Variants calling and GWAS analysis

CE7412: Computational and Systems Biology

2025-04-04

Contents

1 Sequencing methods

DNA sequencing is the determination of the order of the four nucleotides in a nucleic acid molecule.

1.1 First-generation sequencing

The first-generation sequencing emerged in 1970s when Allan Maxam and Walter Gilbert developed a **chemical method** for sequencing, followed by Frederick Sanger who developed the chain-terminator method (also known as a **Sanger sequencing** method). Both methods were used in shotgun sequencing, which involves breaking the DNA into fragments and then sequncing the fragments individually. The first step in Sanger sequencing is that of PCR. However, sample DNA is divided into for reaction tubes fluorescently labelled by di-deoxynucleotide tripospates: ddATP, ddGTP, ddCTP, and ddTTP. The synthesis terminates in DNA fragments in individual neucleotides which are then separated by size and identified using gel electophoresis.The DNA bands are then graphed by autoradiography and the order of the nucleotide bases on the DNA sequence can be directly read from Xray film.

1.2 Next-generation sequencing (NGS)

Next-generation sequencing (NGS) is a massively parallel DNA sequencing technology that can analyze hundreds of thousands of DNA fragments simultaneously. Compared to earlier sequencing technology, NGS provides sequencing information on multiple fragments simultaneously. DNA is first made single strandard, and fluorescently labelled, and then allow to pair with known libraries/reference genomes. Fluorescent signals detected by the sensors is used to identify the DNA sequence.

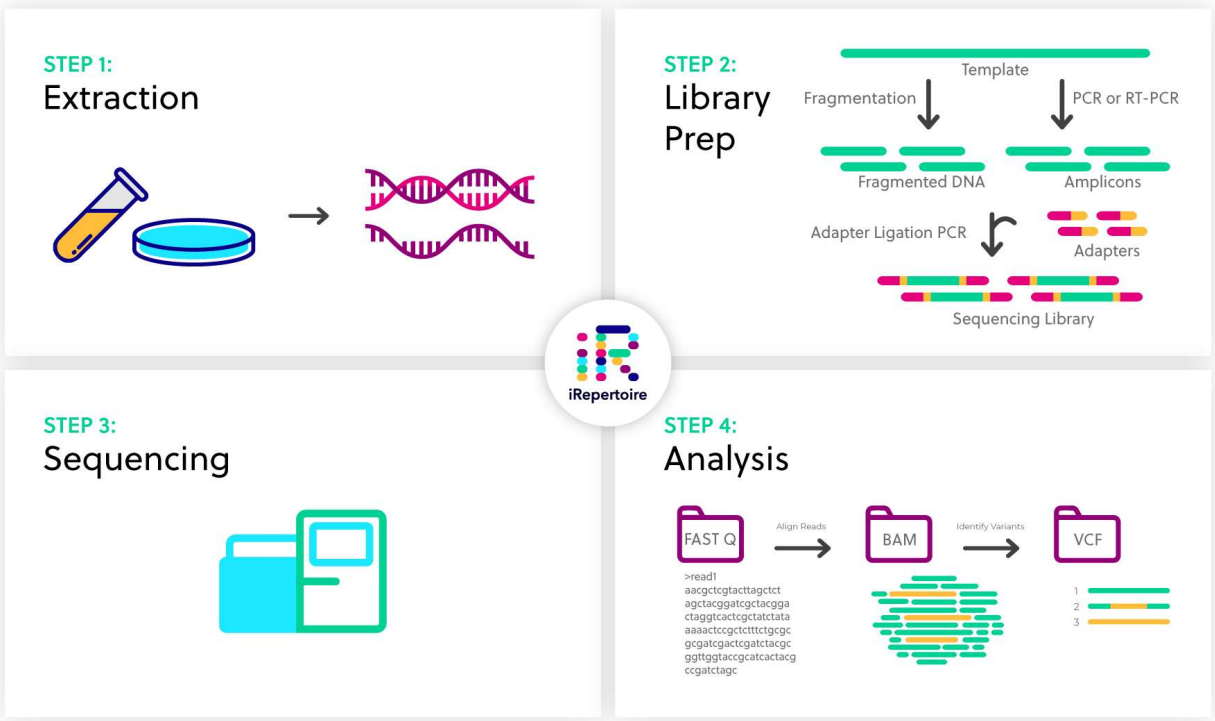


Figure 1: NGS workflow

Next-Generation Sequencing refers to a class of technologies that sequence millions of short DNA fragments in parallel, with a relatively low cost. The length and number of the reads differ based on the technology.

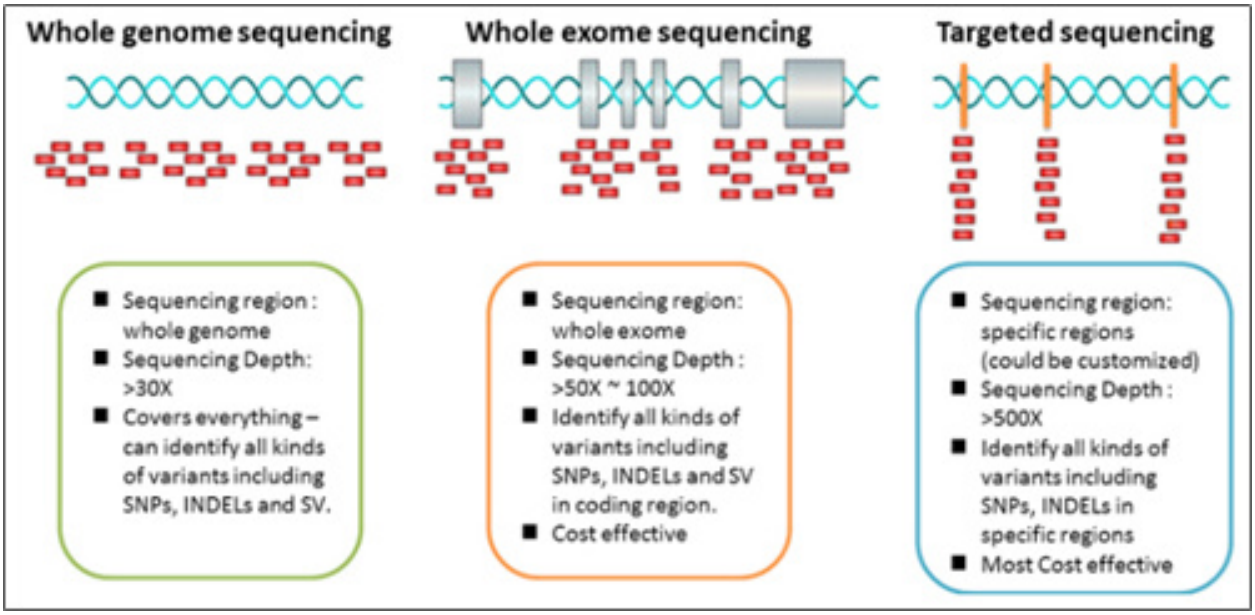
Currently, there are three major commercially available platforms/technologies for NGS: (1) Illumina, (2) Roche, and (3) Life Technologies. The underlying chemistry and technique used in each platform is unique and affects the output.

NGS sequencing enables a wide variety of applications, allowing researchers to ask virtually any question related to the genome, transcriptome, or epigenome of any organism. NGS methods differ primarily by how the DNA or RNA samples are prepared and the data analysis options used. The technology is used to determine the order of nucleotides in entire genomes or targeted regions of DNA or RNA.

1.3 Third-generation sequencing

Third generation sequencing (TGS) is recent sequencing technique that addresses drawbacks of NGS, such as the needs for long reads and poor resolution (ie., poor and weak reads). The foundation of TGS emerged when DNA polymerase was used to obtain a sequence of single DNA molecules, which involves (i) direct imaging of individual DNA molecules using advanced microscopy techniques and (ii) nanopore sequencing technique in which a single molecule of DNA is threaded through a nanopore (a pore of nanoscale). The DNA's passage through the pore creates signals that can be converted to read the DNA sequence

In general, there are two TGS technologies available: (i) Pacific Bioscience (PacBio) single molecule real time sequencing, and (ii) Oxford nanopore technologies.



