3RAD Bioinformatics

2024-05-17

This document provides instructions for working with 3RAD FASTQ sequence data received from the core lab. This includes demultiplexing the sequence data based on 3RAD adapters, removing PCR duplicates, checking sequence quality, and running the stacks pipeline, which will all be done on the command line with access to a high performance computing cluster (HPC). The end result will be variant call format (VCF) files ready for genetic analysis.

Resource Links:
Stacks documention
Running 3RAD Analysis Tutorial used as reference
Filtering Vignette used as reference

Section 1: Moving data onto the cluster

1. Download your files from the core lab to your local computer/laptop

2. Set up Globus Connect. Globus Connect is a file transfer service recommended by FASTER for moving files on and off the HPC cluster. Create an account online and downlaod the Globus Connect Personal app, which will run in the background to allow the web app to access your computer files. Open the web app and connect to your computer and the HPC cluster.

4. Upload projectname to your HPC home directory. Projectname is an empty directory structure I put together to help keep your data organized while you get started. It also includes a scripts directory that contains all of the bioinformatics scripts we will be running. Download it here by clicking on the big green "<CODE>" button, then selecting "download ZIP". Change "projectname" to something more meaningful for your project.

3. Transfer files with Globus Connect. Begin a file transfer on Globus Connect by dragging your newly named projectname directory from your computer into your FASTER project directory. Then, drag your sequence files from your computer into the projectname/raw_reads directory.

5. Copy the sequence files into Teams. Make sure you back up your sequence data by uploading it into your project folder in Teams. Eventually your data will be deleted from the core lab, so it is very important that it is backed up in Teams! That way, if you accidentally delete anything critical in your project directory, you can always recover the original data.

Section 2: Demultiplexing sequence data

1. Create plate FASTQs. The core lab sent us 24 raw FASTQ files corresponding to our 12 i7 primers (x2 because each has forward and reverse reads), but we want to combine these files so that we have just two FASTQ files per plate (i.e. Plate1-forward and Plate1-reverse). From the raw_reads directory where the downloaded FASTQ files are located, run the following code one line at a time to concatenate (combine) them:

cat PROJECTNAMEpt1**R1** > ../raw_plates/plate1/PROJECTNAMEpt1_S1_L001_R1_001.fastq.gz
cat PROJECTNAMEpt1**R2** > ../raw_plates/plate1/PROJECTNAMEpt1_S1_L001_R2_001.fastq.gz
cat PROJECTNAMEpt2**R1** > ../raw_plates/plate2/PROJECTNAMEpt2_S1_L001_R1_001.fastq.gz
cat PROJECTNAMEpt2**R2** > ../raw_plates/plate2/PROJECTNAMEpt2_S1_L001_R2_001.fastq.gz

Great! Now we can move on to the stacks pipeline. First, we will demultiplex our samples with the **process_radtags** function. This function will sort all the raw reads from the plate FASTQ files we just made into 192 individual sample FASTQ files by using the i5 and i7 barcodes we included in our adapters. In order to do that, process_radtags requires a barcodes file for each plate to tell it which pair of 3RAD barcodes correspond to which sample name.

2. Create a barcodes file. Navigate to your 3RAD workbook's "barcodes and indexes" tab and locate your 96 samples from plate 1. Copy rows C, D, and E for these samples into a new spreadsheet. Save this spreadsheet as a tab-delimited text file titled ptX_barcodes.txt and move it to the projectname/barcodes directory on the cluster. Repeat for all plates.

3. Edit the demultiplexing slurm script. In order to run intensive code on the cluster, we will need to submit a slurm job, which schedules our code to be run when the necessary computational resources are available. The slurm job for demultiplexing is titled demux.slurm and is stored in the scripts directory. Open it in your text editor of choice (e.g. nano). A copy has been provided below:

demux.slurm

```
#!/bin/bash

##This is a slurm job file that uses process_radtags to demultiplex (demux) individuals using
a barcode file

##NECESSARY JOB SPECIFICATIONS
#SBATCH –job-name=Demux_pt1
#SBATCH –time=05:00:00
#SBATCH –nodes=1
#SBATCH –ntasks=4
#SBATCH –mem=2G
#SBATCH –output=/scratch/user/u.gw200825/ScHH_3RAD/demuxed_fastqs/Demux_pt1.Out.%j

#Executable Lines
module load GCC/12.2.0
module load OpenMPI/4.1.4
module load Stacks/2.64


#move to main ScHH directory
cd /scratch/user/u.gw200825/ScHH_3RAD

#Process radtags
#Remember to change input files each time you do a different plate

process_radtags -P -p ./raw_plates/plate1 -b ./barcodes/Pt1_barcodes.txt -o ./demuxed_fastqs
-i gzfastq
–inline_inline –renz-1 xbaI –renz-2 ecoRI
–adapter_mm 2
–adapter_1 AGATCGGAAGAGCACACGTCTGAACTCCAGTCAC
–adapter_2 AGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT
-c -q -r -D
```

The first section of this script tells the slurm scheduler what resources you need to reserve. You may need to adjust these depending how many samples you are running. You can also give the job a name and designate a location to output a log file.

