Populations Genomics

General context

The analysis workflow—including quality control (QC) of raw sequencing data, read trimming, mapping to a reference genome, QC of mapping results, and PCR deduplication—is broadly applicable across various sequencing approaches. In this example, we use paired-end Illumina ddRAD-Seq data (150 bp reads), generated from individual samples sequenced at moderate coverage. While downstream analyses beyond SNP calling may vary depending on the experimental design and research objectives, the initial processing steps are consistent and form the foundation for high-quality data analysis.

Although the specific example provided here focuses on a ddRAD-Seq project, the same initial steps can be applied to other types of sequencing datasets. For instance, a similar strategy is commonly used for SNP variant calling in whole-genome sequencing (WGS) projects. However, it is important to note that no single pipeline is universally optimal for all sequencing data types, biological models, or research goals. For moderate- to high-coverage data, the tools and workflows presented here are generally effective. In contrast, different approaches may be more suitable for low-coverage samples. The examples offered here serve as a guide, and while default parameters (e.g., for read mappers like BWA-MEM) typically perform well for most species, adjusting parameters may be necessary based on the specific biological and technical context.

```
Notes for HPC before start
#On an HPC server (typically a Linux-based system), you can visualize directories and subdirectories as a tree structure using the tree command.

tree -d <dirname>

squeue -u <username>

# sacct -j <jobID> --format=JobID,Elapsed,TotalCPU,AllocCPUs,CPUTime,MaxRSS
# seff <jobID>
# sacct -u ksagonas -S <jobID>
# sinfo -p batch -N -o "%4N %8O %t" -n <compute node>
```

Step 1: Quality control of raw sequencing data

Assuming that raw sequencing data have already been downloaded or are otherwise available, the first step in any analysis pipeline is to assess the quality of the data using tools such as **FastQC** and **MultiQC**. FastQC generates detailed reports on a range of quality metrics, including per-base quality scores, GC content, adapter contamination, and sequence duplication levels. These reports help identify potential problems that could compromise downstream analyses.

After running FastQC on each individual sample, we recommend using MultiQC to aggregate the results into a single, comprehensive summary. MultiQC enables rapid identification of patterns and shared quality issues across all samples, allowing you to make informed decisions about

whether additional preprocessing—such as trimming or filtering—is necessary. Although trimming is often considered optional depending on data quality, we recommend performing read trimming in all cases to ensure consistency and improve downstream performance.

Evaluating raw data quality is a critical first step in any genomic analysis workflow. It provides a quick, global overview of dataset integrity and helps ensure the reliability and accuracy of subsequent steps in the pipeline.

In our example, these analyses are performed **in parallel**, not sequentially, which allows for greater efficiency when handling multiple samples.

SLURM Batch Script: fastqc_array.sbatch

```
#!/bin/bash
#SBATCH --partition=batch
#SBATCH --time=15:00
#SBATCH --job-name=fastqc_parallel
#SBATCH --output=fastqc_parallel_%j.out
#SBATCH --error=fastqc_parallel_%j.err
#SBATCH --ntasks=4
#SBATCH --mem=4G

set -euo pipefail

# Load necessary modules
module load gcc/14.2.0 fastqc/0.12.1

# Define input and output directories
OUT_DIR=/mnt/sagonas_a/PopGen_HPC/02FastQC_reports/00RawReads
IN_DIR=/mnt/sagonas_a/PopGen_HPC/00RawReads

# Create output directory if it doesn't exist
mkdir -p "$OUT_DIR"

# Use SLURM_NTASKS if defined, fallback to 1
THREADS=${SLURM_NTASKS:-1}

# Run FastQC in parallel using GNU parallel
find "$IN_DIR" -name "*.fastq.gz" | parallel -j "$THREADS" fastqc -t 1 -o "$OUT_DIR" {}
```

Step 2: Trimming Raw Sequence Reads

Trimming is a crucial preprocessing step in next-generation sequencing (NGS) data analysis. It involves removing low-quality bases, sequencing adapters, and other unwanted sequences (such as overly short reads after trimming) from raw data. This step helps ensure that only high-quality reads proceed to downstream analysis, thereby improving the accuracy of read mapping, variant calling, and other bioinformatics processes. Trimming addresses common issues such as low-quality read ends and residual adapter contamination, both of which can distort alignment and reduce confidence in analytical results.

Several tools are available for trimming, each with its strengths. **Trimmomatic** is a widely used and versatile tool that offers a range of trimming operations with support for paired-end data and multiple filtering criteria. However, **Cutadapt** has gained popularity due to its speed, simplicity, and precision in adapter removal. Cutadapt offers more flexible adapter matching options, better handling of variable-length adapters, and seamless integration into modern workflow managers and Python-based pipelines. In cases where accurate and customizable adapter trimming is critical—such as in RAD-seq or amplicon sequencing—Cutadapt is often preferred. It also supports trimming based on quality scores and can discard reads that fall below a specified length threshold after trimming, making it well-suited for high-throughput datasets with variable-quality reads.

In this workflow, we use **Cutadapt** for its efficiency and adaptability to the specific characteristics of our sequencing data. In the following step we will use **fastqc** again to assess the success of Cutadapt (script not given).