1.4 WGS (Whole-genome sequencing)

WGS is a comprehensive method of analyzing the entire genomic DNA of a cell at a single time by using sequencing techniques such as Sanger sequencing, shotgun approach, or high throughput NGS sequencing. It is also known as full genome sequencing or complete genome sequencing. WGS enables scientists to read the exact sequence of all the letters that make up the complete set of DNA.

1.5 WES (Whole-exome sequencing)

WES focuses on the genomic protein coding regions (exons). Although WES requires additional reagents (probes) and some additional steps (hybridization), it is a cost-effective, widely used NGS method that requires fewer sequencing reagents and takes less time to perform bioinformatic analysis compared to WGS.

Although the human exome represents only 1-5% of the genome, it contains approximately 85% of known disease-related variants. Despite lengthier sample preparation due to the additional target enrichment step, scientists benefit from quicker sequencing and data analysis compared to WGS.

WES provides greater sequencing depth for researchers interested in identifying genetic variants for numerous applications, including population genetics, genetic disease research, and cancer studies.

2 Sequencing depth and read quality

2.1 Sequencing depth and coverage

The biological results and interpretation of sequencing data for different sequencing applications are greatly affected by the number of sequenced **reads** that cover the genomic regions.

The sequencing **depth** measures the average read abundance and it is calculated as the number of bases of all sequenced short reads that match a genome divided by the length of that genome.

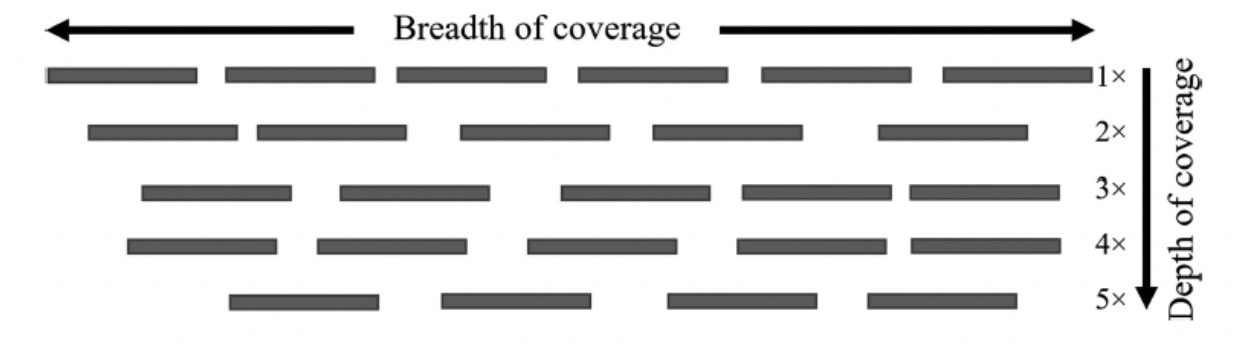


Figure 2: Sequence coverage and depth

The **coverage** gives the average number of reads that align to cover known reference bases. If the reads are equal

$$\text{Coverage} = \frac{\text{read length}(bp) \times \text{number of reads}}{\text{genome size}(bp)}$$

If the reads are not equal in length, the coverage is calculated as

$$\text{Coverage} = \frac{\sum_{i=1}^{n} \text{length of read } i}{\text{genome size}(bp)}$$

where n is the number of sequece reads.

2.2 Base call quality

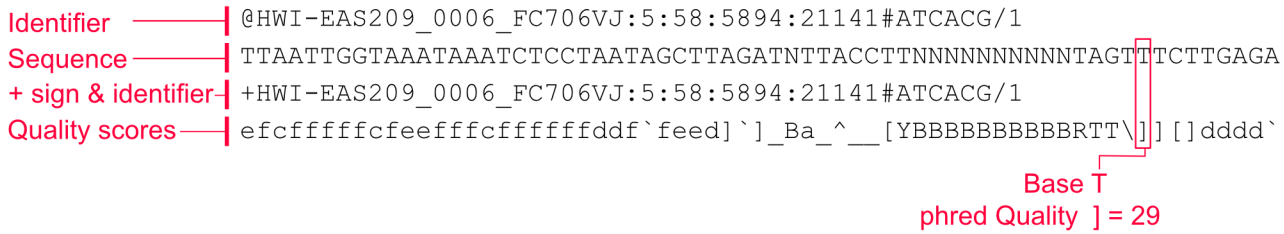
The process of inferring the base at a specific position of the sequenced DNA fragment during the sequencing process is called the **base calling**. Most sequencing platforms are equipped with base calling programs that assign a **Phred quality score** to measure the accuracy of each base called. The Phred quality score (Q-score) transforms the probability of calling a base wrongly into an integer core that is easy to interpret. The Phred score Q is defined as

$$p = 10^{-Q/10}$$
$$Q = -10 \log_{10}(p)$$

where p is the probability of the base call being wrong.

2.3 Fasta and Fastq formats

The **FASTA** format was developed as a text-based format to represent nucleotide or protein sequences. An extension of the FASTA format is **FASTQ** format that is designed to handle base quality metrics output from sequencing machines. In this format, both the sequence and quality scores are represented as single ASCII characters. The format uses four lines for each sequence, and these four lines are stacked on top of each other in text files output by sequencing workflows.



A FastQC file is a quality control report generated by the `FastQC` software, which is used to assess the quality of raw sequence data (typically in FASTQ format) from high-throughput sequencing experiments, providing a visual overview of potential issues like low quality bases, adapter contamination, or nucleotide bias within the reads, allowing researchers to identify problems before further analysis.

3 Efficient data structures for read mapping

Read mapping refers to alignment of reads from high throughput analysis into a reference genome. The basic alignment algorithms do not work here because *millions of reads* have to be aligned to a reference genomes at the same time. Therefore, efficient **indexing algorithms** are needed to organize the sequence of the reference genome and the short reads in a memory efficient manner and to facilitate fast searching of the patterns. The commonly used data structures are:

- suffix trees
- suffix array
- Burrows-Wheeler transform (BWT)

3.1 Suffix Tree

Suffix tree is basically used for pattern matching and finding substrings in a given string of characters. It is constructed as a key and value pairs where all possible suffixes of the reference sequences as *keys* and the positions (indexes) in the sequence as the *values*. The following algorithm, known as **Ukkonen's algorithm** constructs a suffix tree in a linear time.

For example:
let us assume that our *reference sequence* consists of 10 bases as "CTTGGCTGGA$" where the positions are 0, 1, … 9, and $ is in the empty trailing position. Let us form the suffixes (keys) and indexes (values):

$ 10
A $ 9
GA $ 8
GGA $ 7
TGGA $ 6
CTGGA $ 5
GCTGGA $ 4

GGCTGGA $ 3
TGGCTGGA $ 2
TTGGCTGGA $ 1
CTTGGCTGGA $ 0

Note that each line consists of suffix (**key**) and a positon (**value**). Then a suffix tree can be made using key-value pairs as edges and nodes of the tree, respectively.

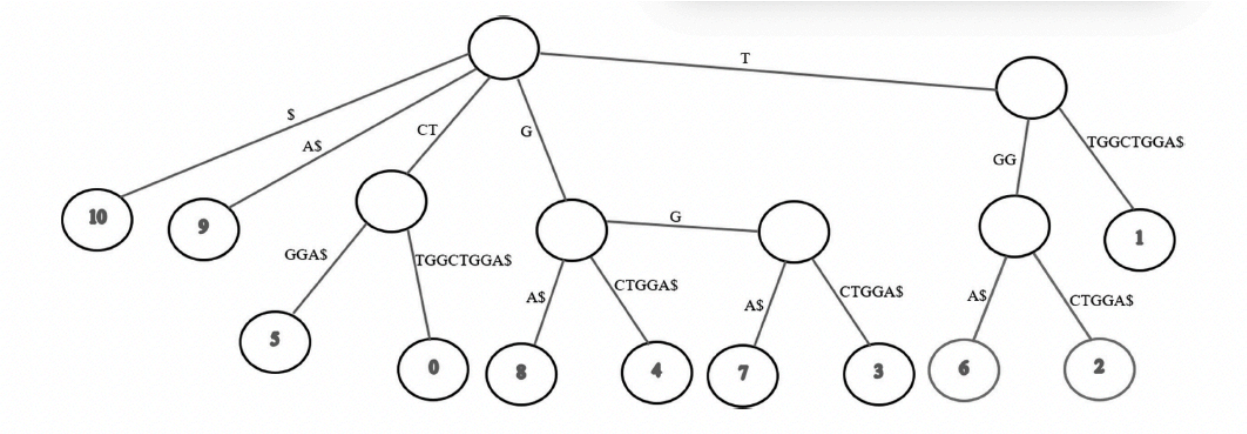


Figure 3: Suffix Tree

Once the suffix tree is built, there are several searching algorithms are available to find the location where a read maps. For instance, to find "TGG" in the tree, we will start looking for T and and then GG. And since there are two leaf nodes, "TGG" can map at positions 2 or 6.