The next section loads the necessary modules. You may need to replace these with the latest versions if these instructions are out of date. Finally, the process_radtags command is used. Check documentation to see what each option is and edit accordingly, outputting your demultiplexed fastq files in the projectname/demuxed_fastqs directory.

4. Submit the demultiplexing slurm job Once this script has been edited appropriately, submit it with the sbatch command:

```
sbatch demux.slurm
```

This submits it to the slurm scheduler to be run. You can check on it's progress with the squeue command (squeue -u USERNAME). Once finished, you should have demultiplexed files from all your plate 1 individuals in your demuxed_fastqs directory, named according to their OCID. If it failed, the job will finish very quickly. Use the "less" command to read your log files to view any errors for troubleshooting, make sure the samples demultiplexed correctly, and see how many reads were retained. Repeat for all plates.

Section 3: Removing PCR Duplicates

1. Make a popmap file. For our next script, we will need to make a popmap file, which is essentially a list of your samples and which population they came from. To make this popmap file, copy your list of sample names for the plates you are analyzing (192 samples if you have 2 plates) into an excel column and then enter their location in the second column. For negative controls, I use "NEG" as the population. Save this file as a .tsv (tab separated values) file and move it to your projectname/popmap directory. Many of the stacks slurm scripts will reference this file in order to retrieve your sample names.

2. Edit the decloning slurm script. Our next stacks function, **clone_filter**, removes PCR duplicates from each sample by using the iTru5-8N decloning indexes we included in our 3RAD libraries. Open the associated slurm script, declone.slurm, in a text editor.

Here, you will notice some new lines that were not present in our declone.slurm file:

```
#SBATCH –array=1-24

#Set the number of runs that each SLURM task in the array will do
PER_TASK=8

#Calculate the start and end values for the current task
START_NUM=(((SLURM_ARRAY_TASK_ID - 1) * PER_TASK + 1)) END_NUM =((
$SLURM_ARRAY_TASK_ID * $PER_TASK ))

#Print the task and run range:
echo This is task $SLURM_ARRAY_TASK_ID, which will do runs $START_NUM to
$END_NUM
```

This is because this is a slurm *array*. Our demux.slurm file submitted a single job that processed an entire plate. An array, however, submits many jobs at the same time, and each job will be responsible for processing a few files. That way, we can process multiple files at the same time to reduce our wait time. In this file, "–array=1-24" indicates that we want to submit 24 separate jobs. We then assign our PER_TASK variable to 8 to indicate that each job will process 8 files (24x8 =192 sample files). START_NUM and END_NUM

will be automatically calculated so that each job knows which set of files it is responsible for, and these will be printed out to the log to facilitate any necessary troubleshooting.

Make necessary edits to the file paths.

3. Submit the decloning slurm script. Outputs should be directed into the projectname/decloned_fastqs directory.

```
sbatch declone.slurm
```

Once again, use the less command to read the log files to see how many PCR duplicates were removed from your samples.

4. Remove extra forward and reverse read designations. At this point, process_radtags and clone_filter have made some changes to our files we need to remove if we want them to work properly in the rest of the stacks pipeline. First, you'll notice all of your decloned files end with .1.1 or .2.2 instead of simply .1 or .2, which we need to fix. If for some reason your files do not have two .1's or .2's, you can skip this step.

While in your decloned_fastqs directory, type the following into your command line:

```
for f in *.1.1*; do echo mv "f""{f/.1.1/.1}"; done
```

This will print out (echo) the planned file name changes so you can double check to make sure it is working as intended. If all the proposed changes look correct, remove "echo" from the command and re-run it to conduct the renaming.

Repeat this step, but with:

```
for f in *.2.2*; do echo mv "f""{f/.2.2/.2}"; done
```

in order to rename your reverse reads as well.

Next, we need to reformat our fastq headers. Similar to what we just fixed, our previous programs have added a /1 or /2 to the end of all of the headers inside our fastq files. These headers already included a flag for forward and reverse reads (1:N:0 or 2:N:0, respectively) which we now need to remove. To do this, edit and submit the following slurm job, which will decompress the files and reformat the headers:

```
sbatch header_reformat.slurm
```

Check to make sure it worked by viewing the headers in the new fastq files with the less command.

Section 3: FastQC and MultiQC

1. Edit and submit fastqc.slurm. Now that we have demultiplexed and decloned sequence data for each individual, we are going to take a look at the quality with FastQC and multiQC. Edit file paths and submit the slurm job:

```
sbatch fastqc.slurm
```

This should create a single fastqc file for each individual and then a combined multiqc file that includes all individuals.