SLURM Batch Script: cutadapt_array.sbatch

```
#!/bin/bash
#SBATCH --partition=batch
#SBATCH --job-name=cutadapt
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --mem=8G
#SBATCH --time=01:00:00
#SBATCH --output=cutadapt_%A_%a.out
#SBATCH --error=cutadapt_%A_%a.err
#SBATCH --array=0-29%6  # Update to match number of samples - 1

set -euo pipefail

module load gcc/13.2.0-iqpfkya py-cutadapt/4.7-eavfyng

cd /mnt/sagonas_a/PopGen_HPC/00RawReads
ls *_R1.fastq.gz | sed 's/_R1.fastq.gz//' > /mnt/sagonas_a/PopGen_HPC/samples.txt

cd /mnt/sagonas_a/PopGen_HPC

SAMPLE=$(sed -n "$((SLURM_ARRAY_TASK_ID + 1))p" samples.txt)
IN_DIR=/mnt/sagonas_a/PopGen_HPC/00RawReads
OUT_DIR=/mnt/sagonas_a/PopGen_HPC/01TrimmedReads

#mkdir -p "$OUT_DIR"

cutadapt -q 20 --quality-base 33 -m 50 \
  -a AGATCGGAAGAGCACACGTCTGAACTCCAGTCA -A
AGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT \
  -o "$OUT_DIR/${SAMPLE}_R1_trimmed.fastq.gz" \
  -p "$OUT_DIR/${SAMPLE}_R2_trimmed.fastq.gz" \
  "$IN_DIR/${SAMPLE}_R1.fastq.gz" "$IN_DIR/${SAMPLE}_R2.fastq.gz"
```

Step 3: Mapping reads against a reference genome

Bowtie2 and BWA-MEM are both widely used tools for aligning sequencing reads to a reference genome, each with specific strengths. Generally, both perform comparably and can be used on the same types of sequencing data.

Bowtie2 is highly efficient and flexible, particularly for aligning short to medium-length reads. It performs well in handling mismatches and small indels, making it a preferred tool for datasets such as RNA-seq and metagenomics, where these challenges are common.

BWA-MEM, on the other hand, is designed for longer reads—like those from Illumina paired-end sequencing or even some third-generation platforms. It excels in accurate gapped alignments and is particularly robust in complex genomic regions. As such, it is often favored for whole-genome resequencing, structural variant detection, and other high-precision applications.

In this tutorial, we focus on **BWA-MEM2**, an improved version of BWA-MEM that offers faster alignment and lower memory usage. While Bowtie2 remains a strong alternative, BWA-MEM2 is especially well-suited for the paired-end Illumina ddRAD-Seq data used in this workflow.

Reference Indexing

Before alignment, most read mappers require the reference genome to be indexed—a preprocessing step that builds data structures for efficient access during mapping. It's important to note that different tools use different indexing formats, so index files are not interchangeable between software. Always check for errors in the logs when running new tools, especially when using pre-existing indexes.

With BWA-MEM2, indexing is straightforward but memory-intensive. To simplify this step, pre-built indexes are provided in the same directory as the reference genome. Below is an example SLURM batch script (reference_index.sbatch) used to generate the required index files:

SLURM Batch Script: reference_index.sbatch

```
#!/bin/bash
#SBATCH --partition=batch
#SBATCH --time=1:00:00
#SBATCH --job-name=indexing
#SBATCH --output=indexing.out
#SBATCH --error=indexing.err
#SBATCH --mem=8G

# Indexing the reference genome using samtools faidx and bwa index options. This step is neccessary for mapping the reads to the reference genome

module load gcc/13.2.0-iqpfkya samtools/1.19.2-wqjp7os bwa/0.7.17-remh23z

REFERENCE=/mnt/sagonas_a/PopGen_HPC/03ReferenceGenome/GCA_964106915.1_rPodGai1.hap1.1_genomic.fa

bwa index $REFERENCE
```

```
samtools faidx $REFERENCE

# Run Picard to generate the dict file for variant calling with GATK
module load gcc/8.2.0 r-rjava/0.9-11 picard/2.18.3

java -jar $PICARD_ROOT/bin/picard.jar CreateSequenceDictionary R=$REFERENCE
O=$REFERENCE.dict

# The output will look like this
#reference.fasta
#reference.fasta.fai     ← samtools index
#reference.dict          ← GATK/Picard
#reference.fasta.bwt     ← bwa index
#reference.fasta.pac     ← bwa index
#reference.fasta.ann     ← bwa index
#reference.fasta.amb     ← bwa index
#reference.fasta.sa      ← bwa index
```

Read Alignment

After indexing, reads can be aligned to the reference genome using the bwa mem algorithm. Below is an SLURM array job script to process multiple samples in parallel.