3.2 Suffix Arrays

Suffix arrays can be constructed from suffix trees. It is basically a sorted array of all suffixes in a given sequence. We can quickly search for locations begin with suffixes "TGG" as 7 and 3.

3.3 Burrows-Wheeler Transform

The Borrows-Wheeler Transform (BWT) is a data structure that transform a string (eg., reference sequence) into a compressible form that allows fast searching. The BWT is used by popular aligners like `BWA` and `Bowtie`. The BTW of the sequence s or $bwt(s)$ is computed by generating cyclic rotations of the sequences and then are sorted alphabetically to form a matrix called BWM (*Borrows-Wheeler Matrix*).

To get the ith characeter of BWT of the string s, whose characters are indexed by $i = 1, 2, ...n$, from the suffix array a, simply use:

$$bwt(s)[i] = s[a[i] - 1] \text{ if } a[i] > 1 \text{ else } s[n]$$

BWT could be obtained from a suffix array in a linear time complexity by using the above transform.

i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10|
s | C | T | T | G | G | C | T | G | G | A |
a | 11| 10| 6 | 1 | 9 | 5 | 8 | 4 | 7 | 3 |

|bwt| A | G | G | $ | G | G | T | T | C | T |

BWT serves two purposes. First, BWT groups the characters of the sequence so that a single character appaers many times in a raw because the column is sorted alphabetically and that can be used for sequence

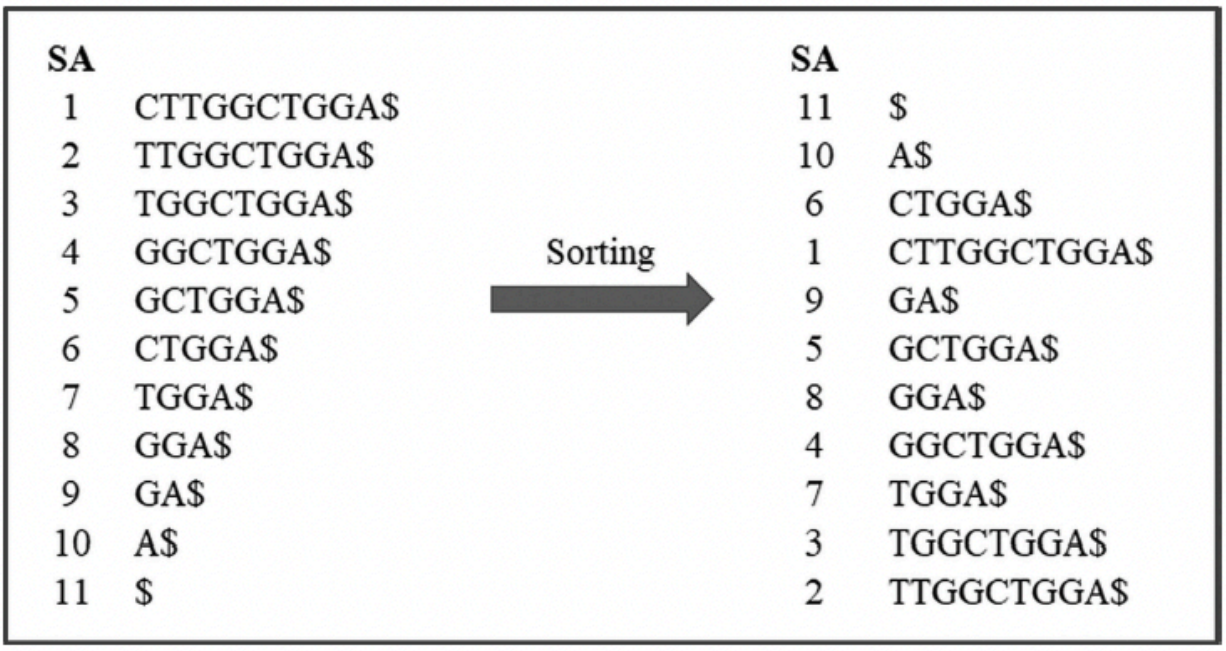


Figure 4: Suffix Array: a sorted array in alphabetical order

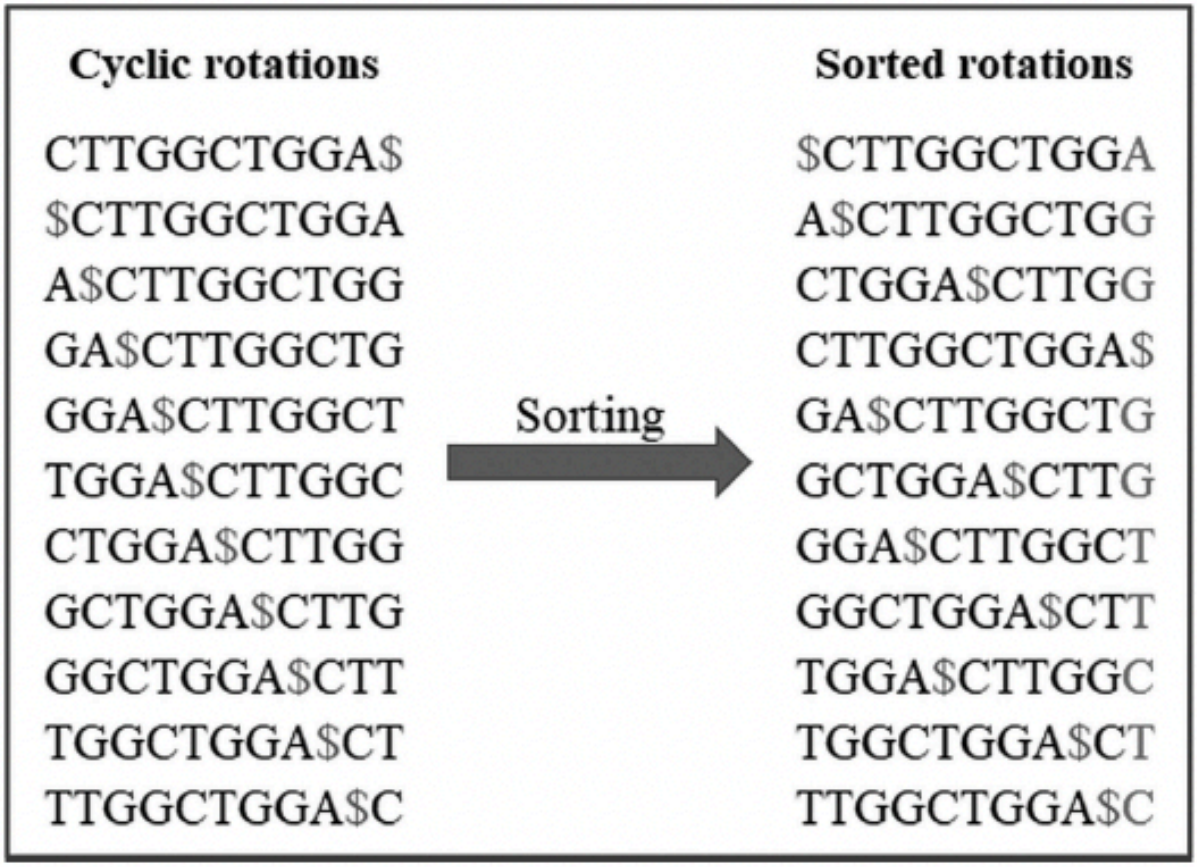


Figure 5: Burrows-Wheeler Transform

compression. The second purpose is BWT data structure can be used for indexing the sequence of reference genome to make finding the position fast.