2. View your multiqc file locally. Move all your fastqc and multiqc files off the cluster and into your computer. Then, open the multiQC html file and take a look at your data! Some things to look at:
1. Negligible runs from negative controls
2. Evenness of read counts among individuals
3. Number of reads per individual
4. Samples passing quality checks other than "Per base sequence content" and "Sequence duplication levels", and "overrepresented sequences" which will fail due to the nature of 3RAD.

Section 4: Preliminary de novo assembly with stacks

First, we will run the stacks pipeline using all default parameters just to get a feel for the software and take a preliminary look at the data. The stacks pipeline consists of ustacks, cstacks, sstacks, tsv2bam, gstacks, and populations. Slurm scripts for all of these commands are located in the scripts folder and will need to be edited to match your files and run conditions. Refer to stacks documentation for help with parameters and options.

1. ustacks
Input fastq files from your decloned_fastqs directory and output into your projectname/stacks directory. Default parameters are M=2, m=3, N=4.

```
sbatch ustacks.slurm
```

2. cstacks
Using all of your inviduals to create a catalog is computationally intensive, so we need to make a new, smaller popmap that will be used for catalog construction. To do this, run the catalog_popmap.txt script from your popmap directory. It will pull 20 (can be changed) random samples from each of your populations to make the catalog popmap. make sure to replace FLA/GAL/SEY with your population names before running.

```
../../scripts/make_catalog_popmap.sh
```

Run cstacks using files from you stacks directory and your newly created catalog_popmap.txt file.

```
sbatch cstacks.slurm
```

3. sstacks
Inputs are taken from your stacks directory by comparing your ustacks files to your newly created catalog file. Outputs ".matches" files into the same stacks directory.

4. tsv2bam
Inputs stacks files from your stacks directory and combines with corresponding reverse reads from your decloned_fastqs directory. Outputs ".bam" files in your stacks directory

5. gstacks
Inputs stacks files from your stacks directory. Outputs "catalog.calls" and "catalog.fa.gz" which contain genotype information for all your samples.

6. populations
Inputs the calaog files created by gstacks. Outputs populations files into your stacks directory. Leave the -r parameter at 0.8, –max-obs-het at 0.75, and change -p depending on the number of populations you have.

These populations output files are the final output of the stacks pipeline. Move them all onto your local computer to work with them in R. The "populations.snps.vcf" file in particular contains all your SNPs and genotype information for all your individuals, and will be used as the foundation for most analyses.

Section 5: Loading data into R

Great work! Now that we have a vcf file of our data, we can begin working on our local computer in R. Create a new R script, give it a title, and save it.

1. Install and load the necessary packages
Some may need to be installed via github (Look up how to do this).

```
require(vcfR)
require(adegenet)
require(strataG)
require(tidyverse)
require(whoa)
require(RADstackshelpR)
require(SNPfiltR)
require(snpR)
```

```
## Warning: package 'data.table' was built under R version 4.3.3
```

2. Read in your new vcf file with vcfR

```
schh_vcf<-read.vcfR("~/sosf_ghri/schh_3rad/stacks/populations.snps.vcf")
```

```
## Scanning file to determine attributes.
## File attributes:
##   meta lines: 14
##   header_line: 15
##   variant count: 10951
##   column count: 201
## Meta line 14 read in.
## All meta lines processed.
## gt matrix initialized.
## Character matrix gt created.
##   Character matrix gt rows: 10951
##   Character matrix gt cols: 201
##   skip: 0
##   nrows: 10951
##   row_num: 0
## Processed variant 1000Processed variant 2000Processed variant 3000Processed variant 4000Processed va
## All variants processed
```

```
#Convert the data to the genind and get gtypes
schh_genind<-vcfR2genind(schh_vcf)
```

```
## Warning in adegenet::df2genind(t(x), sep = sep, ...): Individuals with no
## scored loci have been removed
```

```
schh.gtypes<-genind2gtypes(schh_genind)
```

3. Check duplicates.
This chunk is not executed in this document because it takes a long time to run, but will create a table of all pairs of individuals with >80% similarity.

```
#check for duplicates
schh.dups<-dupGenotypes(schh.gtypes, num.shared = 0.80)
#save pairwise similarity information for future removal of duplicates
save(schh.dups, file = "~/sosf_ghri/schh_3rad/analysis/schh.dups.Rdata")
```

Hopefully, upon viewing the duplicates table, all your intentional replicates have very high similarity, and there are no matching pairs of non-replicates with high similarity (unless these are re-sampling events!) If so, everything is good here and we can move on!

Section 6: Stacks parameter Optimization

Our preliminary run used the default stacks parameters to make sure our data was good and our replicates matched. However, we now need to explore how changing the stacks parameters affects our SNP data to determine the optimal set of parameters to use for our analysis. Read Paris et al. (2017) for an overview of this approach.