SLURM Batch Script: bwa_index.sbatch

```
#!/bin/bash
#SBATCH --partition=batch
#SBATCH --job-name=bwa_align
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --mem=8G
#SBATCH --time=01:00:00
#SBATCH --output=bwa_%A_%a.out
#SBATCH --error=bwa_%A_%a.err
#SBATCH --array=0-29%6  # Update to match number of samples - 1

set -euo pipefail

module load gcc/13.2.0-iqpfkya samtools/1.19.2-wqjp7os bwa/0.7.17-remh23z

cd /mnt/sagonas_a/PopGen_HPC

SAMPLE=$(sed -n "$((SLURM_ARRAY_TASK_ID + 1))p" samples.txt)
REFERENCE=/mnt/sagonas_a/PopGen_HPC/03ReferenceGenome/GCA_964106915.1_rPodGai
1.hap1.1_genomic.fa
IN_DIR=/mnt/sagonas_a/PopGen_HPC/01TrimmedReads
OUT_DIR=/mnt/sagonas_a/PopGen_HPC/04AlignedReads
STATS=$OUT_DIR/Alignment_statistics

mkdir -p $STATS
```

```
bwa mem -t $SLURM_CPUS_PER_TASK $REFERENCE \
 $IN_DIR/${SAMPLE}_R1_trimmed.fastq.gz \
 $IN_DIR/${SAMPLE}_R2_trimmed.fastq.gz \
 | samtools view -Sb - \
 | samtools sort -@ $SLURM_CPUS_PER_TASK -o $OUT_DIR/${SAMPLE}.sorted.bam

samtools index $OUT_DIR/${SAMPLE}.sorted.bam
samtools flagstat $OUT_DIR/${SAMPLE}.sorted.bam > $STATS/${SAMPLE}.flagstat.txt
# samtools stats $OUT_DIR/${SAMPLE}.sorted.bam > $STATS/${SAMPLE}.stats.txt

# Note: For large sequencing datasets, deleting unnecessary files (e.g. unsorted BAM files) once they are no longer needed helps free up valuable storage space on your computing cluster

# QC based on the results of mapping
# Performing quality control after read mapping is a critical step to assess the success of the alignment process and ensure reliable downstream analyses. Tools like Samtools flagstat, Samtools idxstats and Bamtools stats, provide valuable insights into the quality and distribution of mapped reads. All tools summarize key alignment statistics, such as the total number of reads, the proportion mapped to the reference genome, and the fraction of properly paired reads in paired-end reads sequencing data. These metrics help evaluate alignment efficiency and identify potential issues, such as low-quality mappings or unexpected levels of unmapped reads.

# Alternatively we can use bamtools stats similar to samtools flagstat
# bamtools stats -in $OUT_DIR/${SAMPLE}.sorted.bam > $STATS/${SAMPLE}.aligned.stats.txt

# Samtools idxstats is another valuable tool to have an overview of the mapping. It provides per-chromosome statistics about mapped and unmapped reads relative to the length of each scaffold in the reference assembly. The output consists of four columns: scaffold name, scaffold length, the number of mapped reads, and the number of unmapped reads. However, it's important to note that the term "unmapped reads" in this context can be somewhat misleading - it refers to reads assigned to a chromosome but with low-confidence alignments, such as low-quality mappings or secondary alignments. By analyzing the idxstats output, researchers can assess genome coverage by dividing the number of reads by the scaffold length and identify potential issues, such as uneven coverage, due to over-represented contigs or contamination. This makes idxstats an essential step in ensuring the quality and reliability of mapped sequencing data.

samtools idxstats $OUT_DIR/${SAMPLE}.sorted.bam > $STATS/${SAMPLE}.idxstats.txt
```

Post-Mapping Quality Control

Evaluating the success of read mapping is a crucial step before variant calling or other downstream analyses. Several tools provide alignment statistics to help assess data quality:

- **samtools flagstat**: Summarizes total reads, mapped reads, properly paired reads, and other essential metrics.
- **samtools stats**: Offers more detailed statistics, including insert size, mapping quality distributions, and error rates.

- **bamtools stats**: An alternative that provides similar summary statistics in a different format.
- **samtools idxstats**: Reports per-chromosome statistics, including mapped/unmapped reads and scaffold lengths. Note: "unmapped" here may include low-quality or secondary alignments.

These statistics help identify problems like low alignment rates, adapter contamination, overrepresented scaffolds, or potential contamination. For example, genome coverage can be estimated by dividing the number of mapped reads by scaffold length, aiding in the detection of uneven coverage or technical biases.

Step 4: Removing PCR duplicates

Marking PCR duplicates is an essential part of sequencing data processing, particularly in whole-genome sequencing (WGS) workflows. PCR amplification during library preparation can generate duplicate reads originating from the same DNA fragment, which may bias downstream analyses, such as variant calling. Marking duplicates—rather than removing them—allows tools to flag such reads (typically with the 0x400 SAM flag), preserving all information for flexible downstream interpretation.

One of the most commonly used tools for this task is **Picard's MarkDuplicates**, which identifies duplicates based on read start/end positions and outputs both a deduplicated BAM file and a metrics file summarizing duplication levels. While **Samtools markdup** is a faster and more lightweight alternative, especially useful on HPC clusters for large datasets, Picard is often preferred for final pipelines due to its detailed output and community trust. However, keep in mind that Picard uses java that might further delay the analyses.

However, **the importance of duplicate marking varies by experimental design.** For example, in **RAD-Seq** data, especially with single-digest protocols, reads often share identical start positions not because they are PCR duplicates, but due to **the nature of the protocol**. Similarly, **single-end** sequencing reads can have overlapping coordinates despite originating from unique fragments. In these cases, overly aggressive duplicate marking can remove valid data, so decisions should be tailored to the specific project.

> **In this workflow**, we use both **Samtools** and **Picard** for duplicate marking. Samtools is used initially for speed, followed by Picard for comprehensive metrics.

Important Note – Read Groups for GATK Compatibility

For downstream tools like **GATK HaplotypeCaller** and **BaseRecalibrator**, it's critical that all BAM files contain **read group (RG) tags**. These tags identify metadata about the origin of each read and must include the following fields:

- ID (Read Group ID)
- SM (Sample name)
- LB (Library)
- PL (Platform, e.g., ILLUMINA)
- PU (Platform Unit)

Without proper RG tags in the BAM header and reads, **GATK will fail** or report errors. Therefore, after marking duplicates, we use **Picard AddOrReplaceReadGroups** to ensure all BAM files are correctly annotated for compatibility with GATK and other downstream tools.