4 De novo genome assembly

De novo genome assembly comes to play when there is no reference genome available for the organism. The *de novo* genome assembly aims to join reads into a contiguous sequence called a **contig**. Multiple contigs are joined together to form a **scaffold** and multiple scaffolds can be linked to form a chromosome. Assemblying the entire genome is usually challenging but with **deep sequencing** (with high coverage and depth), most of the challenges can be overcome.

The algorithms used for de novo genome assembly are:

- greedy approach
- overlap consensus with Hamiltonian path
- de Bruijn graph with Eulerian path

4.1 Greedy algorithm

It depends on similarity between reads to create a pile up of aligned sequences, which are collapsed to create contigs from the consensus sequences. The greedy algorithm uses pairwise alignment to compare all reads to identify the most similar reads with sufficient overlaps to be merged. The process continues repeatedly until there is no more merging.

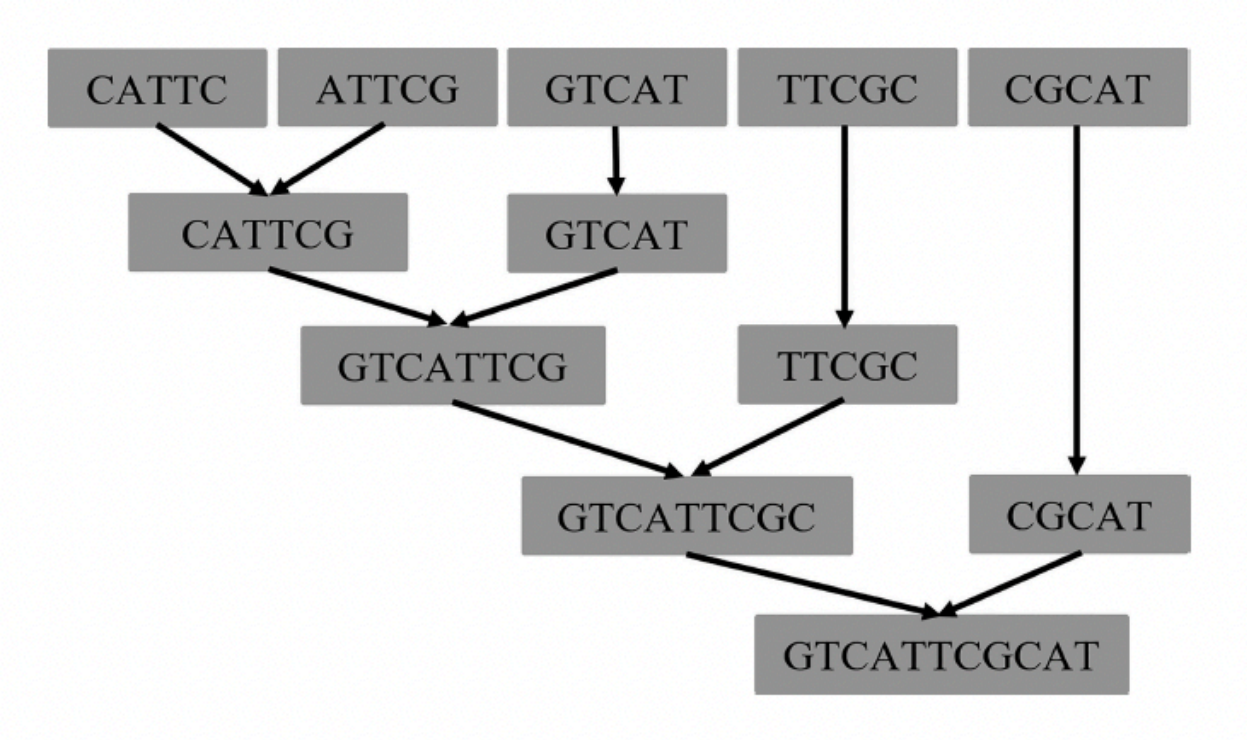


Figure 6: Greedy algorithm for de novo genome assembly

4.2 Overlap consensus graphs

The *overlap-consensus algorithm* performs pairwise alignment and then it represents the reads and the overlaps between the reads with graphs, where contiguous reads are the nodes and their overlaps are the edges.

Each node r_i corresponds to a read and any two reads are connected by an edge $e(r_1, r_2)$ where the suffix is the first read r_1 matches the prefix of the second read r_2. The algorithm then find the **Hamiltonian path** of the graph, which includes the nodes (reads) exactly once. Contigs are then created from the consensus sequencees of the overlapped suffixes and prefixes.

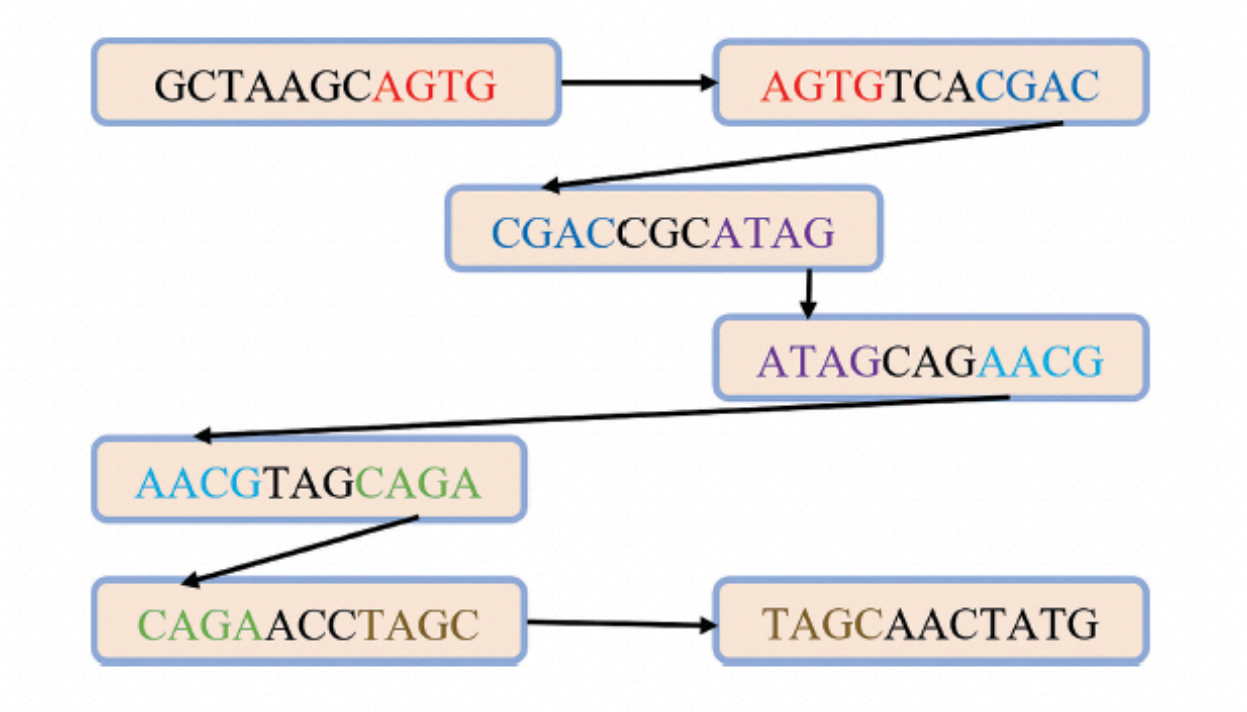


Figure 7: Overlap graphs and Hamiltonian path

4.3 De Bruijn Graphs

In de Bruijn graphs, each read is broken into overlapping substrings of length k called overlapping **k-mers**. The k-mers are then represented by graphs in which each k-mer is a vertex. Any two nodes of any two k-mers that share a commeon prefix and suffix of length $k-1$ are connected by an edge. The contiguous reads are merged by finding the **Eulerian path**, which include every edge exactly once.

5 DNA-Seq analysis

The DNA-Seq analysis pipeline identifies *somatic variants* (changes in DNA, which occurs in cells other than germ cells) within whole exome sequencing (WXS) and whole genome sequencing (WGS) data. Somatic variants are identified by comparing allele frequencies in normal and affected sample alignments, annotating each mutation, and aggregating mutations from multiple cases into one project file.