We will run stacks a total of 16 times, varying specific parameters while holding other parameters constant.

1. Run the stacks pipeline varying ustacks parameter m = 3-7.
Submit a ustacks job (u.stacks.slurm) with m=3, using the default value of 2 for parameter M. The outputs should go into your stacks/parameter_optimization/m3 directory. Once you have verified that it is working properly (the job has been running for a while), edit your ustacks.slurm script so that m=4 and that output is your stacks/parameter_optimization/m4 directory. Change the job name and log file names as well. Submit this job and then repeat for the remaining values of m.

Once ustacks has finished, you should have ustacks outputs in each of your m3 - m7 directories, which we will use as inputs for the remainder of the stacks pipeline. Check these files and your log files to make sure all five runs were successful.

Then, execute the rest of the stacks pipeline using default values and your desired populations parameters. Keep all outputs within the corresponding m3-m7 folders.

2. Visualize parameter m optimization.
Now that we have populations.snps.vcf files for all of the different parameter combinations, we can visualize which parameters work best. Move all your populations files onto your local computer into separate folders.

Then, run the following code in R to visualize the m parameter:

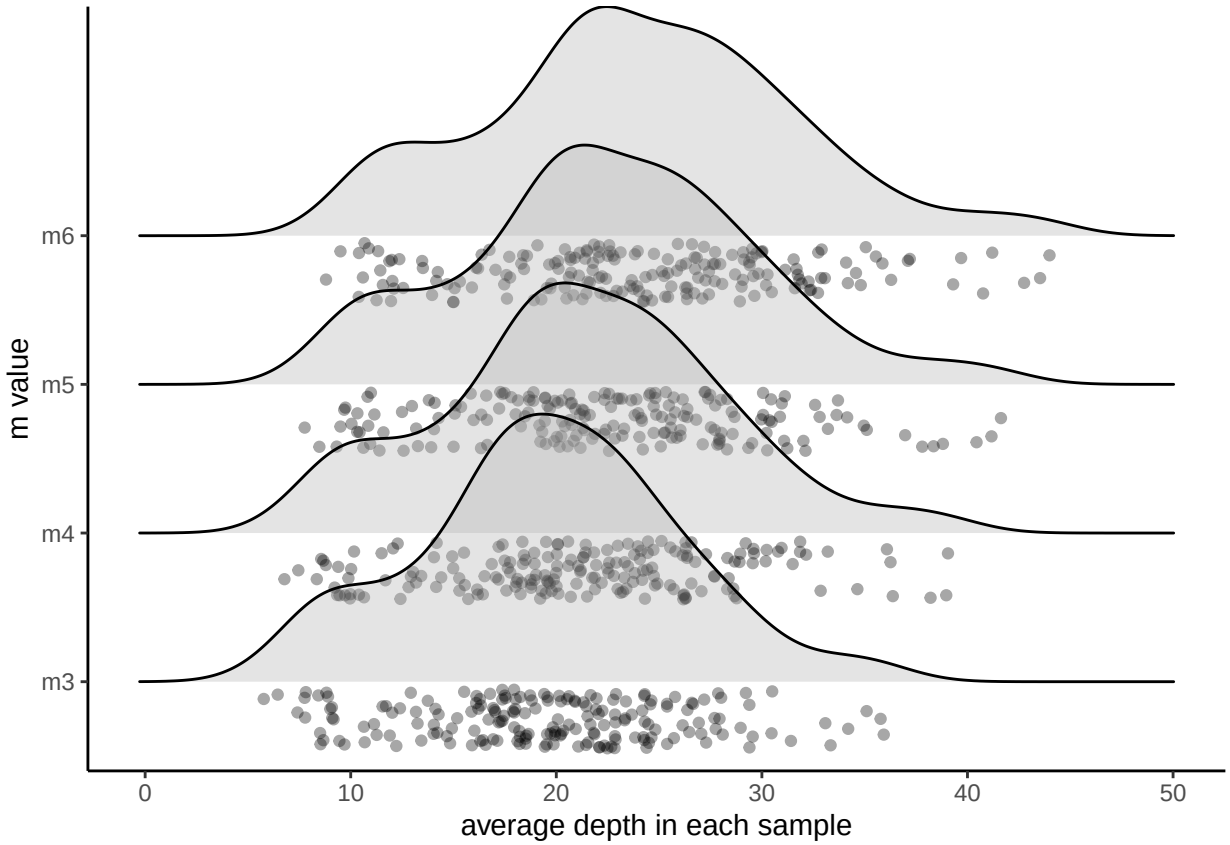
```
m.out=optimize_m(m3="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/m3/populations.snps.vcf",
                 m4="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/m4/populations.snps.vcf",
                 m5="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/m5/populations.snps.vcf",
                 m6="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/m6/populations.snps.vcf")
#How is depth affected by the m parameter?
vis_depth(output=m.out)
```

```
## [1] "Visualize how different values of m affect average depth in each sample"
```

```
## Warning in ggridges::geom_density_ridges(jittered_points = TRUE, position =
## "raincloud", : Ignoring unknown parameters: ‘size‘
```