SLURM Batch Script: AddTags.sbatch

```
#!/bin/bash
#SBATCH --partition=batch
#SBATCH --job-name=markdup
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --mem=16G
#SBATCH --time=01:00:00
#SBATCH --output=markdup_%A_%a.out
#SBATCH --error=markdup_%A_%a.err
#SBATCH --array=0-14%6  # Update to match number of samples - 1

# Duplicates can bias downstream analyses, like variant calling, by inflating read counts artificially and causing false positives. Here we use samtools fixmate as a command that corrects mate pair information in a BAM file. It's specifically used after sorting the BAM file by read name (samtools sort -n) and before marking duplicates using samtools markdup. When dealing with paired-end reads, it's important that the metadata about each read and its mate (e.g. flags, template length, mate positions) is consistent and accurate. If this information is missing or incorrect, samtools markdup cannot reliably identify duplicates, and downstream tools may produce errors or misleading results.

set -euo pipefail

module load gcc/13.2.0-iqpfkya samtools/1.19.2-wqjp7os

cd /mnt/sagonas_a/PopGen_HPC

SAMPLE=$(sed -n "$((SLURM_ARRAY_TASK_ID + 1))p" samples.txt)
IN_DIR=/mnt/sagonas_a/PopGen_HPC/04AlignedReads
OUT_DIR=/mnt/sagonas_a/PopGen_HPC/04AlignedReads/Dedup/Samtools

mkdir -p "$OUT_DIR"

# Sort by read name (queryname) and write to a temp file
samtools sort -n -o $IN_DIR/${SAMPLE}.name_sorted.bam $IN_DIR/${SAMPLE}.sorted.bam

# Run fixmate on the name-sorted BAM, output another temp BAM
samtools fixmate -m $IN_DIR/${SAMPLE}.name_sorted.bam $IN_DIR/${SAMPLE}.fixmate.bam

# Sort fixmate output by coordinate, final BAM before markdup
samtools sort -o $IN_DIR/${SAMPLE}.coord_sorted.bam $IN_DIR/${SAMPLE}.fixmate.bam

# Mark duplicates on coordinate sorted BAM and output final dedup BAM
```

```
samtools markdup -r $IN_DIR/${SAMPLE}.coord_sorted.bam $OUT_DIR/${SAMPLE}.rmdup.bam

# Index final BAM
samtools index $OUT_DIR/${SAMPLE}.rmdup.bam

# Optional: remove intermediate files to save space
rm $IN_DIR/${SAMPLE}.name_sorted.bam $IN_DIR/${SAMPLE}.fixmate.bam
$IN_DIR/${SAMPLE}.coord_sorted.bam
```

Alternatively, we can run Picard to remove/flag Duplicates. For large-scale production and variant calling, Picard's MarkDuplicates is often preferred because of its comprehensive metrics and community trust. For quick or lightweight workflows, especially on HPC systems, samtools markdup is a fine alternative. Some pipelines use both: samtools for marking duplicates quickly during early steps, Picard for final, detailed duplicate marking before variant calling.

```
OUT_DIR=/mnt/sagonas_a/PopGen_HPC/04AlignedReads/Dedup/Picard
PICARD_STATS=$OUT_DIR/Stats

mkdir -p $PICARD_STATS

module load gcc/8.2.0 r-rjava/0.9-11 picard/2.18.3

java -Xmx16G -jar $PICARD_ROOT/bin/picard.jar MarkDuplicates \
I=$IN_DIR/${SAMPLE}.sorted.bam \
O=$OUT_DIR/${SAMPLE}.pcd.rmdup.bam \
M=$PICARD_STATS/${SAMPLE}.pcd.rmdup.metrics.txt \
REMOVE_DUPLICATES=false CREATE_INDEX=true
```

Each read in the BAM must have an associated read group, with the following fields set in the BAM header and applied to each read via the @RG tag. ReadGroupID, Library, Platform, Samplename, and Unit. Without RG tags, tools like HaplotypeCaller and BaseRecalibrator will fail

```
IN_DIR=/mnt/sagonas_a/PopGen_HPC/04AlignedReads/Dedup/Samtools
OUT_DIR=/mnt/sagonas_a/PopGen_HPC/04AlignedReads/Dedup/Samtools/RG

mkdir -p $OUT_DIR

java -Xmx16G -jar $PICARD_ROOT/bin/picard.jar AddOrReplaceReadGroups \
  I=$IN_DIR/${SAMPLE}.rmdup.bam \
  O=$OUT_DIR/${SAMPLE}.rg.rmdup.bam \
  RGID=$SAMPLE \
  RGLB=lib1 \
  RGPL=ILLUMINA \
  RGPU=unit1 \
  RGSM=$SAMPLE

# samtools view -H yourfile.bam | grep '^@RG'
# Need to index the file again
```

```
module load gcc/13.2.0-iqpfkya samtools/1.19.2-wqjp7os

samtools index $OUT_DIR/${SAMPLE}.rg.rmdup.bam
```

Step 5: SNP variant calling

Variant calling is the final critical step in the sequencing analysis workflow, where genetic variants such as SNPs (single nucleotide polymorphisms) and indels (insertions and deletions) are identified by comparing aligned sequencing reads to a reference genome. Several tools are available for variant calling, each offering different features, strengths, and compatibility depending on the type of data and analysis goals.

BCFtools is a lightweight and efficient tool that is often used for rapid variant calling and filtering directly from BAM files. It is well-suited for small- to medium-sized projects and offers straightforward integration into existing command-line workflows. However, while BCFtools is fast and easy to use, it lacks some of the more advanced features and quality controls provided by other tools.

VCFtools, on the other hand, is not a variant caller per se but a widely used tool for filtering, comparing, and analyzing VCF (Variant Call Format) files. It is typically used downstream of the variant calling process, rather than to generate the variants themselves. VCFtools is not updated anymore.