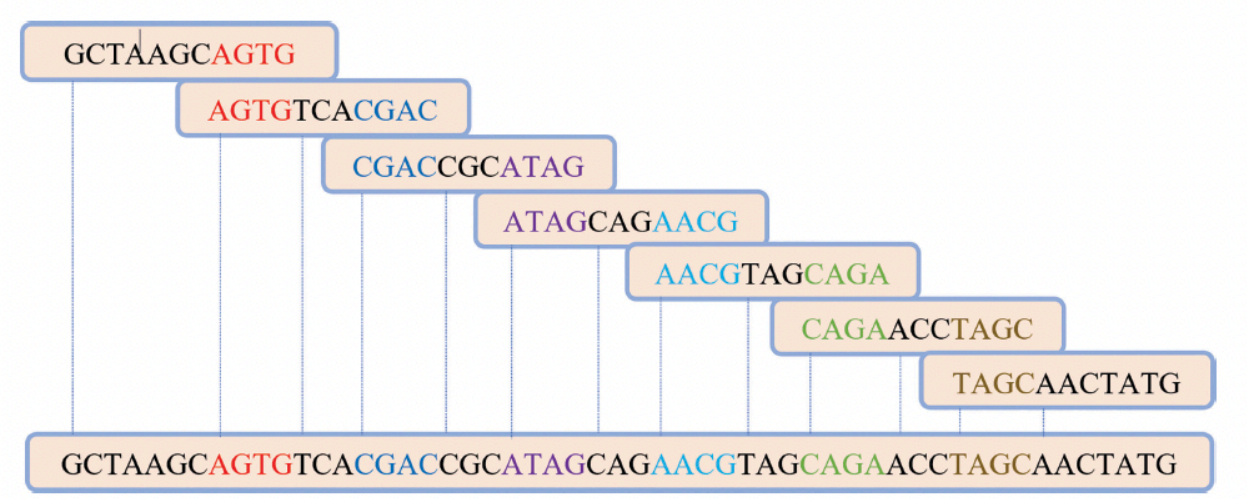


Figure 8: The overlaps and consensus sequence

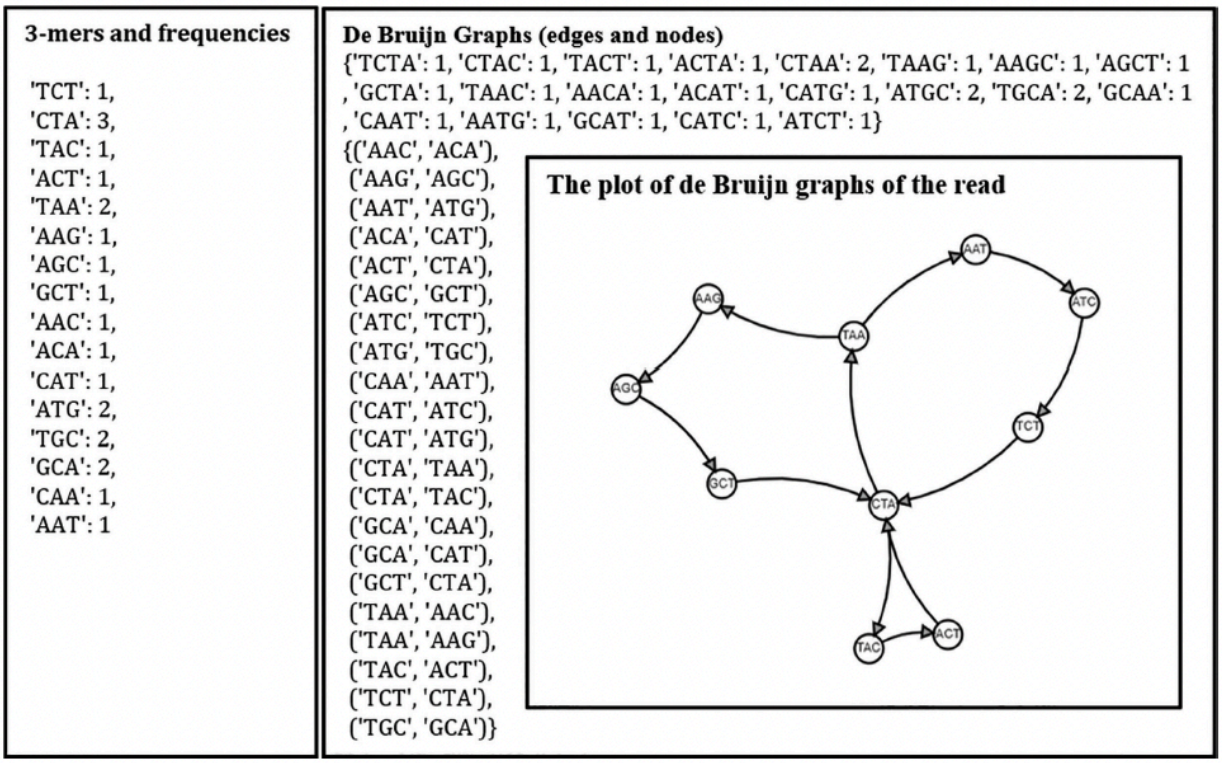


Figure 9: De Bruijn graphs

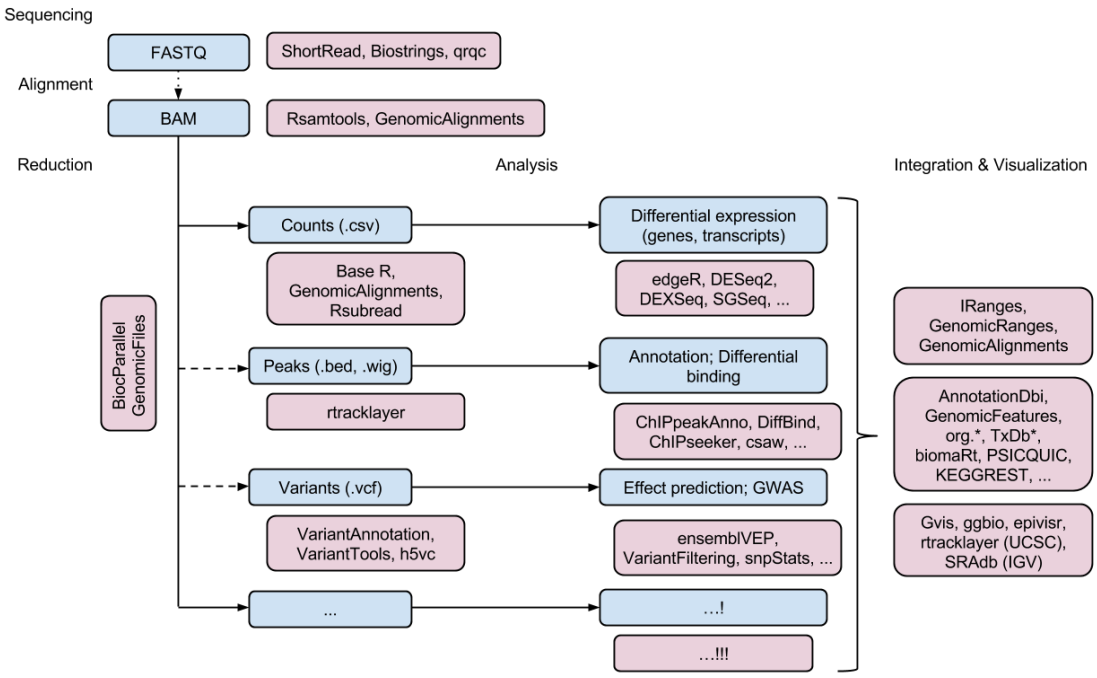
5.1 Steps in DNA-Seq analysis

Variant calling is the process by which we identify variants from sequence data:

- Base calling: Carry out whole genome or whole exome sequencing to create bases for each locations in FASTQ files.
- Filtering
- Read mapping: Align the sequences to a reference genome, creating BAM or CRAM files.
- Quality control
- Variant calling: Identify where aligned reads differ from reference genome and write to a VCF file.
- Filtering
- Population analysis: GWAS

In a DNA sequencing pipeline, "filtering" refers to the process of removing low-quality reads or alignment artifacts from the raw sequencing data before further analysis, typically done by applying quality score thresholds to eliminate reads with poor base calls, adapter sequences, or potential PCR duplicates, significantly improving the accuracy of downstream variant calling and analysis.