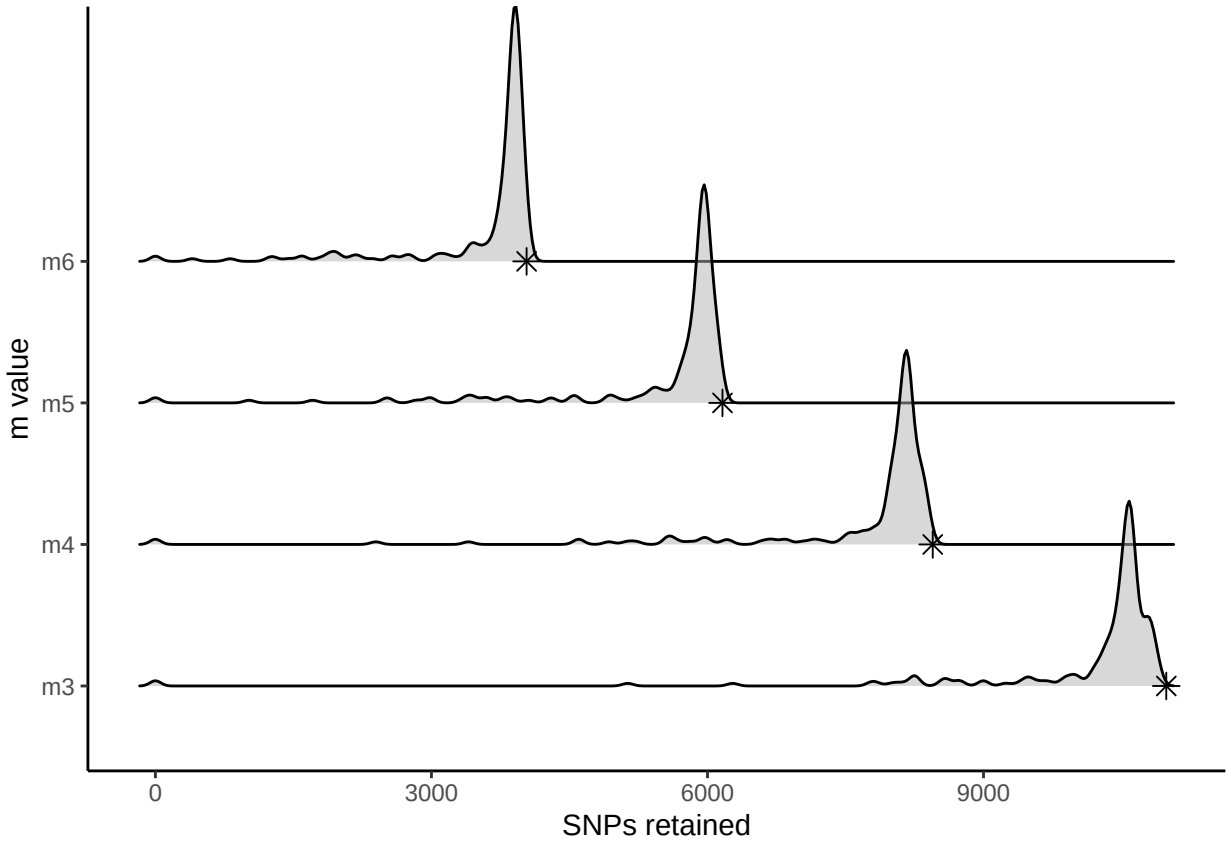
```
## Picking joint bandwidth of 2.01
```

```
## Warning: Removed 8 rows containing non-finite outside the scale range
## (‘stat_density_ridges()‘).
```