The **Genome Analysis Toolkit (GATK)** is a widely used software suite designed for variant discovery and genotyping from high-throughput sequencing data. The GATK Best Practices workflow provides a reliable and standardized approach to identify variants such as single nucleotide polymorphisms (SNPs) and insertions/deletions (indels) in both individual genomes and populations. The **initial** step in this workflow involves using the **HaplotypeCaller** tool, which processes each sample individually to produce GVCF files. These GVCF files contain detailed information on variant sites as well as non-variant regions, enabling a comprehensive summary of each genome. The advantage of this approach is that it allows for efficient joint genotyping later without the need to rerun variant calling for all samples every time a new sample is added. After generating per-sample GVCFs, the **next** key step is joint genotyping using the **GenotypeGVCFs** tool. This step combines the information from all samples to call genotypes simultaneously, improving accuracy by leveraging population-level data and ensuring consistent variant calls across the cohort.

For computational efficiency, especially with large datasets or whole genomes, it is common practice to run GenotypeGVCFs separately on each chromosome or genomic region. This approach parallelizes the workload, speeding up the analysis. However, this produces chromosome-specific VCF files, which need to be merged before further processing. Merging is essential to unify the variants across the whole genome for downstream analysis. This is typically done using tools such as bcftools, which can concatenate chromosome-specific VCFs into a single combined VCF file, taking care to maintain compression and indexing for efficient access. **Variant filtering** is a critical step that follows merging. It helps to distinguish true genetic variants from sequencing artifacts or errors. GATK offers two main strategies for filtering variants. The first is

Variant Quality Score Recalibration (**VQSR**), a sophisticated machine learning-based method that uses known variant resources to build a model distinguishing high-quality from low-quality variants. This method is highly effective for large cohorts and model organisms where reliable resource datasets exist. The second strategy is hard filtering, which applies fixed thresholds on various annotation metrics such as Quality by Depth (QD), Fisher Strand (FS), Mapping Quality (MQ), and others. Hard filtering is particularly useful when resource files for VQSR are unavailable, such as in non-model organisms or small datasets. It is important to note that filtering should always be applied on the combined VCF, rather than on individual chromosome files, to avoid inconsistencies and batch effects. Sometimes, researchers use **SelectVariants** to subset variants into categories like SNPs and indels before applying filtering. This can improve filtering precision since different variant types may require distinct thresholds. However, this step is optional and depends on the goals of the analysis. Compression and indexing are vital technical steps that ensure VCF files are accessible and efficiently processed by downstream tools. The merged VCF file should be compressed using bgzip and indexed using tabix before filtering or further analysis.

In workflows that leverage parallel processing or job arrays, it is important to recognize that while variant calling and joint genotyping can be run in parallel for each chromosome, merging and filtering steps generally require the final combined VCF file and thus cannot be parallelized in the same way. Attempting to filter variants on per-chromosome VCFs independently can lead to errors and inconsistencies. Common issues in this workflow often arise from improper handling of file formats or merging steps. For example, if the merged VCF is not properly compressed or indexed, GATK may fail to read the file, generating errors related to missing codecs or invalid file format. Ensuring that bcftools or other merging tools are used correctly with compression and indexing resolves these issues. Another common pitfall is providing lists of files incorrectly, such as passing newline-separated filenames as a single argument instead of space-separated entries or an array.

In summary, the recommended workflow begins with per-sample variant calling producing GVCFs, followed by joint genotyping of all samples per chromosome. After obtaining chromosome-specific VCFs, these are merged into a single, compressed, and indexed VCF file. Variant filtration, whether through VQSR or hard filtering, is then performed on this combined VCF. Indexing the filtered VCF concludes the core analysis pipeline, preparing it for downstream applications such as annotation or association studies. Resource files used by VQSR, such as known variant datasets, should be chosen carefully and are usually available for well-studied organisms. When these resources are not available, hard filtering remains a valid and effective alternative. Thresholds used in hard filtering should be adapted based on the quality of the sequencing data and experimental design.

An optional, but recommended, enhancement to this pipeline is **Base Quality Score Recalibration (BQSR)**. This step, partially included in your script as commented code, corrects systematic errors made by the sequencer in base quality scores, which directly affect the confidence of variant calls. BQSR uses a known set of variants (obtained from hard-filtered, high-confidence SNPs) as a truth set, recalibrates the base scores accordingly, and improves downstream variant calling. However, BQSR is most beneficial when coverage is moderate to high; in low-coverage studies, it may introduce more noise than it resolves. In your course setting, this step is

left optional and may be skipped due to practical constraints, like limited read depth or lack of a known high-quality variant set.

Overall, this workflow balances accuracy, efficiency, and flexibility, accommodating different project sizes, organism types, and available resources. Proper understanding and implementation of each step ensure high-quality variant calls suitable for robust genetic analyses.