5.2 Bioconductor resources for NGS analysis



Credit: https://dockflow.org/workflow/sequencing/

5.3 Extracting information in genomic regions of interest

Very often, one would want to know if a particular genomic feature falls in a particular genomic region of interest. This task can be handled very well by `GRanges` and `SummarizedExperiment` objects. We will look at a few ways to creating these objects and a few ways we can manupulate them.

```
library(GenomicRanges)
library(rtracklayer) # interface to genome annotation files
library(SummarizedExperiment)
```

Create `GRanges` from different files: GFF, BED, and text files.

The General Feature Format (GFF) is a plain text file format used to describe the features of DNA, RNA, and protein sequences. It's used in bioinformatics to store genomic sequences and annotations. The BED (Browser Extensible Data) format is a text file format used to store genomic regions as coordinates and associated annotations.

```
get_granges_from_gff <- function(file_name) {
  gff <- rtracklayer::import.gff(file_name)
  as(gff, "GRanges")
}

get_granges_from_bed <- function(file_name){
  bed <- rtracklayer::import.bed(file_name)
  as(bed, "GRanges")
}

get_granges_from_text <- function(file_name){
  df <- readr::read_tsv(file_name, col_names = TRUE )
  GenomicRanges::makeGRangesFromDataFrame(df, keep.extra.columns = TRUE)
}

get_annotated_regions_from_gff <- function(file_name) {
  gff <- rtracklayer::import.gff(file_name)
  as(gff, "GRanges")
}
```

Get annotated regions from a GFF file:

```
gr_from_gff <- get_annotated_regions_from_gff(file.path(getwd(), "arabidopsis_chr4.gff"))
head(gr_from_gff)
```

```
## GRanges object with 6 ranges and 10 metadata columns:
##       seqnames     ranges strand |   source       type     score     phase
##          <Rle>  <IRanges>  <Rle> | <factor>   <factor> <numeric> <integer>
##   [1]     Chr4 1-18585056      * |   TAIR10 chromosome        NA      <NA>
##   [2]     Chr4  1180-1536      - |   TAIR10 gene              NA      <NA>
##   [3]     Chr4  1180-1536      - |   TAIR10 mRNA              NA      <NA>
##   [4]     Chr4  1180-1536      - |   TAIR10 protein           NA      <NA>
##   [5]     Chr4  1180-1536      - |   TAIR10 CDS               NA         0
##   [6]     Chr4  1180-1536      - |   TAIR10 exon              NA      <NA>
##                         ID        Name               Note
##                <character> <character>     <CharacterList>
##   [1]               Chr4        Chr4
##   [2]           AT4G00005   AT4G00005 protein_coding_gene
##   [3]         AT4G00005.1 AT4G00005.1
##   [4] AT4G00005.1-Protein AT4G00005.1
##   [5]               <NA>        <NA>
##   [6]               <NA>        <NA>
```

```
##                               Parent       Index Derives_from
##                       <CharacterList> <character>  <character>
##   [1]                                         <NA>         <NA>
##   [2]                                         <NA>         <NA>
##   [3]                         AT4G00005           1         <NA>
##   [4]                                         <NA>  AT4G00005.1
##   [5] AT4G00005.1,AT4G00005.1-Protein         <NA>         <NA>
##   [6]                       AT4G00005.1         <NA>         <NA>
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Get genomic regaions from text files:

```
gr_from_txt <- get_granges_from_text(file.path(getwd(), "arabidopsis_chr4.txt"))
head(gr_from_txt)
```

```
## GRanges object with 6 ranges and 2 metadata columns:
##       seqnames      ranges strand |        type  feature_id
##          <Rle>   <IRanges>  <Rle> | <character> <character>
##   [1]     Chr4   1180-1536      - |        gene   AT4G00005
##   [2]     Chr4  2895-10455      - |        gene   AT4G00020
##   [3]     Chr4 11355-13359      - |        gene   AT4G00026
##   [4]     Chr4 13527-14413      + |        gene   AT4G00030
##   [5]     Chr4 14627-16079      + |        gene   AT4G00040
##   [6]     Chr4 17792-20066      + |        gene   AT4G00050
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Extract `GRanges` of selected gene:

```
## Extract by seqname or metadata
genes_on_chr4 <- gr_from_gff[ gr_from_gff$type == "gene" & seqnames(gr_from_gff) %in% c("Chr4") ]
head(genes_on_chr4)
```

```
## GRanges object with 6 ranges and 10 metadata columns:
##       seqnames      ranges strand |   source     type     score     phase
##          <Rle>   <IRanges>  <Rle> | <factor> <factor> <numeric> <integer>
##   [1]     Chr4   1180-1536      - |   TAIR10     gene        NA      <NA>
##   [2]     Chr4  2895-10455      - |   TAIR10     gene        NA      <NA>
##   [3]     Chr4 11355-13359      - |   TAIR10     gene        NA      <NA>
##   [4]     Chr4 13527-14413      + |   TAIR10     gene        NA      <NA>
##   [5]     Chr4 14627-16079      + |   TAIR10     gene        NA      <NA>
##   [6]     Chr4 17792-20066      + |   TAIR10     gene        NA      <NA>
##                ID        Name                Note          Parent       Index
##       <character> <character>     <CharacterList> <CharacterList> <character>
##   [1]   AT4G00005   AT4G00005 protein_coding_gene                         <NA>
##   [2]   AT4G00020   AT4G00020 protein_coding_gene                         <NA>
##   [3]   AT4G00026   AT4G00026 protein_coding_gene                         <NA>
##   [4]   AT4G00030   AT4G00030 protein_coding_gene                         <NA>
##   [5]   AT4G00040   AT4G00040 protein_coding_gene                         <NA>
##   [6]   AT4G00050   AT4G00050 protein_coding_gene                         <NA>
##       Derives_from
```

```
##        <character>
##   [1]         <NA>
##   [2]         <NA>
##   [3]         <NA>
##   [4]         <NA>
##   [5]         <NA>
##   [6]         <NA>
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Manually creat a region of interest and read it as `GRanges` object:

```
## By range, create synthetic ranges
region_of_interest_gr <- GRanges(
  seqnames = c("Chr4"),
  IRanges(c(10000), width= c(1000))
)
region_of_interest_gr
```

```
## GRanges object with 1 range and 0 metadata columns:
##       seqnames      ranges strand
##          <Rle>   <IRanges>  <Rle>
##   [1]     Chr4 10000-10999      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Find the overlapping genes of `Chr4` in the simulated region:

```
overlap_hits <- findOverlaps(region_of_interest_gr, genes_on_chr4)
features_in_region <- genes_on_chr4[subjectHits(overlap_hits) ]
features_in_region
```

```
## GRanges object with 1 range and 10 metadata columns:
##       seqnames     ranges strand |   source     type     score     phase
##          <Rle>  <IRanges>  <Rle> | <factor> <factor> <numeric> <integer>
##   [1]     Chr4 2895-10455      - |   TAIR10     gene        NA      <NA>
##                ID        Name                Note          Parent       Index
##       <character> <character>     <CharacterList> <CharacterList> <character>
##   [1]   AT4G00020   AT4G00020 protein_coding_gene                        <NA>
##       Derives_from
##        <character>
##   [1]         <NA>
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Create a random `SummarizedExperiment` that uses `GRanges` object and matrix from simulated data.

```
set.seed(4321)
experiment_counts <- matrix( runif(4308 * 6, 1, 100), 4308) # generate six experiments
sample_names <- c(rep("ctrl",3), rep("test",3) ) # three control and test samples
se <- SummarizedExperiment(rowRanges = gr_from_txt, assays = list(experiment_counts), colData = sample_
```

Let us find the overlapping regions in the simulated experiment data. Assay returns number of hits.

```
overlap_hits <- findOverlaps(region_of_interest_gr, se)
data_in_region <- se[subjectHits(overlap_hits) ]
assay(data_in_region)
```

```
##          [,1]     [,2]    [,3]     [,4]    [,5]     [,6]
## [1,] 91.00481 34.41582 42.7602 36.13053 47.6775 19.21672
```

5.4 Predicting open reading frames in long reference sequences

Often times, gene annotations are not usually available. Let us look a first stage pipline for finding potential genes and genomic loci of interest absolutely *de novo* and without information beyond the sequence.

Let us use a simple set of rules to find **open reading frames (ORF)** - sequences that begin with a start codon and end with a stop codon. Will use `systemPipeR` package to find ORF and create `GRanges` object that can be used for downstream analysis.

