteration # 5, exiting
Iterations done
Results:
      Contamination estimates for all samples        : Results/Sample_A/Sample_A_
↪final_mtcont.out
      Contamination estimates for most likely sample : Results/Sample_A/Sample_A_
↪final.cont
      Contamination estimates with conf. intervals   : Results/Sample_A/Sample_A_
↪final.cont.est
      Posterior probability for most likely sample   : Results/Sample_A/Sample_A_
↪final.cont.pdf
      Endogenous consensus call                      : Results/Sample_A/Sample_A_
↪final_endo.fa
      Endogenous consensus log                       : Results/Sample_A/Sample_A_
↪final_endo.log
      Contaminant consensus call                     : Results/Sample_A/Sample_A_
↪final_cont.fa
      Contaminant consensus log                      : Results/Sample_A/Sample_A_
↪final_cont.log
      Config file                                     : Results/Sample_A/Sample_A.
↪diag.config
Total runtime 248.527676105499 s
```

Running a small `cat` again to check the results of the contamination analysis:

```
cat Results/Sample_A/Sample_A_final.cont.est
0.99 0.98
```

This means, we have between 98%-99% contamination in *Sample_A*, making it useless for any downstream analysis.

Doing the same with our other sample now:

```
cat Results/Sample_B/Sample_B_final.cont.est
0.01  0 0.02
```

Which looks good - this sample seems safe to be used for downstream analysis, as it shows between 0 (low) - 1% (avg) - 2% (high) contamination estimate.

Output interpretation

`EST` Files

schmutzi generates a couple of output files that can be used to determine whether your samples are clean or not. The table above in *output_files* describes what kind of output to expect on a successful run of *schmutzi*. The most important file is the one with ending `est` as it provides the contamination estimate for your data.

The content of the `est` file should look like this:

```
X  Y Z
```

Where X is your average estimate, Y your lower estimate and Z your upper estimate. In some cases you will only see two numbers appearing, meaning that this is your upper and lower bounds respectively. It depends on your kind of analysis you'd like to perform whether you want to include edge cases with e.g. upper contamination of 3% estimate or not.

In the case you performed a full evaluation using both the *contDeam* and the *schmutzi* tools, you will see several `est` files, containing estimates in each iteration. *schmutzi* iteratively refines the consensus called by the *mtCont* subprogram, meaning that it will provide intermediate results in these files, numbered ascendingly from 1, 2 to *final*.

```
Sample_B_1_cont.3p.prof
Sample_B_1_cont.5p.prof
Sample_B_1_cont.est
Sample_B_1_cont.fa
Sample_B_1_cont.freq
Sample_B_1_cont.log
Sample_B_1_endo.3p.prof
Sample_B_1_endo.5p.prof
Sample_B_1_endo.fa
Sample_B_1_endo.log
Sample_B_1_mtcont.out
Sample_B_2_cont.3p.prof
Sample_B_2_cont.5p.prof
Sample_B_2_cont.est
Sample_B_2_cont.fa
Sample_B_2_cont.freq
Sample_B_2_cont.log
Sample_B_2_endo.3p.prof
Sample_B_2_endo.5p.prof
Sample_B_2_endo.fa
Sample_B_2_endo.log
Sample_B_2_mtcont.out
```

The first *est* file is based on the *contDeam* results (on the damage patterns), whereas the others are based on the iterative process used when estimating contamination using the mt database.

`FA` Files

These contain for both the endogenous part as well as the contaminant part the respective consensus sequences produced. Note that this has not been filtered at all and should therefore only be used for determining contamination and not for any downsteam analysis.

`Log` Files

These files are the raw output schmutzi produces using a bayesian method to infer the endogenous part of your sample. If you want to use downstream analysis on your data, e.g. calling haplotypes on your mitochondrion, you should apply some filtering on your dataset, which we will do in the next part of our analysis journey.

Consensus Calling for Downstream analysis

Filtered Consensus Calling

In order to get filtered calls, e.g. no SNPs for regions covered with only a single read, one should apply some filtering criteria:

```
/projects1/tools/schmutzi/log2fasta -q 20 Results/Sample_A/Sample_A_final_endo.log >␣
↪Results/Sample_A/Sample_A_q20.fasta
/projects1/tools/schmutzi/log2fasta -q 30 Results/Sample_A/Sample_A_final_endo.log >␣
↪Results/Sample_A/Sample_A_q30.fasta
/projects1/tools/schmutzi/log2fasta -q 20 Results/Sample_B/Sample_B_final_endo.log >␣
↪Results/Sample_B/Sample_B_q20.fasta
/projects1/tools/schmutzi/log2fasta -q 30 Results/Sample_B/Sample_B_final_endo.log >␣
↪Results/Sample_B/Sample_B_q30.fasta
```

It is advisable to choose these parameters increasingly, e.g. with a range of `-q 20, -q 30, -q 40, -q 50` and check whether you still have enough diagnostic positions in the end.

A good way to determine whether we have a lot of undefined positions relative to our used reference genome is by iteratively running several times the above command, to find an acceptable threshold between filtering and reserving enough information for the analysis.

```
tr -d -c 'N' < Results/Sample_A/Sample_A_q20.fasta | awk '{ print length; }'
16,569
tr -d -c 'N' < Results/Sample_A/Sample_A_q30.fasta | awk '{ print length; }'
16,569
```

As you see, for our *Sample_A*, the output doesn't change, meaning we already have pretty high numbers of 'N' in our output, meaning they have been filtered out with such light filtering (q20,q30) already. That does basically tell us, that the SampleA is of bad provenance, having likely contamination and potentially poorly covered bases, too. As you might recall, this is totally fine, since *schmutzi* declared this sample to be heavily contaminated anyways. Therefore we repeat this for *Sample_B* now to see if this behaves better:

```
tr -d -c 'N' < Results/Sample_B/Sample_B_q20.fasta | awk '{ print length; }'
90
tr -d -c 'N' < Results/Sample_B/Sample_B_q30.fasta | awk '{ print length; }'
351
```

As you can see we only have 90 bases not defined with a pretty decent filtering parameter already. When going down to filtering even more conservative with `q=30`, you can see that we are loosing even more positions but still have a reasonable amount of diagnostic positions. I leave it up to you to figure out a good threshold when you loose more than you gain in the end.

Unfiltered Consensus Calling

For modern samples we can use the application `endoCaller` coming with schmutzi instead, as we don't want to run contamination checks on this. This can be done using:

```
/projects1/tools/schmutzi/endoCaller -seq youroutput.fasta -log outputlog.log␣
↪reference.fasta input.bam
```

This will produce a consensus call, which is **unfiltered**. To test what kind of difference this makes, you may for example try running this method on one of our ancient samples comparing the output to a filtered output FastA directly. You will observe that especially in lower coverage situations, the `endoCaller` incorporates SNPs based on e.g. a coverage of 1 or low quality regions, whereas the filtering approach as defined in *Filtered Consensus Calling* .

CHAPTER 3

Sex Determination and X chromosome contamination estimate

One way to determine the nuclear contamination estimate works only in male samples. It is based on the fact that males have only one copy of the X chromosome, so any detectable heterozygosity on the X chromosome in males would suggest certain amount of contamination. Note that the sex of the contaminating sample is not important here, as both male and female contamination would contribute at least one more x chromosome, which would show up in this contamination test.

We will proceed in two steps. First, we need to determine the molecular sex of each sample. Second, we will run ANGSD for contamination estimation on all male samples.

Sex determination

On the cluster

The basic idea here is to compare genomic coverage on the X chromosome with the genomic coverage on autosomal chromosomes. Since males have only one copy of the X, coverage on X should be half the value on autosomes, while in females it should be roughly the same. Similarly, since males one copy of Y, coverage on Y should be half the value of autosomes in males, and it should be zero in females.

The first step is to run `samtools depth` on the bam files. To get an idea on what this tool does, first run it and look at the first 10 lines only:

```
samtools depth -q30 -Q37 -a -b $SNP_POS $BAM_FILE | head
```

Generally, when I write `$SNP_POS` or `$BAM_FILE`, you need to replace those variables with actual file names. Here, `-b $SNP_POS` inputs a text file that contains the positions in the capture experiment. We have prepared various SNP positions files for you, for both 390K and 1240K capture panels. You can find them in `/projects1/Reference_Genomes/Human/hs37d5/SNPCapBEDs` and `/projects1/Reference_Genomes/Human/HG19/SNPCapBEDs`, depending on which reference genome you have used to map the genome you are working with, and which SNP panel was used (e.g. 1240k). The flags `-q30 -Q37` are filters on base- and mapping quality, and the `-a` flag forces output of every site, even those with coverage 0 (which is important for counting sites). Finally, `$BAM_FILE` denotes the - wait for it - bam file.

The output of this command should look like this:

```
1    752567   0
1    776546   1
1    832918   0
1    842013   0
1    846864   0
1    869303   0
1    891021   5
1    896271   0
1    903427   0
1    912660   1
```

(Use `Ctrl-C` to stop the command if it stalls.) The columns denote chromosome, position and the number of reads covering that site. We now need to write a little script that counts those read numbers for us, distinguishing autosomes, X chromosome and Y chromosome. Here is my version of this in `awk`:

```
BEGIN {
    xReads = 0
    yReads = 0
    autReads = 0

    xSites = 0
    ySites = 0
    autSites = 0
}
{
    chr = $1
    pos = $2
    cov = $3
    if(chr == "chrX") {
        xReads += cov
        xSites += 1
    }
    else if(chr == "chrY") {
        yReads += cov
        ySites += 1
    }
    else {
        autReads += cov
        autSites += 1
    }
}
END {
    OFS="\t"
    print("xCoverage", xSites > 0 ? xReads / xSites : 0)
    print("yCoverage", ySites > 0 ? yReads / ySites : 0)
    print("autCoverage", autSites > 0 ? autReads / autSites : 0)
}
```

`awk` is a very useful UNIX utility that is perfect for doing simple counting- or other statistics on file streams. You can learn awk yourself if you want, but for now the only important thing are the code lines which read

```
if(chr == "X") {
    ...
}
else if(chr == "Y") {
    ...
}
else {
    ...
}
```

As you can see, these lines check whether the chromosome is X or Y or neither of them (autosomes). Here you need to make sure that the names of the chromosomes are the same as in the reference that you used to align the sequences. You can quickly check that from the output of the `samtools depth` command above. If the first column looks like `chr1` or `chr2` instead of `1` or `2`, than you need to change the awk script lines above to:

```
if(chr == "chrX") {
    ...
}
else if(chr == "chrY") {
    ...
}
else {
    ...
}
```

Makes sense, right? OK, so now that you have your little awk script with the correct chromosome names to count sites, you can pipe your samtools command into it:

```
samtools depth -q30 -Q37 -a -b $SNP_POS $BAM_FILE | head -1000 | awk -f␣
↪sexDetermination.awk
```

where I assume that the `awk`-code above is copied into a file called `sexDetermination.awk` in the current directory. Here, I am only piping the first 1000 lines into the awk script to see whether it works. The output should look like:

```
xCoverage   0
yCoverage   0
autCoverage 2.19565
```

OK, so here we did not see any X- or Y-coverage, simply because the first 1000 lines of the `samtools depth` command only output chromosome 1. But at least you now know that it works, and you can now prepare the main run over all samples. For that we need to write a shell script that loops over all samples and submits samtools-awk pipeline to SLURM. Open an empty file with an editor and write a file called `runSexDetermination.sh` or something like it. In my particular project, that file looks like this:

```
#!/usr/bin/env bash

BAMDIR=/data/schiffels/MyProject/mergedBams.backup
SNP_POS=/projects1/Reference_Genomes/Human/hs37d5/SNPCapBEDs/1240KPosGrch37.bed
AWK_SCRIPT=~/dev/GAworkshop/sexDetermination.awk
OUTDIR=/data/schiffels/GAworkshop

for SAMPLE in $(ls $BAMDIR); do
    BAM=$BAMDIR/$SAMPLE/$SAMPLE.mapped.sorted.rmdup.bam
    OUT=$OUTDIR/$SAMPLE.sexDetermination.txt
    CMD="samtools depth -q30 -Q37 -a -b $SNP_POS $BAM | awk -f $AWK_SCRIPT > $OUT"
    echo "$CMD"
    # sbatch -c 2 -o $OUTDIR/$SAMPLE.sexDetermination.log --wrap="$CMD"
done
```

Here, I am merely printing all commands to first check them all and convince myself that they "look" alright. To execute this script, make it executable via `chmod u+x runSexDetermination.sh`, and run it via `./`

runSexDetermination.sh.

Indeed, the output look like this:

```
samtools depth -q30 -Q37 -a -b /projects1/Reference_Genomes/Human/hs37d5/SNPCapBEDs/
↪1240KPosGrch37.bed /data/schiffels/MyProject/mergedBams.backup/JK2128udg/JK2128udg.
↪mapped.sorted.rmdup.bam | awk -f /home/adminschif/dev/GAworkshop/sexDetermination.
↪awk > /data/schiffels/GAworkshop/JK2128udg.sexDetermination.txt
samtools depth -q30 -Q37 -a -b /projects1/Reference_Genomes/Human/hs37d5/SNPCapBEDs/
↪1240KPosGrch37.bed /data/schiffels/MyProject/mergedBams.backup/JK2131udg/JK2131udg.
↪mapped.sorted.rmdup.bam | awk -f /home/adminschif/dev/GAworkshop/sexDetermination.
↪awk > /data/schiffels/GAworkshop/JK2131udg.sexDetermination.txt
samtools depth -q30 -Q37 -a -b /projects1/Reference_Genomes/Human/hs37d5/SNPCapBEDs/
↪1240KPosGrch37.bed /data/schiffels/MyProject/mergedBams.backup/JK2132udg/JK2132udg.
↪mapped.sorted.rmdup.bam | awk -f /home/adminschif/dev/GAworkshop/sexDetermination.
↪awk > /data/schiffels/GAworkshop/JK2132udg.sexDetermination.txt
...
```

which looks correct. So I now put a comment (#) in from of the echo, and remove the comment from the sbatch, and run the script again. Sure enough, the terminal tells me that 40 jobs have been submitted, and with squeue, I can convince myself that they are actually running. After a few minutes, jobs should be finished, and you can look into your output directory to see all the result files. You should check that the result files are not empty, for example by listing the results folder via *ls -lh* and look at column 4, which displays the size of the files in byte. It should be larger than zero for all output files (and zero for the log files, because there was no log output):

```
adminschif@cdag1 /data/schiffels/GAworkshop $ ls -lh
total 160K
-rw-rw-r-- 1 adminschif adminschif  0 May  4 10:16 JK2128udg.sexDetermination.log
-rw-rw-r-- 1 adminschif adminschif 56 May  4 10:20 JK2128udg.sexDetermination.txt
-rw-rw-r-- 1 adminschif adminschif  0 May  4 10:16 JK2131udg.sexDetermination.log
-rw-rw-r-- 1 adminschif adminschif 56 May  4 10:20 JK2131udg.sexDetermination.txt
-rw-rw-r-- 1 adminschif adminschif  0 May  4 10:16 JK2132udg.sexDetermination.log
-rw-rw-r-- 1 adminschif adminschif 56 May  4 10:20 JK2132udg.sexDetermination.txt
...
```

On your laptop

OK, so now we have to transfer those *.txt files over to our laptop. Open a terminal on your laptop, create a folder and *cd* into that folder. In my case, I can then transfer the files via

```
scp adminschif@cdag1.cdag.shh.mpg.de:/data/schiffels/GAworkshop/*.sexDetermination.
↪txt .
```

(Don't forget the final dot, it determines the target directory which is the current directory.)

We now want to prepare a table to load into Excel with four columns: Sample, xCoverage, yCoverage, autCoverage. For that we again have to write a little shell script, which in my case looks like this:

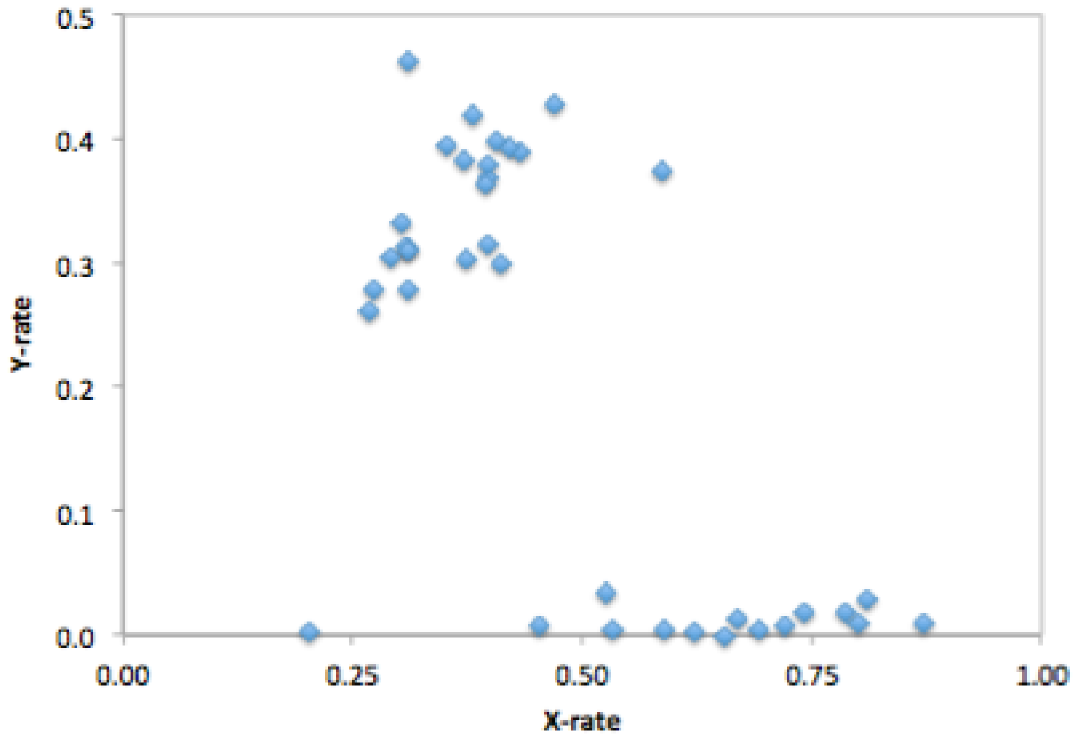
```
#!/usr/bin/env bash

printf "Sample\txCov\tyCov\tautCov\n"

for FILENAME in $(ls ~/Data/GAworkshop/*.sexDetermination.txt); do
    SAMPLE=$(basename $FILENAME .sexDetermination.txt)
    XCOV=$(grep xCoverage $FILENAME | cut -f2)
    YCOV=$(grep yCoverage $FILENAME | cut -f2)
    AUTCOV=$(grep autCoverage $FILENAME | cut -f2)
    printf "$SAMPLE\t$XCOV\t$YCOV\t$AUTCOV\n"
done
```

Make your script executable using `chmod` as shown above, and run it. The result looks in my case like this:

```
schiffels@damp132140 ~/dev/GAworkshopScripts $ ./printSexDeterminationTable.sh
Sample      xCov    yCov    autCov
JK2128udg   1.20947 1.17761 1.25911
JK2131udg   1.31687 1.41748 1.44766
...
```

OK, so now we need to load this into Excel. On a mac, you can make use of a nifty little utility called *pbcopy*, which allows you to pipe text from a command directly into the computer's clipboard: `./printSexDeterminationTable.sh | pbcopy` does the job. You can now open Excel and use `CMD-V` to copy things in. On Windows or Linux, you should pipe the output of the script into a file, e.g. `./printSexDeterminationTable.sh > table.txt`, and load `table.txt` into Excel.

Finally, use Excel to form ratios xCov/autCov and yCov/autCov, so the relative coverage of the X- and Y-chromosome, compared to autosomes. You could now for example plot those two numbers as a 2D-scatter plot in Excel and look whether you see two clusters corresponding to males and females. An example, taken from a recent paper (Fu et al. 2016 "The genetic history of Ice Age Europe"), looks like this:



As you can see, in this case the relative Y chromosome coverage provides a much better separation of samples into (presumably) male and female, so here the authors used a relative y coverage of >0.2 to determine males, and <0.05 to determine females. Often, unfortunately, clustering is much less pronounced, and you will have to manually decide how to flag samples as "male", "female" or "unknown".

Nuclear contamination estimates in Males

Now that we have classified at least some samples as "probably male", we can use their haploid X chromosome to estimate nuclear contamination. For this, we use the ANGSD-software. According to the ANGSD-Documentation, estimating X chromosome contamination from BAM files involves two steps.

The first step counts how often each of the four alleles is seen in variable sites in the X chromosome of a sample:

```
angsd -i $BAM -r X:5000000-154900000 -doCounts 1 -iCounts 1 -minMapQ 30 -minQ 30 -out
↪$OUT
```

Here, I assume that the X chromosome is called `X`. If in your bam file it's called `chrX`, you need to replace the region specification in the `-r` flag above. Note that the range 5Mb-154Mb is used in the example in the website, so I just copied it here. The *$OUT* file above actually denotes a filename-prefix, since there will be several output files from this command, which attach different file-endings after the given prefix.

To loop this command again over all samples, write a shell script as shown above, check the correct commands via an `echo` command and if they are correct, submit them using `sbatch`. My script looks like this:

```
#!/usr/bin/env bash

BAMDIR=/data/schiffels/MyProject/mergedBams.backup
OUTDIR=/data/schiffels/GAworkshop/xContamination
mkdir -p $OUTDIR

for SAMPLE in $(ls $BAMDIR); do
    BAM=$BAMDIR/$SAMPLE/$SAMPLE.mapped.sorted.rmdup.bam
    OUT=$OUTDIR/$SAMPLE.angsdCounts
    CMD="angsd -i $BAM -r X:5000000-154900000 -doCounts 1 -iCounts 1 -minMapQ 30 -
↪minQ 30 -out $OUT"
    echo "$CMD"
    # sbatch -o $OUTDIR/$SAMPLE.angsdCounts.log --wrap="$CMD"
done
```

This should run very fast. Check whether the output folder is populated with non-empty files. You cannnot look at them easily because they are binary files.

The second step in ANGSD is the actual contamination estimation. Here is the command line recommended in the documentation:

```
/projects1/tools/angsd_0.910/misc/contamination -a $PREFIX.icnts.gz \
-h /projects1/tools/angsd_0.910/RES/HapMapChrX.gz 2> $OUT
```

Here, the executable is given with the full path because it is somewhat hidden. The `$PREFIX` variable should be replaced by the output-file prefix given in the previous (allele counting) command for the same sample. The HapMap file is provided by ANGSD and contains global allele frequency estimates used for the contamination calculation. Note that here we are not piping the standard out into the output file `$OUT`, but the standard error, indicated in bash via the special pipe `2>`. The reason is that this ANGSD-program writes its results into the standard error rather than the standard output.

Again, you have to loop this through all samples like this:

```
#!/usr/bin/env bash

BAMDIR=/data/schiffels/MyProject/mergedBams.backup
OUTDIR=/data/schiffels/GAworkshop/xContamination
mkdir -p $OUTDIR

```

```
for SAMPLE in $(ls $BAMDIR); do
    PREFIX=$OUTDIR/$SAMPLE.angsdCounts
    OUT=$OUTDIR/$SAMPLE.xContamination.out
    HAPMAP=/projects1/tools/angsd_0.910/RES/HapMapChrX.gz
    CMD="/projects1/tools/angsd_0.910/misc/contamination -a $PREFIX.icnts.gz -h
↪$HAPMAP 2> $OUT"
    echo "$CMD"
    # sbatch --mem=2000 -o $OUTDIR/$SAMPLE.xContamination.log --wrap="$CMD"
done
```

If this worked correctly, you should now have a contamination estimate for each sample. For a single sample, the output looks a bit messy, but the last line should read:

```
Method2: new_llh Version: MoM:0.072969 SE(MoM):5.964563e-02 ML:0.079651 SE(ML):7.
↪892058e-16
```

This is the line indicating the contamination estimate using the "Methods of Moments" (MoM), and its standard error SE(MoM). You can grep all those lines:

```
adminschif@cdag1 /data/schiffels/GAworkshop/xContamination $ grep 'Method2: new_llh'␣
↪*.out
JK2131udg.xContamination.out:Method2: new_llh Version: MoM:0.285843 SE(MoM):3.993658e-
↪02 ML:0.281400 SE(ML):4.625781e-14
JK2132udg.xContamination.out:Method2: new_llh Version: MoM:0.133319 SE(MoM):9.339797e-
↪02 ML:0.140492 SE(ML):0.000000e+00
JK2133udg.xContamination.out:Method2: new_llh Version: MoM:0.159191 SE(MoM):4.549252e-
↪02 ML:0.160279 SE(ML):8.657070e-15
JK2134udg.xContamination.out:Method2: new_llh Version: MoM:-0.008918 SE(MoM):4.
↪884321e-03 ML:-0.003724 SE(ML):9.784382e-17
...
```

You now want to include those results into your Excel table with the sex determination estimates. Copy them over to your laptop like shown above, in my case:

```
mkdir -p ~/Data/GAworkshop/contamination
scp adminschif@cdag1.cdag.shh.mpg.de:/data/schiffels/GAworkshop/xContamination/*.
↪xContamination.out ~/Data/GAworkshop/contamination/
```

and you can now generate a simpler output using a little bash script like this:

```
#!/usr/bin/env bash

printf "SAMPLE\tCONTAM\tSE\n"
for FILENAME in $(ls ~/Data/GAworkshop/contamination/*.xContamination.out); do
    SAMPLE=$(basename $FILENAME .xContamination.out)
    CONTAM=$(grep 'Method2: new_llh' $FILENAME | cut -d' ' -f4 | cut -d: -f2)
    SE=$(grep 'Method2: new_llh' $FILENAME | cut -d' ' -f5 | cut -d: -f2)
    printf "$SAMPLE\t$CONTAM\t$SE\n"
done
```

If you run this, you may find that in some cases the output is empty, because angsd failed. You should then go back and check - for those samples - the **.log* output from the contamination run above to see what was the reason for failure. In some cases, SLURM killed the job because it exceeded memory. You should then increase the memory set in the `--mem` flag in *sbatch*. In other cases, angsd failed for unknown reasons... nothing we can do about currently.

Finally, you can use this table, feed it into Excel and find male samples with low contamination to proceed with in the analysis.

CHAPTER 4

Genotype Calling from Bam Files

In this session we will process the BAM files of your samples to derive genotype calls for a selected set of SNPs. The strategy followed in this workshop will be to analyse (ancient) test samples together with a large set of modern reference populations. While there are several choices on which data set to use as modern reference, here we will use the Human Origins (HO) data set, published in Lazaridis et al. 2014, which consists of more than 2,300 individuals world-wide human populations, which is a good representation of global human diversity. Those samples were genotyped at around 600,000 SNP positions.

For our next step, we therefore need to use the list of SNPs in the HO data set and for each test individual call a genotype at these SNPs if possible. Following Haak et al. 2015, we will not attempt to call diploid genotypes for our test samples, but will simply pick a single read covering each SNP positions and will represent that sample at that position by a single haploid genotype. Fortunately, the population genetic analysis tools are able to deal with this pseudo-haploid genotype data.

Genotyping the test samples

The main work horse for this genotyping step will be `simpleBamCaller`, a program I wrote to facilitate this type of calling. You will find it at `/projects1/tools/workshop/2016/GenomeAnalysisWorkshop/simpleBamCaller`. You should somehow put this program in your path, e.g. by adding the directory to your `$PATH` environment variable. An online help for this tool is printed when starting it via `simpleBamCaller -h`.

A typical command line for chromosome 20 may look like:

```
simpleBamCaller -f $SNP_FILE -c 20 -r $REF_SEQ -o EigenStrat -e $OUT_PREFIX $BAM1
↪$BAM2 $BAM3 > $OUT_GENO
```

Let's go through the options one by one:

1. `-f $SNP_FILE`: This option gives the file with the list of SNPs and alleles. The program expects a three-column format consisting of chromosome, position and the two alleles separated by a comma. For the Human Origins SNP panel we have prepared such a file. You can find it at `/projects1/users/schiffels/PublicData/HumanOriginsData.backup/EuropeData.positions.txt`. In case you have

aligned your samples to HG19, which uses chromosome names start with `chr`, you need to use the file `EuropeData.positions.altChromName.txt` in the same directory.

2. `-c 20` The chromosome name. In simpleBamCaller you need to call each chromosome separately. It's anyway recommended to parallelise across chromosomes to speed up calling. Note that if your BAM files are aligned to HG19, you need to give the correct chromosome name, i.e. `chr20` in this case. However, note that the Human Origins data set uses just plain numbers as chromosomes, so you want to use the option `--outChrom 20` in simpleBamCaller to convert the chromosome name to plain numbers.
3. `-r $REF_SEQ`. This is the reference sequence that was used to align your bam file to. You find them under `/projects1/Reference_Genomes`.
4. `-o EigenStrat`: This specifies that the output should be in EigenStrat format, which is the format of the HO data set, with which we need to merge our test data. EigenStrat formatted data sets consist of three files: i) a SNP file, with the positions and alleles of each SNP
5. `-e $OUT_PREFIX`. This will be the file prefix for the
6. `$BAM1 $BAM2 ...` This are the bam files of your test samples. Note that you will call simultaneously in all samples to generate a joined EigenStrat files for a single chromosome for all samples.

You should try the command line, perhaps piping the result into *head* to only look at the first 10 lines of the genotype-output. You may encounter an error of the sort "inconsistent number of genotypes. Check that bam files have different readgroup sample names". In that case you will first have to fix your bam files (see section below). If you do not encounter this error, you can skip the next section.

Note: Fixing the bam read group

For multi-sample calling, bam files need to contain a certain flag in their header specifying the read group. This flag also contains the name of the sample whose data is contained in the BAM file. Standard mapping pipelines like BWA and EAGER might not add this flag in all cases, so we have to do that manually. Fortunately, it is relatively easy to do this using the Picard software. The documentation of the `AddOrReplaceReadGroups` tool shows an example command line, which in our case can look like this:

```
picard AddOrReplaceReadGroups I=$INPUT_BAM O=$OUTPUT_BAM RGLB=$LIB_NAME RGPL=
↪$PLATFORM RGPU=$PLATFORM_UNIT RGSM=$NAME
```

Here, the only really important parameter is `RGSM=$NAME`, where `$NAME` should be your sample name. The other parameters are required but not terribly important: `$LIB_NAME` can be some library name, or you can simply use the same name as your sample name. `$PLATFORM` should be `illumina` but can be any string, `$PLATFORM_UNIT` can be `unit1` or something like that.

If you encounter any problems with validation, you can additionally set the parameter `VALIDATION_STRINGENCY=SILENT`.

In order to run this tool on all your samples you should submit jobs via a shell script, which in my case looked like this:

```
#!/usr/bin/env bash

BAMDIR=/data/schiffels/MyProject/mergedBams.backup

for SAMPLE in $(ls $BAMDIR); do
    BAM=$BAMDIR/$SAMPLE/$SAMPLE.mapped.sorted.rmdup.bam
    OUT=$BAMDIR/$SAMPLE/$SAMPLE.mapped.sorted.rmdup.fixedHeader.bam
    CMD="picard AddOrReplaceReadGroups I=$BAM O=$OUT RGLB=$SAMPLE RGPL=illumina␣
↪RGPU=HiSeq RGSM=$SAMPLE"
    echo "$CMD"
    # sbatch -c 2 -o $OUTDIR/$SAMPLE.readgroups.log --wrap="$CMD"
done
```

As in the previous session, write your own script like that, make it executable using `chmod u+x`, run it, check that the printed commands look correct, and then remove the comment from the *sbatch* line to submit the jobs.

Continuing with genotyping

If the calling pipeline described above works with the fixed bam files, the first lines of the output (using `head`) should look like this:

```
[warning] tag DPR functional, but deprecated. Please switch to `AD` in future.
[mpileup] 5 samples in 5 input files
<mpileup> Set max per-file depth to 1600
00220
22000
00220
20922
22092
22090
20000
29992
22292
20090
```

The first three lines are just output to stderr (they won't appear in the file when you pipe via `> $OUT_FILE`). The 10 last lines are the called genotypes on the input SNPs. Here, a 2 denotes the reference allele, 0 denotes the alternative allele, and 9 denotes missing data. If you also look at the first 10 lines of the **.snp.txt* file, set via the *-e* option above, you should see something like this:

```
20_97122    20      0       97122   C       T
20_98930    20      0       98930   G       A
20_101362   20      0       101362  G       A
20_108328   20      0       108328  C       A
20_126417   20      0       126417  A       G
20_126914   20      0       126914  C       T
20_126923   20      0       126923  C       A
20_127194   20      0       127194  T       C
20_129063   20      0       129063  G       A
20_140280   20      0       140280  T       C
```

which is the EigenStrat output for the SNPs. Here, the second column is the chromosome, the fourth column is the position, and the 5th and 6th are the two alleles. Note that simpleBamCaller automatically restricts the calling to the two alleles given in the input file. EigenStrat output also generates an `*.ind.txt` file, again set via the `-e` prefix flag. We will look at it later.

OK, so now that you know that it works in principle, you need to again write a little shell script that performs this calling for all samples on all chromosomes. In my case, it looks like this:

```
#!/usr/bin/env bash

BAMDIR=/data/schiffels/MyProject/mergedBams.backup
REF=/projects1/Reference_Genomes/Human/hs37d5/hs37d5.fa
SNP_POS=/projects1/users/schiffels/PublicData/HumanOriginsData.backup/EuropeData.
↪positions.autosomes.txt
OUTDIR=/data/schiffels/MyProject/genotyping
```

```
mkdir -p $OUTDIR

BAM_FILES=$(ls $BAMDIR/*/*.mapped.sorted.rmdup.bam | tr '\n' ' ')
for CHR in {1..22}; do
    OUTPREFIX=$OUTDIR/MyProject.HO.eigenstrat.chr$CHR
    OUTGENO=$OUTPREFIX.geno.txt
    CMD="simpleBamCaller -f $SNP_POS -c $CHR -r $REF -o EigenStrat -e $OUTPREFIX $BAM_
↪FILES > $OUTGENO"
    echo "$CMD"
    # sbatch -c 2 -o $OUTDIR/$SAMPLE.sexDetermination.log --mem=2000 --wrap="$CMD"
done
```

Note that I am now looping over 22 chromosomes instead of samples (as we have done in other scripts). The line beginning with `BAM_FILES=...` looks a bit cryptic. The syntax `$(...)` will put the output of the command in brackets into the `BAM_FILES` variable. The `tr '\n' ' '` bit takes the listing output from `ls` and convert new lines into spaces, such that all bam files are simply put behind each other in the `simpleBamCaller` command line. Before you submit, look at the output of this script by piping it into `less -S`, which will not wrap the very long command lines and allows you to inspect whether all files are given correctly. When you are sure it's correct, remove the comment from the `sbatch` line and comment out the `echo` line to submit.

Note: A word about DNA damage

If the samples you are analysing are ancient samples, the DNA will likely contain DNA damage, so C->T changes which are seen as C->T and G->A substitutions in the BAM files. There are two ways how to deal with that. First, if your data is not UDG-treated, so if it contains the full damage, you should restrict your analysis to Transversion SNPs only. To that end, simply add the `-t` flag to `simpleBamCaller`, which will automatically output only transversion SNPs. If your data is UDG-treated, you will have much less damage, but you can still see damaged sites in particular in the boundary of the reads in your BAM-file. In that case, you probably want to make a modified bam file for each sample, where the first 2 bases on each end of the read are clipped. A useful tool to do that is TrimBam, which we will not discuss here, but which I recommend to have a look at if you would like to analyse Transition SNPs from UDG treated libraries.

Merging across chromosomes

Since the EigenStrat format consists of simple text files, where rows denote SNPs, we can simply merge across all chromosomes using the UNIX `cat` program. If you `cd` to the directory containing the eigenstrat output files for all chromosomes and run `ls` you should see something like:

```
MyProject.HO.eigenstrat.chr10.geno.txt
MyProject.HO.eigenstrat.chr10.ind.txt
MyProject.HO.eigenstrat.chr10.snp.txt
MyProject.HO.eigenstrat.chr11.geno.txt
MyProject.HO.eigenstrat.chr11.ind.txt
MyProject.HO.eigenstrat.chr11.snp.txt
...
```

A naive way to merge across chromosomes might then simply be:

```
cat MyProject.HO.eigenstrat.chr*.geno.txt > MyProject.HO.eigenstrat.allChr.geno.txt
cat MyProject.HO.eigenstrat.chr*.snp.txt > MyProject.HO.eigenstrat.allChr.snp.txt
```

(Note that the `*.ind.txt` file will be treated separately below). However, these `cat` command lines won't do the job correctly, because they won't merge the chromosomes in the right order. To ensure the correct order, I recommend printing all files in a loop in a sub-shell like this:

```
(for CHR in {1..22}; do cat MyProject.HO.eigenstrat.chr$CHR.geno.txt; done) >␣
↪MyProject.HO.eigenstrat.allChr.geno.txt
(for CHR in {1..22}; do cat MyProject.HO.eigenstrat.chr$CHR.snp.txt; done) >␣
↪MyProject.HO.eigenstrat.allChr.snp.txt
```

Here, each `cat` command only outputs one file at a time, but the entire loop runs in a sub-shell denoted by brackets, whose output will be piped into a file.

Now, let's deal with the `*.ind.txt` file. As you can see, `simpleBamCaller` created one `*.ind.txt` per chromosome, but we only need one file in the end, so I suggest you simply copy the one from chromosome 1. But at the same time, we want to fix the third column of the `*.ind.txt` file to something more tellinf than "Unknown". So copy the file from chromosome 1, and open it in an editor, and replace all "Unknown" to the population name of that sample. The sex (2nd column) is not necessary.

Note: You should now have three final eigenstrat files for your test samples: A `*.geno.txt` file, a `*.snp.txt` file and a `*.ind.txt` file.

Merging the test genotypes with the Human Origins data set

As last step in this session, we need to merge the data set containing your test samples with the HO reference panel. To do this, we will use the `mergeit`-program from the Eigensoft package, which is already installed on the cluster.

This program needs a parameter file that - in my case - looks like this:

```
geno1:      /projects1/users/schiffels/PublicData/HumanOriginsData.backup/EuropeData.
↪eigenstratgeno.txt
snp1:       /projects1/users/schiffels/PublicData/HumanOriginsData.backup/EuropeData.
↪simple.snp.txt
ind1:       /projects1/users/schiffels/PublicData/HumanOriginsData.backup/EuropeData.
↪ind.txt
geno2:      /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.allChr.geno.
↪txt
snp2:       /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.allChr.snp.
↪txt
ind2:       /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.ind.txt
genooutfilename:    /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.
↪merged.geno.txt
snpoutfilename:     /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.
↪merged.snp.txt
indoutfilename:     /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.
↪merged.ind.txt
outputformat: EIGENSTRAT
```

If you have such a parameter file, you can run `mergeit` simply like this:

```
mergeit -p $PARAM_FILE
```

and to submit to SLURM:

```
sbatch -o $LOG --mem=2000 --wrap="mergeit -p $PARAM_FILE"
```

where `$PARAM_FILE` should be replaced by your parameter file, of course.

To test whether it worked correctly, you should check the resulting “indoutfilename” as specified in the parameter file, to see whether it contains both the individuals of the reference panel and the those of your test data set.

Note that the output of the `mergeit` program is by default a binary format called “PACKEDANCESTRYMAP”, which is fine for smartpca but not for other analyses we’ll be doing later, so I explicitly put the outputformat in the parameter file to force the output to be eigenstrat.

CHAPTER 5

Principal Component Analysis

In this lesson we'll make a principal component plot. For that we will use the program `smartpca`, again from the Eigensoft package. The recommended way to perform PCA involving low coverage test samples, is to construct the Eigenvectors only from the high quality set of modern samples in the HO set, and then simply project the ancient or low coverage samples onto these Eigenvectors. This allows one to project even samples with as few as 10,000 SNPs into PCA plot (compare with ~600,000 SNPs for HO samples).

Running SmartPCA

So the first thing to decide is on the populations used to construct the PCA. You can find a complete list of HO population at `/projects1/users/schiffels/PublicData/HumanOriginsData.backup/HO_populations.txt`. You can in principle use all of these populations to construct the EigenVectors. Note however that this will fill the first principal components with global human diversity axes, like African/Non-African, Asian/Europe, Native Americans... So it depends on your particular research question on whether you want to narrow down the populations used to construct the PCA. For example, if you are working on Native American samples, you may want to consider running a PCA with only Native American and perhaps Siberian populations. You can also make several runs with different subsets of populations of course.

Note: In any case, if you want to restrict the samples used for contructing the PCA, you should copy the populations file above to your directory and modify it accordingly, i.e. remove populations you do not want. At the very least, I recommend removing the following "populations": Chimp, Denisovan, Gorilla, hg19ref, Iceman, LaBrana, LBK, Loschbour, MA1, Macaque, Marmoset, Mezmaiskaya, Motala, Orangutan, Saqqaq, Swedish Farmer, Swedish HunterGatherer, Vindija light.

Now we need to build a parameter file for `smartpca`. Mine looks like this:

```
genotypename:       /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.
↪merged.geno.txt
snpname:    /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.merged.snp.
↪txt
indivname:  /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.merged.ind.
↪txt
```

```
evecoutname:        /data/schiffels/GAworkshop/pca/MyProject.HO.merged.pca.evec.txt
evaloutname:        /data/schiffels/GAworkshop/pca/MyProject.HO.merged.pca.eval.txt
poplistname:        /home/adminschif/GAworkshop/pca_populations.txt
lsqproject: YES
```

The first three lines contain the three genotype files you generated from merging your test samples with the HO data set, so the output of the `mergeit` program. The next two lines are two output files, and you have to make sure the directory where these two files will be written into exists. The next line contains a file with the list of populations that you want to use to construct the PCA, as discussed above. The last line contains a flag that is recommended for low coverage and ancient data.

You can now run `smartpca` on that parameter file and submit to SLURM via:

```
sbatch --mem=8000 -o /data/schiffels/GAworkshop/pca/smartpca.log --wrap="smartpca -p␣
↪smartpca.params.txt"
```

Here I reserved 8GB of memory, which I would recommend for a large data set such as HO. Once finished, transfer the resulting `*.evec.txt` file back to your laptop.

Plotting the results

There are several ways to make nice publication-quality plots (Excel is usually not one of them). Popular tools include R, matplotlib. A relatively new development which is highly recommended is the Jupyter Notebook, which can be used with both R, matplotlib and many other scientific compute environments. I personally also heavily use the commercial software DataGraph, you can download a free demo if you are interested. To keep it simple, here we will simply use R, because it works right out of the box and has an easy installation. So please go ahead and install R from the website if you don't have it already installed.

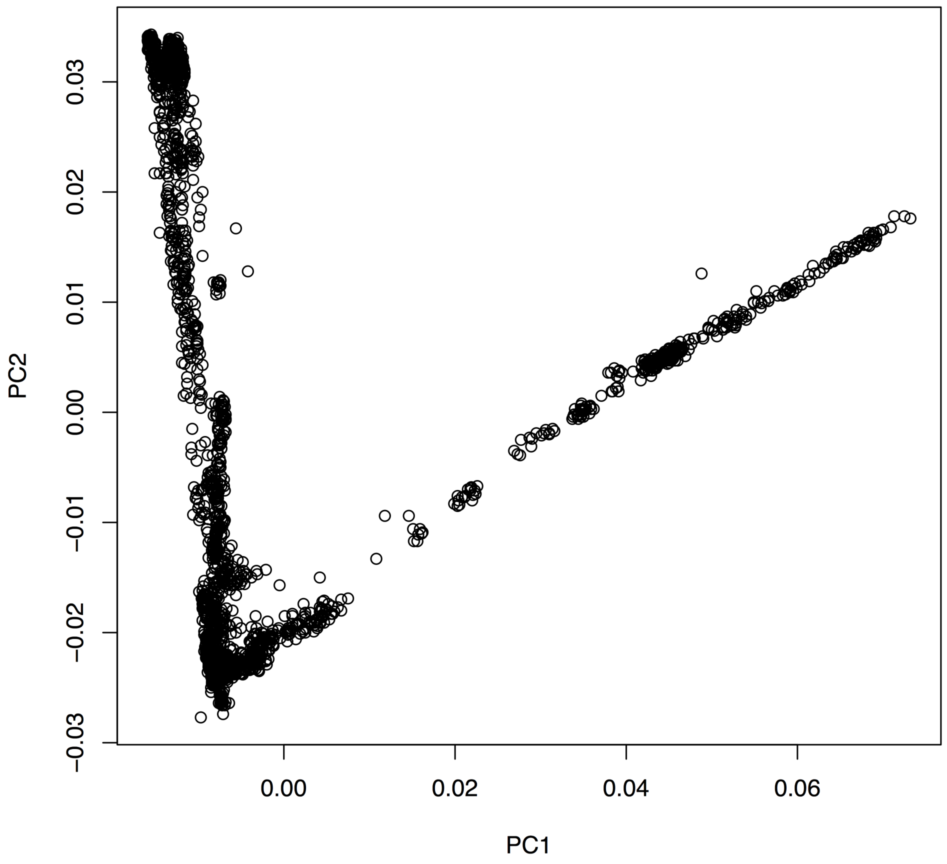
When you startup R, you get the console, into which you can interactively type commands, including plot commands, and look at the results on a screen. Here are my first two commands:

```
fn = "~/Data/GAworkshop/pca/MyProject.HO.merged.pca.evec.txt"
evecDat = read.table(fn, col.names=c("Sample", "PC1", "PC2", "PC3", "PC4", "PC5",
                                     "PC6", "PC7", "PC8", "PC9", "PC10", "Pop"))
```

where `fn` obviously should point to the `*.evec.txt` file produced from `smartpca`. The `read.table` command reads the data into a so called "DataFrame", which is pretty much an ordinary table with some extra features. We explicitly say what the 12 column names should be in the command, as you can see. The first column is the Sample name, the last is the population name, and the middle 10 columns denote the 10 first principle components for all samples. You can have a look at the data frame by typing `head(evecDat)`, which should yield:

```
    Sample    PC1    PC2     PC3     PC4     PC5     PC6     PC7     PC8     PC9   ␣
↪PC10      Pop
1   SA1004 0.0549 0.0100 -0.0502 -0.0016  0.0003  0.0009  0.0016  0.0409 -0.0368 -0.
↪0625  Khomani
2    SA064 0.0502 0.0083 -0.0619 -0.0038  0.0020  0.0016  0.0055  0.0664 -0.0424 -0.
↪0878  Khomani
3    SA073 0.0552 0.0110 -0.0600 -0.0043  0.0007 -0.0001  0.0007  0.0411 -0.0393 -0.
↪0868  Khomani
4    SA078 0.0465 0.0058 -0.0749 -0.0024 -0.0041  0.0011 -0.0101  0.0307 -0.0330 -0.
↪0671  Khomani
5    SA083 0.0418 0.0047 -0.0631 -0.0040  0.0041  0.0004  0.0035  0.0620 -0.0372 -0.
↪0855  Khomani
6 BOT2.031 0.0668 0.0148 -0.0888 -0.0040 -0.0011 -0.0027 -0.0068 -0.0310  0.0096  0.
↪0072 Taa_West
```

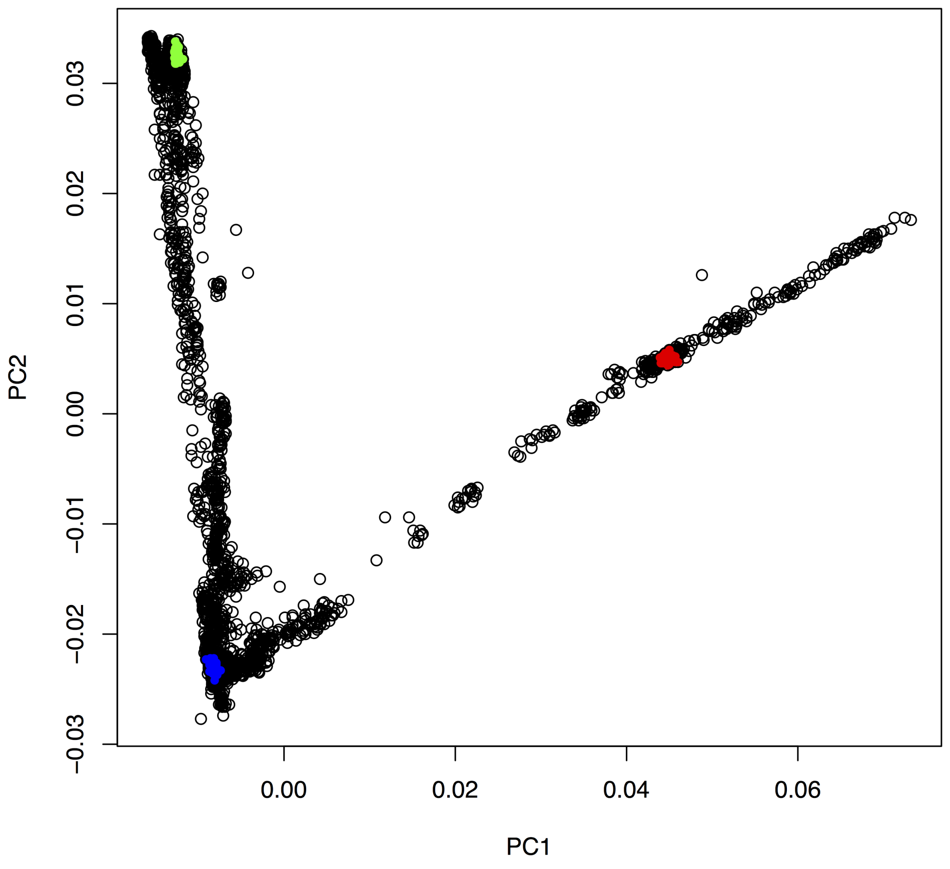
You see that it's pretty much a table. You can now very easily produce a plot of PC1 vs. PC2, by typing `plot(evecDat$PC1, evecDat$PC2, xlab="PC1", ylab="PC2")`, which in my case yields a boring figure like this:



Now, obviously, we would like to highlight the different populations by color. A quick and dirty solution is to simply plot different subsets of the data on top of each other, like this:

```
plot(evecDat$PC1, evecDat$PC2, xlab="PC1", ylab="PC2")
d = evecDat[evecDat$Pop=="Yoruba",]
points(d$PC1, d$PC2, col="red", pch=20)
d = evecDat[evecDat$Pop=="French",]
points(d$PC1, d$PC2, col="blue", pch=20)
d = evecDat[evecDat$Pop=="Han",]
points(d$PC1, d$PC2, col="green", pch=20)
```

You can copy and paste all those lines simultaneously into the console, by the way. This sequence of commands gives us:



OK, but how do we systematically show all the interesting populations? In principle, R makes this easily possible: Instead of choosing a single color and symbols (the `col` and `pch` options), you can give R vectors to these options, which contain one value for each sample. To make this clearer, run `plot(evecDat$PC1, evecDat$PC2, col=evecDat$Pop)`, which should produce a _very_ colorful, but also useless, plot, where each population has its own color (although R cycles only 8 colors, so you will have every color used for many populations). OK, this is not useful. We should have a broader categorization into continental groups.

The way I have come up with first involves making a new tabular file with two columns, to denote the continental groups that the populations are in, like this:

```
BantuKenya  African
BantuSA     African
Canary_Islanders    African
Dinka       African
Ethiopian_Jew       African
Mayan       NativeAmerican
Mixe        NativeAmerican
Mixtec      NativeAmerican
Quechua     NativeAmerican
Surui       NativeAmerican
Ticuna      NativeAmerican
Zapotec     NativeAmerican
Algerian    NorthAfrican
Egyptian    NorthAfrican
```

```
Libyan_Jew  NorthAfrican
Moroccan_Jew        NorthAfrican
Tunisian    NorthAfrican
Tunisian_Jew        NorthAfrican
...
```

The names in the first column should be taken from the population names in your merged `*.ind.txt` file that you input to `smartpca`. An example file can be found in the Google Drive folder under `HO_popGroups.txt`. You can load this file into a data frame in R via:

```
popGroups=read.table("~/Google_Drive/Projects/GAworkshopScripts/HO_popGroups.txt",␣
↪col.names=c("Pop", "PopGroup"))
```

You can again convince yourself that it worked by typing `head(popGroups)`. We can now make use of a very convenient feature in R which lets us easily merge two data frames together. What we need is a new data frame which consists of the `evecDat` data frame, but with an additional column indicating the continental group. This involves a lookup in `popGroups` for every population in `evecDat`. This command does the job:

```
mergedEvecDat = merge(evecDat, popGroups, by="Pop")
```

You can see via `head(mergedEvecDat)`:

```
        Pop Sample     PC1     PC2     PC3     PC4    PC5     PC6    PC7     PC8    ␣
↪PC9    PC10 PopGroup
1 Abkhasian abh107 -0.0080 -0.0211 -0.0040 -0.0003 0.0073 -0.0025 0.0096 -0.0204 -0.
↪0052 -0.0126    Asian
2 Abkhasian abh133 -0.0077 -0.0217 -0.0043 -0.0006 0.0073 -0.0022 0.0081 -0.0222 -0.
↪0053 -0.0137    Asian
3 Abkhasian abh119 -0.0077 -0.0214 -0.0041 -0.0009 0.0057 -0.0019 0.0109 -0.0205 -0.
↪0043 -0.0147    Asian
4 Abkhasian abh122 -0.0078 -0.0214 -0.0039 -0.0017 0.0050 -0.0015 0.0082 -0.0171 -0.
↪0042 -0.0116    Asian
5 Abkhasian  abh27 -0.0077 -0.0218 -0.0039 -0.0011 0.0039 -0.0024 0.0076 -0.0205 -0.
↪0055 -0.0121    Asian
6 Abkhasian  abh41 -0.0077 -0.0209 -0.0046 -0.0015 0.0054 -0.0028 0.0047 -0.0208 -0.
↪0078 -0.0130    Asian
```

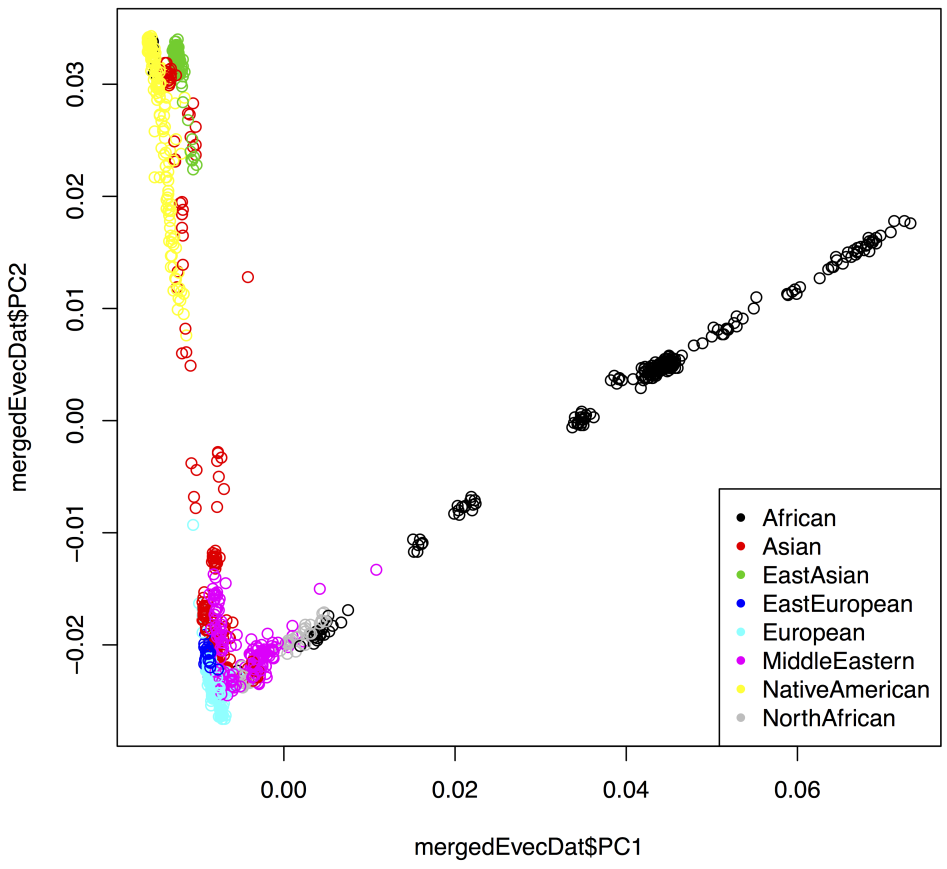
that there now is a new column to the right called `PopGroup`, which correctly contains the group for each sample. Note that this new dataframe only contains rows with populations that are actually in your original `popGroups` data set, so in the file you created. You can see this by running `nrow`:

```
> nrow(mergedEvecDat)
[1] 1306
> nrow(evecDat)
[1] 2257
```

You see that in my case the `mergedEvecDat` only contains 1306 samples, whereas the full data set had 2257 samples. So you can use this to select specific populations you would like to have plotted.

OK, so now, as a first step, we can improve our simple first plot by using the color to indicate the continental group:

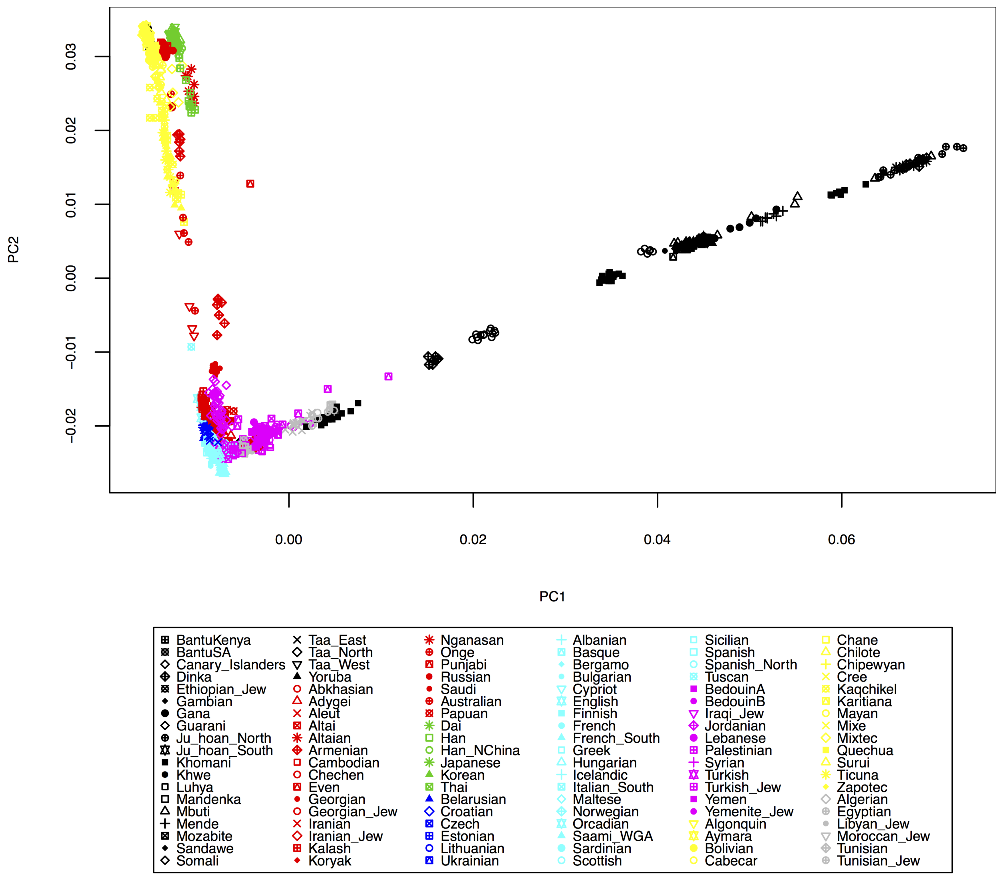
```
plot(mergedEvecDat$PC1, mergedEvecDat$PC2, col=mergedEvecDat$PopGroup)
legend("bottomright", legend=levels(mergedEvecDat$PopGroup),␣
↪col=1:length(levels(mergedEvecDat$PopGroup)), pch=20)
```



The final solution for me was to also separate populations by symbol, which involves a bit more hacking. First, to use different symbols for different populations, you can give a simple vector of symbols to the `plot` command via `pch=as.integer(mergedEvecDat$Pop) %% 24`. The trick here is that first you convert `mergedEvecDat$Pop` to an integer enumerating all populations, and then you use the `modulo` operation to cycle through 24 different numbers. The complete solution in my case looks like this:

```
fn <- "~/Data/GAworkshop/pca/MyProject.HO.merged.pca.evec.txt"
evecDat <- read.table(fn, col.names=c("Sample", "PC1", "PC2", "PC3", "PC4", "PC5",
                                      "PC6", "PC7", "PC8", "PC9", "PC10", "Pop"))
popGroups <- read.table("~/Google_Drive/GA_workshop Jena/HO_popGroups.txt", col.
↪names=c("Pop", "PopGroup"))
popGroupsWithSymbol <- cbind(popGroups, (1:nrow(popGroups)) %% 26)
colnames(popGroupsWithSymbol)[3] = "symbol"
mergedEvecDat = merge(evecDat, popGroupsWithSymbol, by="Pop")

layout(matrix(c(1,2), ncol=1), heights=c(1.5, 1))
par(mar=c(4,4,0,0))
plot(mergedEvecDat$PC1, mergedEvecDat$PC2, col=mergedEvecDat$PopGroup,␣
↪pch=mergedEvecDat$symbol, cex=0.6, cex.axis=0.6, cex.lab=0.6, xlab="PC1", ylab="PC2
↪")
plot.new()
par(mar=rep(0, 4))
legend("center", legend=popGroupsWithSymbol$Pop, col=popGroupsWithSymbol$PopGroup,␣
↪pch=popGroupsWithSymbol$symbol, ncol=6, cex=0.6)
```

which produces:



Of course, here I haven't yet included my test individuals, but you can see easily how to include them in the `HO_popGroups.txt` file. Also, in `plot` you can use the `xlim` and `ylim` options to zoom into specific areas of the plot, e.g. try `xlim=c(-0.01,0.01), ylim=c(-0.03,-0.01)` in the `plot` command above.

CHAPTER 6

Outgroup F3 Statistics

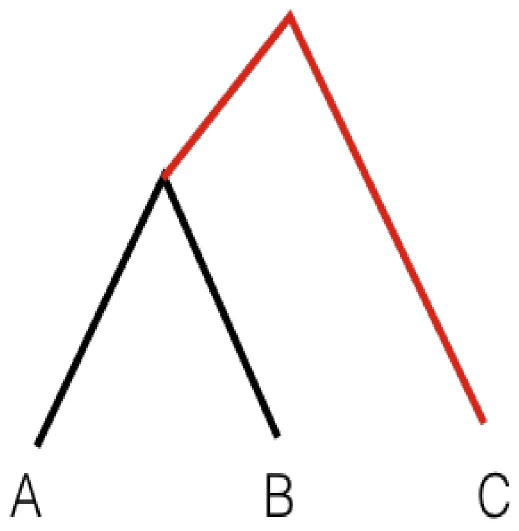
Outgroup F3 statistics are a useful analytical tool to understand population relationships. F3 statistics, just as F4 and F2 statistics measure allele frequency correlations between populations and were introduced by Nick Patterson in his 2012 paper.

F3 statistics are used for two purposes: i) as a test whether a target population (C) is admixed between two source populations (A and B), and ii) to measure shared drift between two test populations (A and B) from an outgroup (C). In this session we'll use the second of these use cases.

F3 statistics are in both cases defined as the product of allele frequency differences between population C to A and B, respectively:

```
F3=<(c-a)(c-b)>
```

Here, <> denotes the average over all genotyped sites, and `a`, `b` and `c` denote the allele frequency in the three populations. Outgroup F3 statistics measure the amount of _shared genetic drift between two populations from a common ancestor. In a phylogenetic tree connecting A, B and C, Outgroup F3 statistics measure the common branch length from the outgroup, here indicated in red:



For computing F3 statistics including error bars, we will use the `qp3Pop` program from Admixtools. You can have a look at the readme for that tool under `/projects1/tools/adminxtools_3.0/README.3PopTest` (the typo "adminx" is actually in the path).

The README and name of the tools is actually geared towards the first use case of F3 described above, the test for admixture. But since the formula is exactly the same, we can use the same tool for Outgroup F3 Statistics as well. One ingredient that you need is a list of population triples. This should be a file with three population names in each row, separated by space, e.g.:

```
JK2134 AA Yoruba
JK2134 Abkhasian Yoruba
JK2134 Adygei Yoruba
JK2134 AG2 Yoruba
JK2134 Albanian Yoruba
JK2134 Aleut Yoruba
JK2134 Algerian Yoruba
JK2134 Algonquin Yoruba
JK2134 Altai Yoruba
JK2134 Altaian Yoruba
...
```

Note that in this case the first population is a single sample, the second loops through all HO populations, and the third one is a fixed outroup, here Yoruba. For Non-African population studies you can use "Mbuti" as outgroup, which is commonly used as an unbiased outgroup to all Non-Africans.

Analysing groups of samples (populations)

If you only analyse a single population, or a few, you can manually create lists of population triples. In that case, first locate the list of all Human Origins populations here: `/projects1/users/schiffels/PublicData/HumanOriginsData.backup/HO_populations.txt`, and construct a file with the desired population triples using an awk-one-liner:

```
awk '{print "YourPopulation", $1, "Mbuti"}' $HO_populations > $OUT
```

Here, "YourPopulation" should be replaced by the population in you `*.ind.txt` file that you want to focus on, and "Mbuti" is the outgroup (pick another one if appropriate). Then, construct a parameter file like this:

```
genotypename:   /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.merged.
↪geno.txt
snpname:   /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.merged.snp.
↪txt
indivname:   /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.ind.txt
popfilename:  <YOUR_POPULATION_TRIPLE_LIST>
```

and run it via

```
qp3Pop -p $PARAMETER_FILE > $OUT
```

Analysing individual samples

In my case, I selected 6 samples that showed low levels of contamination and chose to run them independently through F3 statistics. You may also choose to group test samples together into one population. In my case, I create 6 separate population lists like this:

```
#!/usr/bin/env bash
OUTDIR=/data/schiffels/GAworkshop//data/schiffels/GAworkshop/f3stats
mkdir -p $OUTDIR
for SAMPLE in JK2134 JK2918 JK2888 JK2958 JK2911 JK2972; do
    HO_POPLIST=/projects1/users/schiffels/PublicData/HumanOriginsData.backup/HO_
↪populations.txt
    OUT=$OUTDIR/$SAMPLE.f3stats.poplist.txt
    awk -v s=$SAMPLE '{print s, $1, "Mbuti"}' $HO_POPLIST > $OUT
done
```

Here, the `awk` command loops through all rows in `$HO_POPLIST` and prints it into a new row with the sample name (assigned as variable `s` in awk through a command line option `-v s=$SAMPLE`), and "Mbuti" in last position. If you follow a similar approach of looping through multiple samples, you should check the output poplist files that they are correct.

Similar to the `mergeit` and the `smartpca` programs we have already used, `qp3Pop` requires a parameter file as input. In my case, for the first sample it looks like this:

```
genotypename:   /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.merged.
↪geno.txt
snpname:   /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.merged.snp.
↪txt
indivname:   /data/schiffels/GAworkshop/genotyping/MyProject.HO.eigenstrat.ind.txt
popfilename:  /data/schiffels/GAworkshop//data/schiffels/GAworkshop/f3stats/JK2134.
↪f3stats.poplist.txt
```

Important: The `qp3Pop` program assumes that all population names in the `popfilename` are present in the `*.ind.txt` file of the input data, specifically in the third column of that file, which indicates the population. In my case, I intend to compute a separate statistic for each of my ancient samples individually, rather than an entire population. Therefore, I manually edited the `*.ind.txt` file an artificially assigned each of my individuals its own "population", which is simply called the same as the individual.

The first three lines of the parameter file specify the EIGENSTRAT data set, similar to what we put into the `smartpca` parameter file. The fourth parameter denotes the population list we generated above. In my case, I need to prepare 6 such parameter files and submit them all:

```
#!/usr/bin/env bash

INDIR=/data/schiffels/GAworkshop/genotyping
OUTDIR=/data/schiffels/GAworkshop//data/schiffels/GAworkshop/f3stats
for SAMPLE in JK2134 JK2918 JK2888 JK2958 JK2911 JK2972; do
    GENO=$INDIR/MyProject.HO.eigenstrat.merged.geno.txt
    SNP=$INDIR/MyProject.HO.eigenstrat.merged.snp.txt
    IND=MyProject.HO.eigenstrat.ind.txt
    POPLIST=$OUTDIR/$SAMPLE.f3stats.poplist.txt

    PARAMSFILE=$OUTDIR/$SAMPLE.f3stats.qp3Pop.params.txt
    printf "genotypename:\t$GENO\n" > $PARAMSFILE
    printf "snpname:\t$SNP\n" >> $PARAMSFILE
    printf "indivname:\t$IND\n" >> $PARAMSFILE
    printf "popfilename:\t$POPLIST\n" >> $PARAMSFILE

    LOG=$OUTDIR/$SAMPLE.qp3Pop.log
    OUT=$OUTDIR/$SAMPLE.qp3Pop.out
    sbatch --mem 4000 -o $LOG --wrap="qp3Pop -p $PARAMSFILE > $OUT"
done
```

This should run for 10-20 minutes. When finished, transfer the resulting files to your laptop using `scp`.

Plotting

The output from `qp3Pop` looks like this:

```
parameter file: /tmp/qp3Pop_wrapper35005211521595368
### THE INPUT PARAMETERS
##PARAMETER NAME: VALUE
genotypename: /data/schiffels/MyProject/genotyping/MyProject.onlyTVFalse.HO.merged.
↪geno
snpname: /data/schiffels/MyProject/genotyping/MyProject.onlyTVFalse.HO.merged.snp
indivname: /data/schiffels/MyProject/genotyping/MyProject.noGroups.onlyTVFalse.HO.
↪merged.ind
popfilename: /data/schiffels/MyProject/f3stats/JK2134.f3stats.poplist.txt
## qp3Pop version: 300
nplist: 224
number of blocks for block jackknife: 549
snps: 593655
                      Source 1              Source 2              Target           f_
↪3        std. err           Z     SNPs
 result:                JK2134                    AA              Yoruba       0.
↪026824       0.001010      26.547   56353
 result:                JK2134             Abkhasian              Yoruba       0.
↪147640       0.002229      66.231   56447
 result:                JK2134                Adygei              Yoruba       0.
↪144566       0.002139      67.583   56467
 result:                JK2134                   AG2              Yoruba       0.
↪139170       0.008287      16.794     9499
 result:                JK2134              Albanian              Yoruba       0.
↪149385       0.002321      64.364   56435
 result:                JK2134                 Aleut              Yoruba       0.
↪134388       0.002287      58.768   56431
 result:                JK2134              Algerian              Yoruba       0.
↪116380       0.002052      56.727   56416
```

```
result:                 JK2134            Algonquin                Yoruba       0.
↪126845       0.002526      50.224    56396
...
```

The key rows are the ones starting with `result:`. We can exploit that and select all relevant rows using `grep`. In my case, I can even join the results across all samples using:

```
grep 'result:' *.qp3Pop.out
```

assuming that I am executing this inside the directory where I copied the per-sample result files. When you run this, the output looks like this:

```
JK2134.f3stats.txt: result:                JK2134                 AA             ␣
↪Yoruba       0.026824       0.001010      26.547    56353
JK2134.f3stats.txt: result:                JK2134          Abkhasian             ␣
↪Yoruba       0.147640       0.002229      66.231    56447
JK2134.f3stats.txt: result:                JK2134             Adygei             ␣
↪Yoruba       0.144566       0.002139      67.583    56467
JK2134.f3stats.txt: result:                JK2134                AG2             ␣
↪Yoruba       0.139170       0.008287      16.794     9499
JK2134.f3stats.txt: result:                JK2134           Albanian             ␣
↪Yoruba       0.149385       0.002321      64.364    56435
JK2134.f3stats.txt: result:                JK2134              Aleut             ␣
↪Yoruba       0.134388       0.002287      58.768    56431
JK2134.f3stats.txt: result:                JK2134           Algerian             ␣
↪Yoruba       0.116380       0.002052      56.727    56416
JK2134.f3stats.txt: result:                JK2134          Algonquin             ␣
↪Yoruba       0.126845       0.002526      50.224    56396
JK2134.f3stats.txt: result:                JK2134              Altai             ␣
↪Yoruba       0.004572       0.003126       1.462    48731
JK2134.f3stats.txt: result:                JK2134            Altaian             ␣
↪Yoruba       0.122992       0.002173      56.590    56409
...
```

As you see, we don't want columns 1 and 2. You can use `awk` to filter out only columns 3, 4, 5, 6, 7, 8:

```
grep 'result:' *.qp3Pop.out | awk '{print $3, $4, $5, $6, $7, $8, $9}' > all.qp3Pop.
↪out
```

We can now again load this combined file into R, using:

```
f3dat = read.table("~/Data/GAworkshop/f3stats/all.qp3Pop.out",
           col.names=c("PopA", "PopB", "PopC", "F3", "StdErr", "Z", "SNPs"))
```

Have a look at this via `head(f3dat)`.

Now, in my case, with multiple individuals tested, I first want to look at one particular individual separately. For that, I first create a subset of the data:

```
s = f3dat[f3dat$PopA == "JK2972",]
```

As a second step, we would like to order this in a descending order according to the F3 statistics. Try this:

```
head(s[order(-s$F3),])
```

which will first order `s` according to the `F3` column, and then print out only the first few lines with the highest F3 statistics for that individual. So go and save that new order via:

```
sOrdered = s[order(-s$F3),]
```

OK, so we now want to plot those highest values including error bars. For that we'll need the `errbar` function which first has to be installed. Install the package "Hmisc":

```
install.packages("Hmisc")
```

from a suitable mirror (for me, the Germany mirror didn't work, I succeeded with the Belgian one).

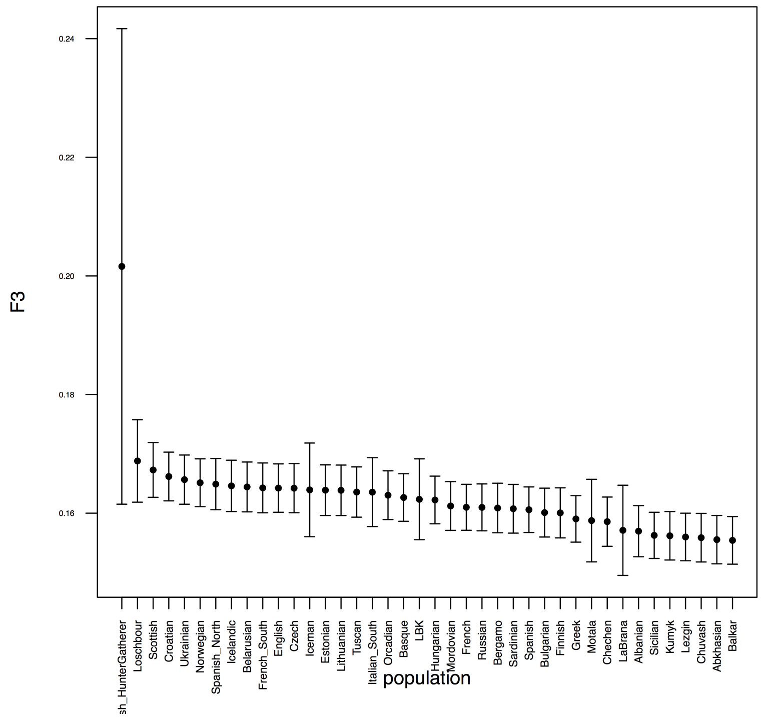
Next, activate that package via `library(Hmisc)`.

You should now be able to view the help for `errbar` by typing `?errbar`.

OK, let's now make a plot:

```
errbar(1:40, sOrdered$F3[1:40],
       (sOrdered$F3+sOrdered$StdErr)[1:40],
       (sOrdered$F3-sOrdered$StdErr)[1:40], pch=20, las=2, cex.axis=0.4, xaxt='n',
       xlab="population", ylab="F3")
axis(1, at=1:40, labels=sOrdered$PopB[1:40], las=2, cex.axis=0.6)
```

which should yield:



Here is the entire R program:

```
f3dat = read.table("~/Data/GAworkshop/f3stats/all.qp3Pop.out",
            col.names=c("PopA", "PopB", "PopC", "F3", "StdErr", "Z", "SNPs"))
s = f3dat[f3dat$PopA == "JK2972",]
sOrdered = s[order(-s$F3),]
errbar(1:40, sOrdered$F3[1:40],
       (sOrdered$F3+sOrdered$StdErr)[1:40],
       (sOrdered$F3-sOrdered$StdErr)[1:40], pch=20, las=2, cex.axis=0.4, xaxt='n',
       xlab="population", ylab="F3")
axis(1, at=1:40, labels=sOrdered$PopB[1:40], las=2, cex.axis=0.6)
```

You can plot this for other individuals/populations by replacing the subset command (`s=...`) with another selected individual/population.

Finally, if you want to print this into a PDF, you can simply surround the above commands by:

```
pdf("myPDF.pdf")
...
dev.off()
```

which will produce a PDF with the graph in it.

CHAPTER 7

ADMIXTURE analysis

Admixture is a very useful and popular tool to analyse SNP data. It performs an unsupervised clustering of large numbers of samples, and allows each individual to be a mixture of clusters.

The excellent documentation of ADMIXTURE can be found at `/projects1/tools/admixture_1.3.0/admixture-manual.pdf`.

Converting to Plink format

ADMIXTURE requires input data in PLINK bed format, unlinke the EigenStrat format that we have already used for both *Principal Component Analysis* and *Outgroup F3 Statistics*. Fortunately, it is easy to convert from Eigenstrat to bed using Eigensoft's tool `convertf`, which is installed on the cluster. To use it, you again have to prepare a parameter file with the following syntax:

```
genotypename: <Eigenstrat.geno>
snpname: <Eigenstrat.snp>
indivname: <Eigenstrat.ind>
outputformat: PACKEDPED
genotypeoutname: <out.bed>
snpoutname: <out.bim>
indivoutname: <out.fam>
```

Note that `PACKEDPED` is `convertf`'s way to describe the `BED` format. Note that for `admixture`, the output file endings are crucial, please use `*.bed`, `*.bim` and `*.fam`. You can then start the conversion by running

```
sbatch --mem=2000 -o $LOG_FILE --wrap="convertf -p $PARAMETER_FILE"
```

This should finish in a few minutes.

Running ADMIXTURE

You can now run `admixture`. The basic syntax is dead-easy: `admixture $INPUT.bed $K`. Here `$K` is a number indicating the number of clusters you want to infer.

How do we know what number of clusters K to use? Well, first of all you should run several numbers of clusters, e.g. all numbers between K=3 and K=12 for a start. Then, `admixture` has a built-in method to evaluate a "cross-validation error" for each K. Computing this "cross-validation error" requires simply to give the flag `--cv` right after the call to `admixture`. However, since this tool already runs for relatively long, we will omit this feature here, but it is strongly recommended to use it for analysis.

One slightly awkward behaviour of `admixture` is that it writes its output simply into the directory where you start it from. Also, you cannot manually name the output files, but they will be called similarly as the input file, but instead of the ending `.bed` it produces two files with endings `.$K.Q` and `.$K.P`. Here, `$K` is again the number of clusters you have chosen.

OK, so here is a template for a submission script:

```
#!/usr/bin/env bash

BED_FILE=...
OUTDIR=...
mkdir -p $OUTDIR

for K in {3..12}; do
    CMD="cd $OUTDIR; admixture -j8 $BED_FILE $K" #Normally you should give --cv as␣
↪first option to admixture
    echo $CMD
    # sbatch -c 8 --mem 12000 --wrap="$CMD"
done
```

Note that the command is sequence of commands: First `cd` into the output directory, then run admixture in order to get the output files where we want them.

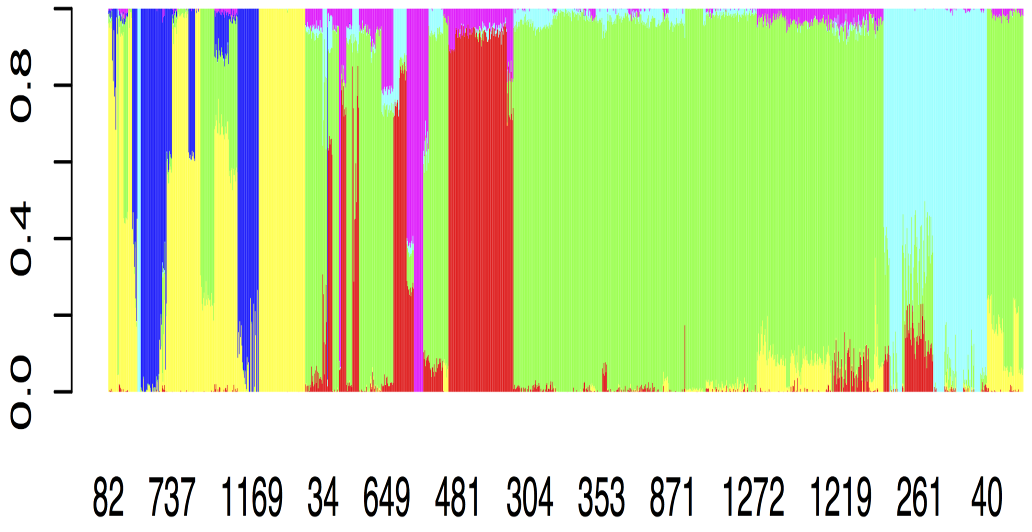
If things run successfully, you should now have a `.Q` and a `.P` file in your output directory for every `K` that you ran.

Plotting in R

Here is the code for making the typical ADMIXTURE-barplot for K=6:

```
tbl=read.table("~/Data/MyProject/admixture/MyProject.HO.merged.6.Q")
indTable = read.table("~/Data/MyProject/admixture/MyProject.HO.merged.ind",
               col.names = c("Sample", "Sex", "Pop"))
popGroups = read.table("~/Google Drive/GA_Workshop Jena/HO_popGroups.txt", col.
↪names=c("Pop", "PopGroup"))

mergedAdmixtureTable = cbind(tbl, indTable)
mergedAdmWithPopGroups = merge(mergedAdmixtureTable, popGroups, by="Pop")
ordered = mergedAdmWithPopGroups[order(mergedAdmWithPopGroups$PopGroup),]
barplot(t(as.matrix(subset(ordered, select=V1:V6))), col=rainbow(6), border=NA)
```

which gives:



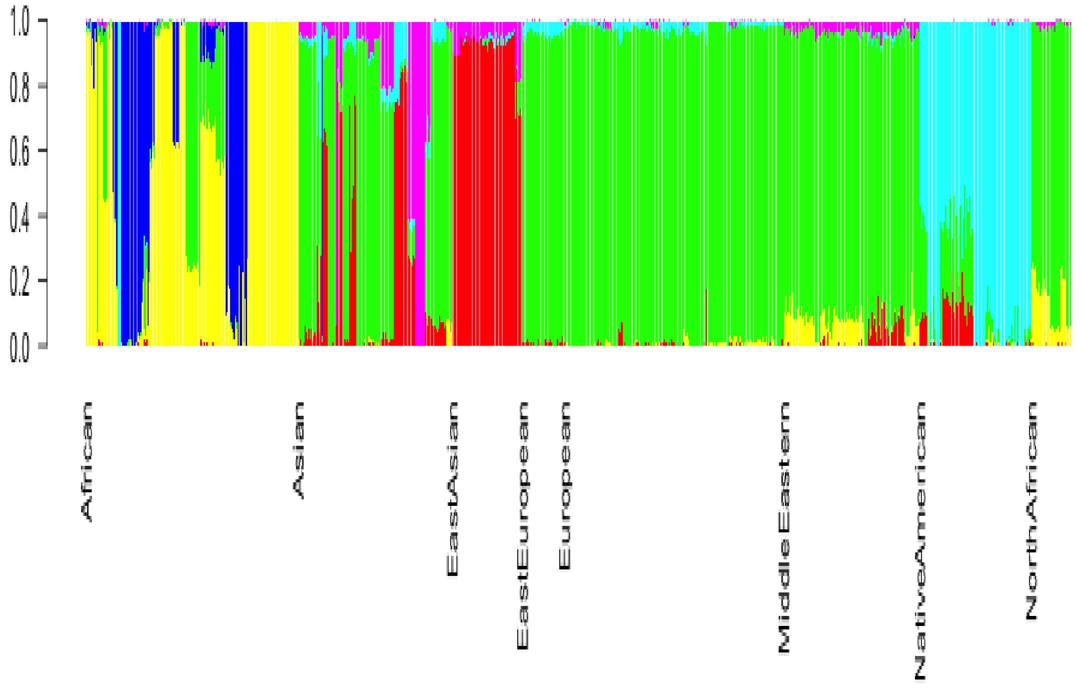
OK, so this is already something, at least the continental groups are ordered, but we would like to also display the population group names below the axis. For this, we'll write a function in R:

```
barNaming <- function(vec) {
    retVec <- vec
    for(k in 2:length(vec)) {
        if(vec[k-1] == vec[k])
            retVec[k] <- ""
    }
    return(retVec)
}
```

and we can then replot:

```
par(mar=c(10,4,4,4))
barplot(t(as.matrix(ordered[,2:7])), col=rainbow(6), border=NA,
        names.arg=barNaming(ordered$PopGroup), las=2)
```

which should give:


1.0
0.8
0.6
0.4
0.2
0.0
African
Asian
EastAsian
EastEuropean
European
MiddleEastern
NativeAmerican
NorthAfrican

CHAPTER 8

Indices and tables

- genindex
- modindex
- search