Note: Hard Filtering vs VariantFiltration in GATK: When working with variant call data in GATK, it's important to ensure the variants you analyze are of high quality. Two related concepts often come up in this context: hard filtering and the VariantFiltration tool. Hard filtering refers to the practice of applying fixed, user-defined cutoff thresholds to specific variant annotations. These annotations might include metrics such as Quality by Depth (QD), Fisher Strand bias (FS), Mapping Quality (MQ), and others. For example, you might decide to exclude variants with QD less than 2.0 or FS greater than 60. These cutoffs are predetermined, simple, and non-adaptive rules used to filter out variants that are likely false positives based on empirical experience or literature. The VariantFiltration tool in GATK is the actual software utility that implements this hard filtering strategy. It takes a VCF file as input and applies the user-specified filter expressions, tagging variants that fail those thresholds with filter labels in the output VCF. This means VariantFiltration is the means to perform hard filtering within the GATK framework. It does not modify or remove variants but marks those that don't pass filters, allowing downstream tools or analyses to exclude or handle them differently. It's important to distinguish this from another GATK filtering method called Variant Quality Score Recalibration (VQSR). VQSR is a more advanced, machine-learning based filtering approach that models the quality metrics of known, trusted variant sites and assigns a probabilistic score to new variants. Unlike hard filtering, VQSR adapts its thresholds based on data distributions and is generally considered more accurate but requires large datasets and reliable known variant resources. In practice, hard filtering with VariantFiltration is often used when VQSR is not feasible—such as in smaller datasets, non-model organisms without established variant databases, or as an initial filtering step. The flexibility of VariantFiltration allows users to tailor filter thresholds according to their data quality and experimental design. In summary, hard filtering is a concept describing filtering variants by fixed thresholds, and VariantFiltration is the GATK tool that performs this filtering. When using GATK, applying hard filters means configuring and running VariantFiltration with appropriate cutoffs. This approach provides a straightforward and effective way to clean variant calls, especially when VQSR is not an option.

SLURM Batch Script: GatkHaplo_array.sbatch

```
#!/bin/bash
#SBATCH --partition=batch
#SBATCH --job-name=GATK
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --mem=16G
#SBATCH --time=01:00:00
#SBATCH --output=GATK_%A_%a.out
```

```
#SBATCH --error=GATK_%A_%a.err
#SBATCH --array=0-14%6

module load gcc/14.2.0 gatk/4.5.0.0

cd /mnt/sagonas_a/PopGen_HPC

SAMPLE=$(sed -n "$((SLURM_ARRAY_TASK_ID + 1))p" samples.txt | tr -d '[:space:]')
REFERENCE=/mnt/sagonas_a/PopGen_HPC/03ReferenceGenome/GCA_964106915.1_rPodGai
1.hap1.1_genomic.fa
IN_DIR=/mnt/sagonas_a/PopGen_HPC/04AlignedReads/Dedup/Samtools/RG
OUT_DIR_BASE=/mnt/sagonas_a/PopGen_HPC/05VariantCalling
OUT_DIRS=("GVCF" "Filtered")

# Create output directories
for d in "${OUT_DIRS[@]}"; do mkdir -p "${OUT_DIR_BASE}/${d}"; done

# Step 1: Run HaplotypeCaller on each sample in GVCF mode

 gatk HaplotypeCaller \
  -R $REFERENCE \
  -I $IN_DIR/${SAMPLE}.rg.rmdup.bam \
  -O $OUT_DIR_BASE/GVCF/${SAMPLE}.g.vcf.gz \
  -ERC GVCF

# In the following steps we can apply BSQR (Base Quality Score Recalibration). BQSR is a machine
learning-based process in GATK that detects systematic errors made by the sequencer in
estimating the quality score of each base call. Then adjusts those scores, improving downstream
variant calling accuracy. In other words, base quality scores directly influence Which bases are
trusted during variant calling, the confidence of variant calls (especially SNPs and indels), and
minimize biases that could be lead to false positives (bad variants called as real) and false
negatives (real variants missed due to underestimated confidence) calls. BSQR can be avoided
when coverage is very low and recalibration would introduce noise or when we re using tools non
sensitive to recalibrated scores

# # 2. Apply hard filters to isolate high-confidence SNPs
# gatk VariantFiltration \
#   -R $REFERENCE \
#   -V $OUT_DIR_BASE/GVCF/${SAMPLE}.g.vcf.gz \
#   -O $OUT_DIR_BASE/GVCF/${SAMPLE}.g.filtered.vcf.gz \
#   --filter-expression "QD < 2.0 || FS > 60.0 || MQ < 40.0" \
#   --filter-name "FAIL"

# # 3. Then retain only PASS variants:
# module load gcc/13.2.0-iqpfkya bcftools/1.19-odjvyvt
# bcftools view -f PASS -v snps $OUT_DIR_BASE/GVCF/${SAMPLE}.g.filtered.vcf.gz -Oz -o
$OUT_DIR_BASE/GVCF/${SAMPLE}.g.known_sites.vcf.gz
# tabix -p vcf $OUT_DIR_BASE/GVCF/${SAMPLE}.g.known_sites.vcf.gz
```

```
# # 4. Use this VCF as your --known-sites input for BQSR
# # Applying BQSR to re-call variants with recalibrated BAM, improves quality scores and results in better variant calls

# BQSR_DIR=/mnt/sagonas_a/PopGen_HPC/06BQSR
#
KNOWN_SITES=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/GVCF/${SAMPLE}.g.known_sites.vcf.gz

# # Create output directory
# mkdir -p $BQSR_DIR

# # Run BaseRecalibrator
# gatk BaseRecalibrator \
#   -R $REFERENCE \
#   -I ${IN_DIR}/${SAMPLE}.rmdup.bam \
#   --known-sites $KNOWN_SITES \
#   -O ${BQSR_DIR}/${SAMPLE}.recal_data.table

# # Apply the recalibration
# gatk ApplyBQSR \
#   -R $REFERENCE \
#   -I ${IN_DIR}/${SAMPLE}.rmdup.bam \
#   --bqsr-recal-file ${BQSR_DIR}/${SAMPLE}.recal_data.table \
#   -O ${BQSR_DIR}/${SAMPLE}.bqsr.bam

# # Analyze covariates
# gatk AnalyzeCovariates \
#   -before ${BQSR_DIR}/${SAMPLE}.recal_data.table \
#   -after ${BQSR_DIR}/${SAMPLE}.recal_data.table \
#   -plots ${BQSR_DIR}/${SAMPLE}.bqsr_plots.pdf

# In the next step we should re-run HaplotypeCaller
```