```
library(Biostrings)
library(systemPipeR) # to predict ORF
```

Read the genomic sequence of `arabidopsis chloroplast` from fasta file.

```
# Use arabidopsis_chloroplast genome sequence as input (from a fasta file)
dna_object <- readDNAStringSet(file.path(getwd(), "arabidopsis_chloroplast.fa"))
```

Compute the properties of reference genome

```
bases <- c("A", "C", "T", "G")
raw_seq_string <- strsplit(as.character(dna_object), "")

seq_length <- width(dna_object[1])
counts <- letterFrequency(dna_object[1], letters = c("A", "T", "G", "C"))
probs <- unlist(lapply(counts, function(base_count){signif(base_count / seq_length, 2) }))
dna_object[1]
```

```
## DNAStringSet object of length 1:
##      width seq                                                names
## [1] 154478 ATGGGCGAACGACGGGAATTGAA...TAATAACTTGGTCCCGGGCATC chloroplast
```

```
probs
```

```
## [1] 0.31 0.32 0.18 0.18
```

```
predicted_orfs <- predORF(dna_object, n='all', type='gr', mode='ORF', strand = 'both',
                          longest_disjoint = TRUE)
predicted_orfs
```

```
## GRanges object with 2501 ranges and 2 metadata columns:
##           seqnames        ranges strand | subject_id inframe2end
##              <Rle>     <IRanges>  <Rle> |  <integer>   <numeric>
##      1 chloroplast   86762-93358      + |          1           2
##   1162 chloroplast     2056-2532      - |          1           3
##      2 chloroplast   72371-73897      + |          2           2
##   1163 chloroplast   77901-78362      - |          2           1
##      3 chloroplast   54937-56397      + |          3           3
##    ...         ...           ...    ... .        ...         ...
##   2497 chloroplast 129757-129762      - |       1336           3
##   2498 chloroplast 139258-139263      - |       1337           3
##   2499 chloroplast 140026-140031      - |       1338           3
##   2500 chloroplast 143947-143952      - |       1339           3
##   2501 chloroplast 153619-153624      - |       1340           3
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Find the longest ORF length of random genome and keep the ORF that are longer than that in the given genome:

```
# write a function to return longest ORF
get_longest_orf_in_random_genome <- function(x,
                                             length = 1000,
                                             probs = c(0.25, 0.25, 0.25, 0.25),
                                             bases = c("A","C","T","G")) {
  random_genome <- paste0(sample(bases, size = length, replace = TRUE, prob = probs), collapse = "")
  random_dna_object <- DNAStringSet(random_genome)
  names(random_dna_object) <- c("random_dna_string")
  orfs <- predORF(random_dna_object, n = 1, type = 'gr', mode='ORF', strand = 'both',
                  longest_disjoint = TRUE)
  return(max(width(orfs)))
}

# generate 10 simulated random genomes
random_lengths <- unlist(lapply(1:10, get_longest_orf_in_random_genome, length = seq_length,
                                probs = probs, bases = bases))

# get length of longest random ORF
longest_random_orf <- max(random_lengths)

# Keep only the predicted ORF longer than the longest random ORF
keep <- width(predicted_orfs) > longest_random_orf
orfs_to_keep <- predicted_orfs[keep]
orfs_to_keep
```

```
## GRanges object with 10 ranges and 2 metadata columns:
##         seqnames        ranges strand | subject_id inframe2end
##            <Rle>     <IRanges>  <Rle> |  <integer>   <numeric>
##    1 chloroplast   86762-93358      + |          1           2
##    2 chloroplast   72371-73897      + |          2           2
##    3 chloroplast   54937-56397      + |          3           3
##    4 chloroplast   57147-58541      + |          4           1
##    5 chloroplast   33918-35141      + |          5           1
```

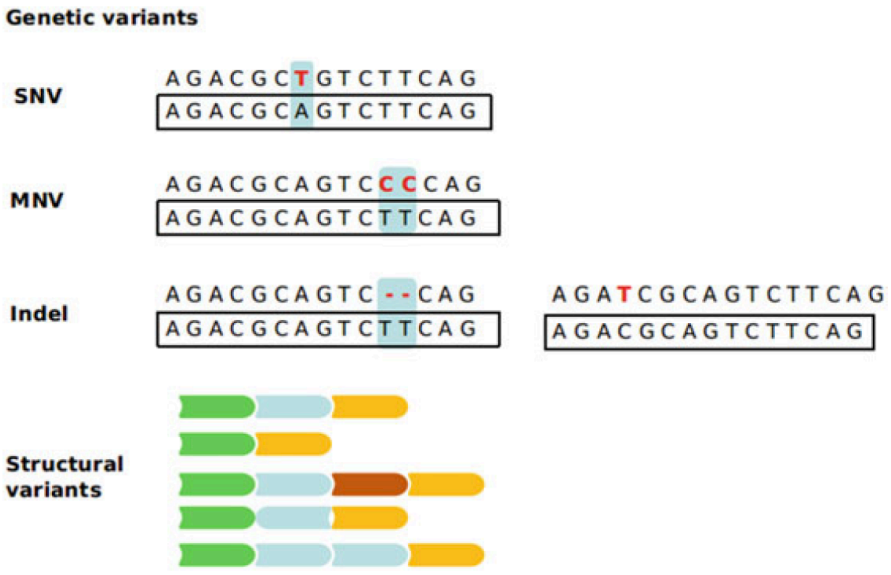
```
##     6 chloroplast   32693-33772      + |          6          2
##     7 chloroplast 109408-110436      + |          7          3
##     8 chloroplast 114461-115447      + |          8          2
##     9 chloroplast 141539-142276      + |          9          2
##    10 chloroplast   60741-61430      + |         10          1
##    -------
##    seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