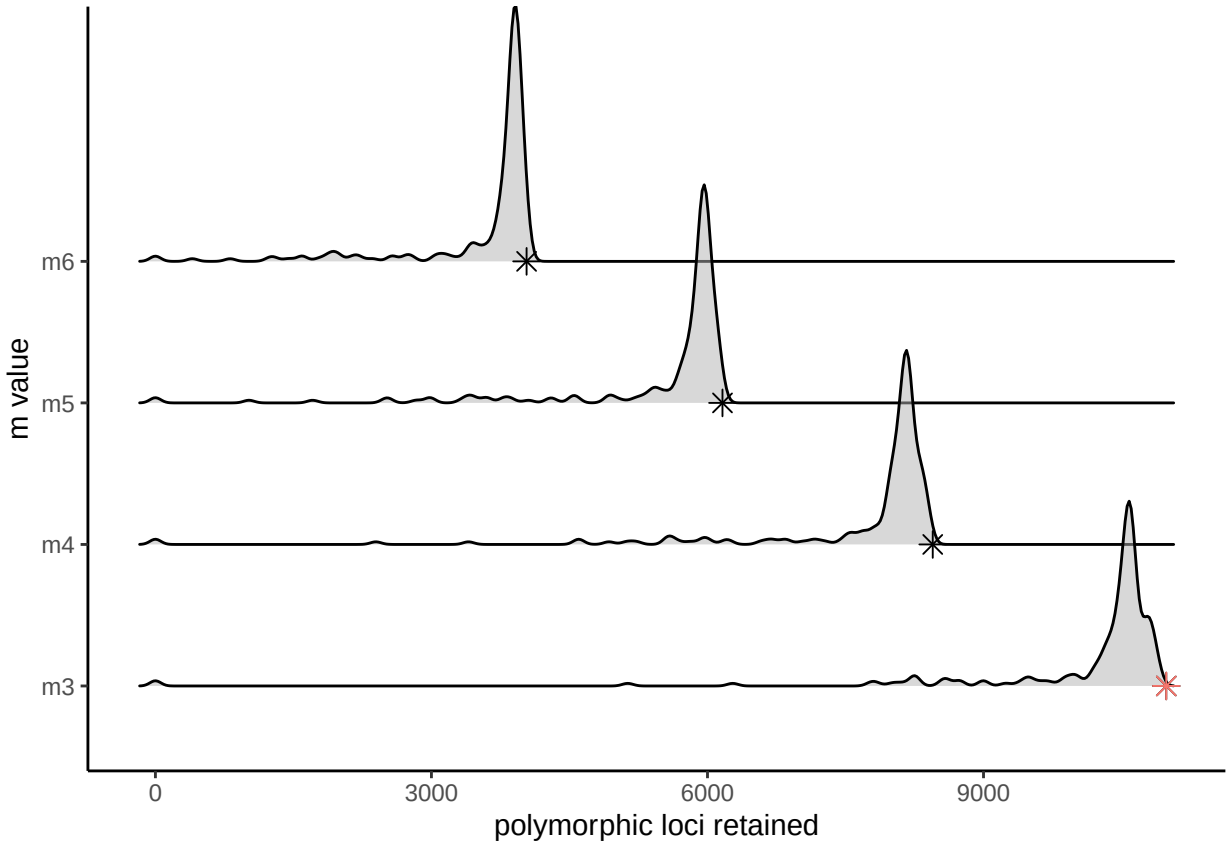

```
#How is SNP calling affected by the m parameter?
vis_snps(output = m.out, stacks_param = "m")
```

```
## Visualize how different values of m affect number of SNPs retained.
## Density plot shows the distribution of the number of SNPs retained in each sample,
## while the asterisk denotes the total number of SNPs retained at an 80% completeness cutoff.

## Picking joint bandwidth of 57.2
```




```
#How is # of loci affected by the m parameter?
vis_loci(output = m.out, stacks_param = "m")
```

```
## Visualize how different values of m affect number of polymorphic loci retained.
## Density plot shows the distribution of the number of loci retained in each sample,
## while the asterisk denotes the total number of loci retained at an 80% completeness cutoff. The optir

## Picking joint bandwidth of 57.2
```



The first figure shows how average depth changes with different values of m. Remember, m increases the number of identical sequences needed to form a stack, so by increasing it, we are increasing the minimum depth of each stack. Therefore, we expect to see average depth increase at higher values of m.