SLURM Batch Script: GenomicsDBImport _array.sbatch

Step 2: GenomicsDBImport to combine all GVCFs. GenomicsDBImport is used in GATK's joint genotyping process. After calling variants per sample in GVCF mode (via HaplotypeCaller -ERC GVCF), you need to combine all sample GVCFs to jointly call genotypes across the cohort. This is essential for ensuring consistent variant calls across all samples, improving sensitivity for low-frequency variants and avoiding per-sample biases

```
#!/bin/bash
#SBATCH --partition=batch
#SBATCH --job-name=GenomicsDB
#SBATCH --array=0-21%6      # Adjust this to match number of chromosomes
#SBATCH --cpus-per-task=4
#SBATCH --mem=32G
#SBATCH --time=04:00:00
```

```
#SBATCH --output=logs/genomicsdb_%A_%a.out
#SBATCH --error=logs/genomicsdb_%A_%a.err

module load gcc/14.2.0 gatk/4.5.0.0

cd /mnt/sagonas_a/PopGen_HPC

REFERENCE=/mnt/sagonas_a/PopGen_HPC/03ReferenceGenome/GCA_964106915.1_rPodGai
1.hap1.1_genomic.fa
GVCF_DIR=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/GVCF
GENDB_DIR=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/GenomicsDB
SAMPLES_FILE="samples.txt"

mkdir -p $GENDB_DIR

# First we will create a list of samples that will be imported
SAMPLES_MAP=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/sample_gvcf_map.txt
if [ ! -f "$SAMPLES_MAP" ]; then
  while read -r SAMPLE; do
    SAMPLE_CLEAN=$(echo "$SAMPLE" | tr -d '[:space:]')
    echo -e "${SAMPLE_CLEAN}\t${GVCF_DIR}/${SAMPLE_CLEAN}.g.vcf.gz"
  done < "$SAMPLES_FILE" > "$SAMPLES_MAP"
fi

# Second we will create a chromosome list from the reference genome
CHROM_LIST=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/chroms.txt
if [ ! -f "$CHROM_LIST" ]; then
  cut -f1 ${REFERENCE}.fai > "$CHROM_LIST"
fi

# Get chromosome for this array task
CHR=$(sed -n "$((SLURM_ARRAY_TASK_ID + 1))p" "$CHROM_LIST")
OUT_DIR=${GENDB_DIR}/${CHR}

# Delete existing workspace if it exists
if [ -d "$OUT_DIR" ]; then
  echo "Deleting existing workspace: $OUT_DIR"
  rm -rf "$OUT_DIR"
fi

gatk GenomicsDBImport \
  --genomicsdb-workspace-path "$OUT_DIR" \
  --sample-name-map "$SAMPLES_MAP" \
  --reader-threads 4 \
  -L "$CHR"
```

SLURM Batch Script: GenotypeGVCFs_array.sbatch

```
#!/bin/bash
```

```
#SBATCH --partition=batch
#SBATCH --job-name=GenotypeGVCFs
#SBATCH --cpus-per-task=4
#SBATCH --mem=8G
#SBATCH --time=04:00:00
#SBATCH --output=logs/genotypegvcfs_%A_%a.out
#SBATCH --error=logs/genotypegvcfs_%A_%a.err
#SBATCH --array=0-21%6  # Update to match number of samples - 1

# Alternatively for full automation, we can remove the last line (--array) entirely and submit the bach like this
# N=$(wc -l < /mnt/sagonas_a/PopGen_HPC/05VariantCalling/chroms.txt)
# sbatch --array=0-$(($N - 1)) GenotypeGVCFs_array.sbatch

module load gcc/14.2.0 gatk/4.5.0.0

cd /mnt/sagonas_a/PopGen_HPC

REFERENCE=/mnt/sagonas_a/PopGen_HPC/03ReferenceGenome/GCA_964106915.1_rPodGai1.hap1.1_genomic.fa
GENDB_DIR=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/GenomicsDB
OUT_DIR=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/JointGenotyped
CHROM_LIST=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/chroms.txt

mkdir -p "$OUT_DIR"

# Get chromosome for this array task
CHR=$(sed -n "$((SLURM_ARRAY_TASK_ID + 1))p" "$CHROM_LIST")

gatk GenotypeGVCFs \
  -R "$REFERENCE" \
  -V gendb://$GENDB_DIR/$CHR \
  -O $OUT_DIR/${CHR}.vcf.gz

# Explore the vcf files
# module load gcc/13.2.0-iqpfkya bcftools/1.19-odjvyvt
# bcftools view -h <${CHR}.vcf.gz> #header
# bcftools view --no-header <${CHR}.vcf.gz> | less -S
```

SLURM Batch Script: Hardfiltering_array.sbatch

```
#!/bin/bash
#SBATCH --partition=batch
#SBATCH --job-name=HardFilter
#SBATCH --cpus-per-task=2
#SBATCH --mem=8G
#SBATCH --time=02:00:00
#SBATCH --output=logs/hardfilter_%A_%a.out
#SBATCH --error=logs/hardfilter_%A_%a.err
```

```
#SBATCH --array=0-21%6

module load gcc/14.2.0 gatk/4.5.0.0

cd /mnt/sagonas_a/PopGen_HPC

REFERENCE=/mnt/sagonas_a/PopGen_HPC/03ReferenceGenome/GCA_964106915.1_rPodGai
1.hap1.1_genomic.fa
VCF_DIR=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/JointGenotyped
OUT_DIR=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/HardFiltered
CHROM_LIST=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/chroms.txt

mkdir -p "$OUT_DIR"

# Get chromosome name for this task
CHR=$(sed -n "$((SLURM_ARRAY_TASK_ID + 1))p" "$CHROM_LIST")

INPUT_VCF=${VCF_DIR}/${CHR}.vcf.gz
SNPS_VCF=${OUT_DIR}/${CHR}_snps.vcf.gz
INDELS_VCF=${OUT_DIR}/${CHR}_indels.vcf.gz

# 1. Select SNPs
gatk SelectVariants \
 -R "$REFERENCE" \
 -V "$INPUT_VCF" \
 --select-type-to-include SNP \
 -O "$SNPS_VCF"

# 2. Filter SNPs
gatk VariantFiltration \
 -R "$REFERENCE" \
 -V "$SNPS_VCF" \
 -O "${OUT_DIR}/${CHR}_snps_filtered.vcf.gz" \
 --filter-name "QD_lt2" --filter-expression "QD < 2.0" \
 --filter-name "FS_gt60" --filter-expression "FS > 60.0" \
 --filter-name "MQ_lt40" --filter-expression "MQ < 40.0" \
 --filter-name "MQRankSum_lt-12.5" --filter-expression "MQRankSum < -12.5" \
 --filter-name "ReadPosRankSum_lt-8" --filter-expression "ReadPosRankSum < -8.0"

# 3. Select INDELs
gatk SelectVariants \
 -R "$REFERENCE" \
 -V "$INPUT_VCF" \
 --select-type-to-include INDEL \
 -O "$INDELS_VCF"

# 4. Filter INDELs
gatk VariantFiltration \
 -R "$REFERENCE" \
```

```
 -V "$INDELS_VCF" \
 -O "${OUT_DIR}/${CHR}_indels_filtered.vcf.gz" \
 --filter-name "QD_lt2" --filter-expression "QD < 2.0" \
 --filter-name "FS_gt200" --filter-expression "FS > 200.0" \
 --filter-name "ReadPosRankSum_lt-20" --filter-expression "ReadPosRankSum < -20.0"
```

SLURM Batch Script: MergingVCFs_array.sbatch

```
#!/bin/bash
#SBATCH --partition=batch
#SBATCH --job-name=VariantFiltration
#SBATCH --cpus-per-task=2
#SBATCH --mem=8G
#SBATCH --time=02:00:00
#SBATCH --output=logs/variantfiltration_%j.out
#SBATCH --error=logs/variantfiltration_%j.err
#SBATCH --array=0-21%6

# Before running VariantFiltration we need to merge the per-chromosome VCFs
module load gcc/13.2.0-iqpfkya bcftools/1.19-odjvyvt

OUT_DIR=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/JointGenotyped
MERGED_VCF=$OUT_DIR/combined.vcf.gz

# List all chromosome VCFs
VCF_LIST=$(ls /mnt/sagonas_a/PopGen_HPC/05VariantCalling/JointGenotyped/*.vcf.gz | tr '\n' ' ')

# Merge all chromosome VCFs (concatenate since chromosomes don't overlap)
bcftools concat -a -O z -o $MERGED_VCF $VCF_LIST

# Index the merged VCF as a tbi file
bcftools index --tbi $MERGED_VCF
```

SLURM Batch Script: VariantFiltration.sbatch

```
#!/bin/bash
#SBATCH --partition=batch
#SBATCH --job-name=VariantFiltration
#SBATCH --mem=8G
#SBATCH --time=02:00:00
#SBATCH --output=variantfiltration.out
#SBATCH --error=variantfiltration.err

# In this step we will filter the SNPs. We will use VariantFiltration to flag low quality SNPs

module load gcc/14.2.0 gatk/4.5.0.0

REFERENCE=/mnt/sagonas_a/PopGen_HPC/03ReferenceGenome/GCA_964106915.1_rPodGai
1.hap1.1_genomic.fa
```

```
INPUT_VCF=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/JointGenotyped/combined.vcf.gz
OUTPUT_VCF=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/Filtered/combined.filtered.vcf.g
z
FILTERED_PASS_VCF=/mnt/sagonas_a/PopGen_HPC/05VariantCalling/Filtered/combined.filter
ed.vcf.gz


mkdir -p $(dirname "$OUTPUT_VCF")

gatk VariantFiltration \
 -R "$REFERENCE" \
 -V "$INPUT_VCF" \
 --filter-name "QD_filter" --filter-expression "QD < 2.0" \
 --filter-name "FS_filter" --filter-expression "FS > 60.0" \
 --filter-name "MQ_filter" --filter-expression "MQ < 40.0" \
 --filter-name "MQRankSum_filter" --filter-expression "MQRankSum < -12.5" \
 --filter-name "ReadPosRankSum_filter" --filter-expression "ReadPosRankSum < -8.0" \
 --filter-name "SOR_filter" --filter-expression "SOR > 3.0" \
 -O "$OUTPUT_VCF"

# After running VariantFiltration in GATK, your VCF file will have variants flagged with filter annotations based on the criteria you set. The next step is to filter the variants based on those flags

module load gcc/13.2.0-iqpfkya bcftools/1.19-odjvyvt

bcftools view -f PASS $OUTPUT_VCF -Oz -o $FILTERED_PASS_VCF
bcftools index $FILTERED_PASS_VCF

#To extract SNPs
bcftools view -v snps $FILTERED_PASS_VCF -Oz -o combined.filtered_pass_snps.vcf.gz

#To extract indels
bcftools view -v indels $FILTERED_PASS_VCF -Oz -o combined.filtered_pass_indels.vcf.gz
```