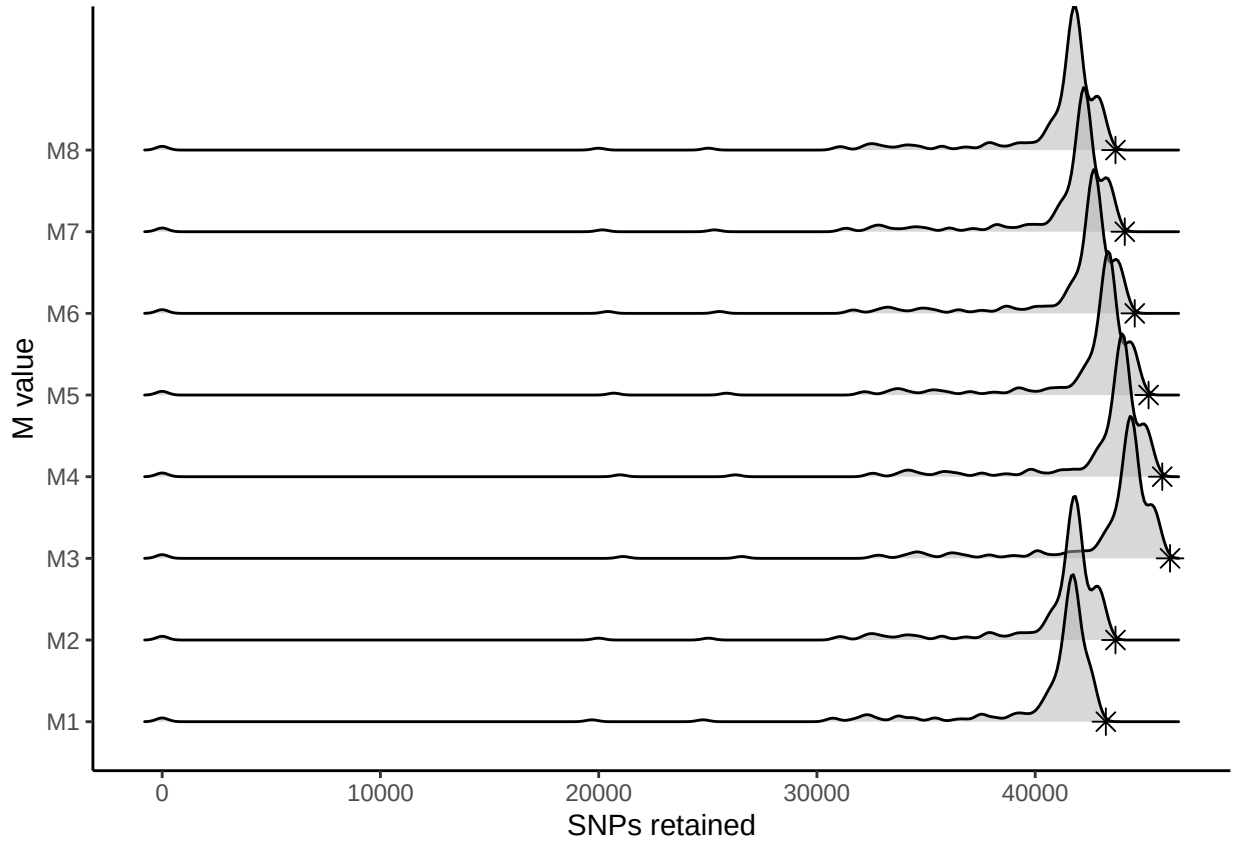
The second and third figures shows how many SNPs and polymorphic loci are retained at each value of m. Generally, fewer stacks formed by a higher minimum depth threshold will mean fewer SNPs and polymorphic loci. We want to maximize the number fo polymorphic loci available to us, and the optimal value for m is designated by the red star in figure 3.

3. Run the stacks pipeline varying ustacks parameter M = 0-8.
Repeat the process from step one, but this time the ustacks m parameter will be set to your optimized value and the M parameter will range from 0 to 8. Keep all outputs within the corresponding M0-M8 directories.

4. Visualize parameter M optimization
Run the following code to visualize parameter M:

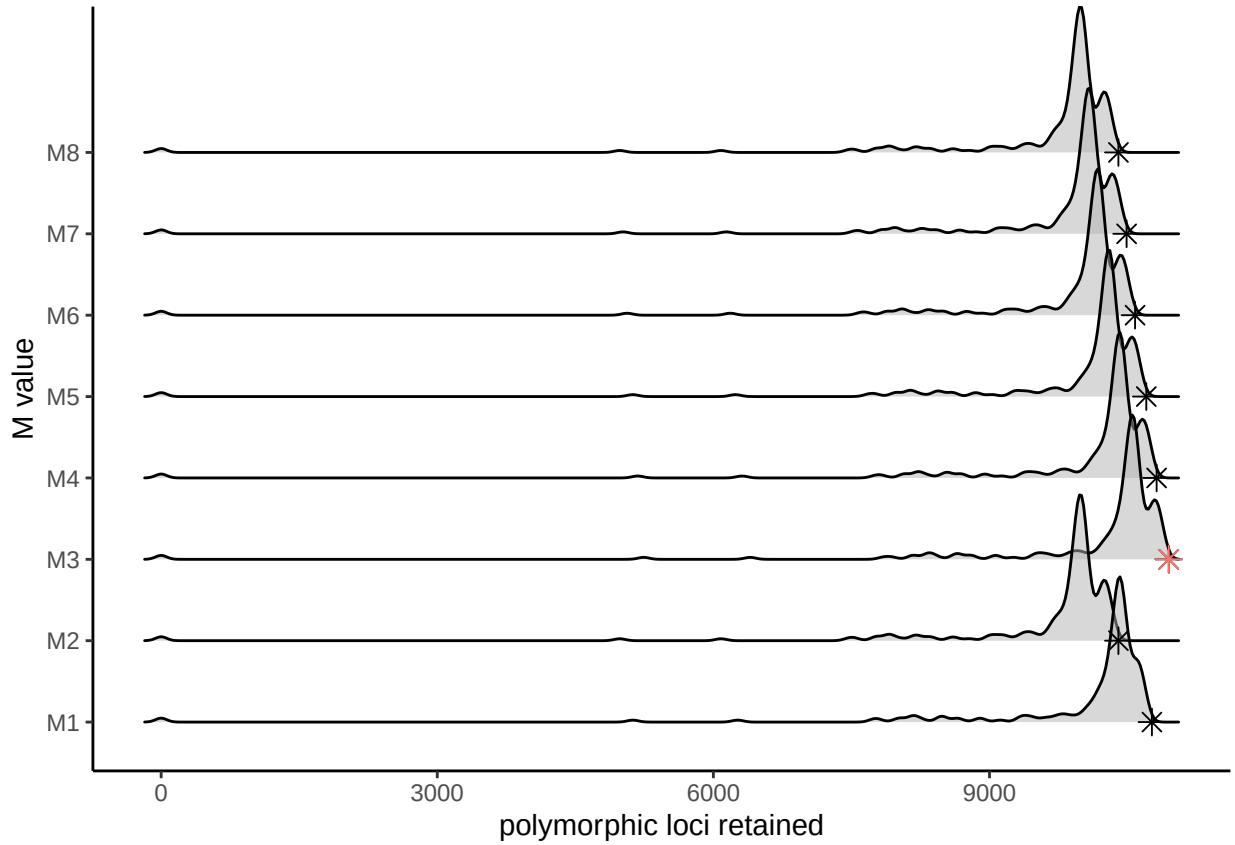
```
M.out<-optimize_bigM(M1="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/M.1/populations.snps.vcf",
                     M2="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/M.2/populations.snps.vcf",
                     M3="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/M.3/populations.snps.vcf",
                     M4="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/M.4/populations.snps.vcf",
                     M5="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/M.5/populations.snps.vcf",
                     M6="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/M.6/populations.snps.vcf",
                     M7="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/M.7/populations.snps.vcf",
                     M8="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/M.8/populations.snps.vcf")

#How does M affect the number of SNPs?
vis_snps(output = M.out, stacks_param = "M")
```

```
## Visualize how different values of M affect number of SNPs retained.
## Density plot shows the distribution of the number of SNPs retained in each sample,
## while the asterisk denotes the total number of SNPs retained at an 80% completeness cutoff.

## Picking joint bandwidth of 268
```




```
#How does M affect the number of polymorphic loci?
vis_loci(output = M.out, stacks_param = "M")
```

```
## Visualize how different values of M affect number of polymorphic loci retained.
## Density plot shows the distribution of the number of loci retained in each sample,
## while the asterisk denotes the total number of loci retained at an 80% completeness cutoff. The optim

## Picking joint bandwidth of 59.9
```



These two figures once again show the number of SNPs and polymorphic loci retained for different values of M. The optimal value of M that maximizes polymorphic loci is once again designated by a red star in the second figure.

5. Run the stacks pipeline varying cstacks parameter n = M-1 to M+1.
This time, the ustacks parameters will remain constant, which means we can use the ustacks output from the previous run with optimized m and M values. With these files, run the remainder of the stacks pipeline while varying the cstacks parameter n from M-1 to M+1 and outputting in the corresponding directories.

6. Visualize parameter n optimization

```
n.out = optimize_n(nequalsMminus1="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/nminus/populatio
           nequalsM="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/n/populations.snps.vcf",
           nequalsMplus1="~/sosf_ghri/schh_3rad/stacks/parameter_optimization/nplus/populations.snps.vc

vis_snps(output = n.out, stacks_param = "n")
```

7. Re-run populations with your final optimized data Now that we have the optimal values for M, m, and n, re-run populations on your optimized data with the –write-single-snp option added, and output into your projectname/stacks/optimized directory. We only keep one SNP per locus in the final dataset because SNPs within a locus are likely highly correlated and may cause problems with linkage disequilibrium.

Section 7: Read in optimal vcf file and save as vcfR object

```r
schh_vcf_0.1 = read.vcfR("~/sosf_ghri/schh_3rad/stacks/optimized/populations.snps.vcf")
```

```
## Scanning file to determine attributes.
## File attributes:
##   meta lines: 14
##   header_line: 15
##   variant count: 10951
##   column count: 201
## Meta line 14 read in.
## All meta lines processed.
## gt matrix initialized.
## Character matrix gt created.
##   Character matrix gt rows: 10951
##   Character matrix gt cols: 201
##   skip: 0
##   nrows: 10951
##   row_num: 0
## Processed variant 1000Processed variant 2000Processed variant 3000Processed variant 4000Processed va
## All variants processed
```

```r
save(schh_vcf_0.1, file = "~/sosf_ghri/schh_3rad/analysis/schh_vcf_0.1.Rdata")
```