```
##writing the resutls to a file in fasta format
extracted_orfs <- BSgenome::getSeq(dna_object, orfs_to_keep)
names(extracted_orfs) <- paste0("orf_", 1:length(orfs_to_keep))
writeXStringSet(extracted_orfs, "saved_orfs.fa")
```

6 Genomic variants

Genomic variation refers to DNA sequence differences among individuals or populations.

6.1 Types of genomic variants



6.1.1 SNVs (single nucleotide variants):

also known as **single base substitutions**, are the simplest type of variation as they only involve the change of one base for another in a DNA sequence. These can be subcategorized into **transitions** (Ti) and **transversions** (Tv); *Ti*s are changes between two purines or between two pyrimidines, whereas the latter involve a change from a purine to a pyrimidine or vice versa.

If the SNV is common in a population (usually with an allele frequency $> 1\%$), then it is referred to as a **SNP (single nucleotide polymorphism)**.

6.1.2 MNVs (multi-nucleotide variants):

which are sequence variants that involve consecutive change of two or more bases. Similarly to SNVs, there are some MNVs that are found at higher frequencies in the population, which are referred to as **MNPs (multi-nucleotide polymorphism)**.

6.1.3 Indels (insertions and deletions):

which involve the gain or loss of one or more bases in a sequence. Usually, what is referred to as **indel** tends to be only a few bases in length.

6.1.4 Structural variants:

which are genomic variations that involve larger segments of the genome:

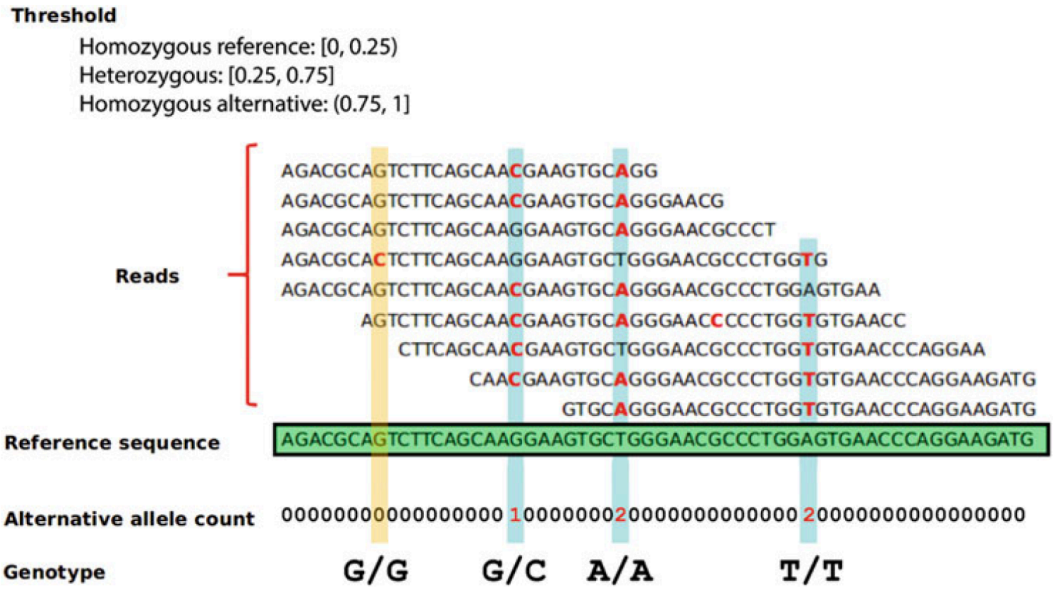
- **inversions**, which is when a certain sequence in the genome gets reversed end to end
- **copy number variants (CNV)** including amplifications, when a fraction of genome gets duplicated one or more times
- larger **deletions**, when large segments of the genome get lost
- larger **insertions**

There is not a strict rule defining the number of base pairs that make the difference between an indel and a structural variant, but usually, a gain or loss of DNA would be called a **structural variant** if it involved more than *one kilobase* of sequence.

6.2 Methods of variant calling

6.2.1 Naive variant calling

A naive approach to determining the **genotypes** from a pile of sequencing reads mapped to a site in the genome is to count the number of variants in alternate alleles against the reference. And then establish hard genotype thresholds.



In this method, reads are aligned to the reference sequence (green) and a threshold of the proportion of reads supporting each allele for calling genotypes is established (top). Then, at each position, the proportion of reads supporting the alternative allele is calculated and, based on the dosage of the alternative allele, a genotype is established. Yellow: a position where a variant is present but the proportion of alternative alleles does not reach the threshold ($1/6 < 0.25$).

6.2.2 Bayesian variant calling

The Bayesian approach includes information about the prior probability of a variant occurring at that site and the amount of information supporting each of the potential genotypes. The posterior probability of a genotype g given genomic data d is given by

$$P(g|d) = \frac{P(d|g)P(g)}{P(d)}$$

where $P(d|g)$ is the likelihood of sequence data d given the genotype and $P(g)$ the prior probability of genotype g. The prior probabilities of a paticular genotype can be estimated over the whole sequence. The likelood can be generated from the alignment of sequences (For every combination of reference allele (site types) and nucleotide in every read, the probability of the observed allele being the same as the reference is calculated. These probabilities are then multiplied for all nucleotides in the reads at that position).

$P(d)$ can be computed as

$$P(d) = \sum_{g'} P(d|g')P(g')$$

6.2.3 Heuristic methods

these methods rely on heuristic quantities and algorithms to call a variant site, such as a minimum coverage and alignment quality thresholds, and stringent cut-offs like a minimum number of reads supporting a variant allele.

6.3 Finding SNPs and indels from sequence data using `VariantTools`

A key task in sequence analysis is to take an alignment of high-throughput sequences stored in BAM file (Binary Alignment Map) and compute a list of variant positions. Note that after FASTAQ file alignment to the reference genome, the outputs appear in BAM files. The results of variant calling is usually stored in a VCF file of variants.

```
library(GenomicRanges) # handles genomic locations within a genome
library(gmapR) # align short-range data
library(rtracklayer) #interface to genome annotation files and the UCSC genome browser
library(VariantAnnotation)
library(VariantTools)
```

We will use a set of synthetic reads from human genome chromosome 17. Let us first read `.bam` and `.fa` files:

```
# get the directory of bam files folder
bam_folder <- file.path(getwd())

# reference genome
bam_file <- file.path( bam_folder, "hg17_snps.bam")
```

```
# testing genome
fasta_file <- file.path(bam_folder,"chr17.83k.fa")
```

Create a GmapGenome object

```
fa <- rtracklayer::FastaFile(fasta_file)
genome <- gmapR::GmapGenome(fa, create=TRUE)
```

Create a parameter object and call the Varints

```
qual_params <- TallyVariantsParam(genome = genome, minimum_mapq = 20) # min Q
var_params <- VariantCallingFilters(read.count = 19, p.lower = 0.01)

called_variants <- callVariants(bam_file, qual_params, calling.filters = var_params)
head(called_variants)
```

```
## VRanges object with 6 ranges and 17 metadata columns:
##           seqnames    ranges strand         ref              alt     totalDepth
##              <Rle> <IRanges>  <Rle> <character> <characterOrRle> <integerOrRle>
##   [1] NC_000017.10        64      *           G                T            759
##   [2] NC_000017.10        69      *           G                T            812
##   [3] NC_000017.10        70      *           G                T            818
##   [4] NC_000017.10        73      *           T                A            814
##   [5] NC_000017.10        77      *           T                A            802
##   [6] NC_000017.10        78      *           G                T            798
##            refDepth       altDepth   sampleNames softFilterMatrix | n.read.pos
##       <integerOrRle> <integerOrRle> <factorOrRle>         <matrix> |  <integer>
##   [1]            739             20          <NA>                  |         17
##   [2]            790             22          <NA>                  |         19
##   [3]            796             22          <NA>                  |         20
##   [4]            795             19          <NA>                  |         13
##   [5]            780             22          <NA>                  |         19
##   [6]            777             21          <NA>                  |         17
##       n.read.pos.ref raw.count.total count.plus count.plus.ref count.minus
##            <integer>       <integer>  <integer>      <integer>   <integer>
##   [1]             64             759         20            739           0
##   [2]             69             812         22            790           0
##   [3]             70             818         22            796           0
##   [4]             70             814         19            795           0
##   [5]             70             802         22            780           0
##   [6]             70             798         21            777           0
##       count.minus.ref count.del.plus count.del.minus read.pos.mean
##             <integer>      <integer>       <integer>     <numeric>
##   [1]               0              0               0       30.9000
##   [2]               0              0               0       40.7273
##   [3]               0              0               0       34.7727
##   [4]               0              0               0       36.1579
##   [5]               0              0               0       38.3636
##   [6]               0              0               0       39.7143
##       read.pos.mean.ref read.pos.var read.pos.var.ref     mdfne mdfne.ref
##                 <numeric>    <numeric>        <numeric> <numeric> <numeric>
##   [1]           32.8755      318.558          347.804        NA        NA
```

```
##   [2]           35.4190      377.004          398.876        NA        NA
##   [3]           36.3442      497.762          402.360        NA        NA
##   [4]           36.2176      519.551          402.843        NA        NA
##   [5]           36.0064      472.327          397.070        NA        NA
##   [6]           35.9241      609.076          390.463        NA        NA
##       count.high.nm count.high.nm.ref
##           <integer>         <integer>
##   [1]            20               738
##   [2]            22               789
##   [3]            22               796
##   [4]            19               769
##   [5]            22               780
##   [6]            21               777
##   -------
##   seqinfo: 1 sequence from chr17.83k genome
##   hardFilters(4): nonRef nonNRef readCount likelihoodRatio
```

Write VCF file

```
VariantAnnotation::sampleNames(called_variants) <- "sample_name"
vcf <- VariantAnnotation::asVCF(called_variants)
VariantAnnotation::writeVcf(vcf, "hg17.vcf")
```

Load the annotations and feature positions

```
get_annotated_regions_from_gff <- function(file_name) {
  gff <- rtracklayer::import.gff(file_name)
  as(gff, "GRanges")
}

get_annotated_regions_from_bed <- function(file_name){
  bed <- rtracklayer::import.bed(file_name)
  as(bed, "GRanges")
}
```

get the test genome

```
genes <- get_annotated_regions_from_gff(file.path( bam_folder, "chr17.83k.gff3"))
```

calculate which variants overlap with genes and subest the genes

```
overlaps <- GenomicRanges::findOverlaps(called_variants, genes)
genes[subjectHits(overlaps)][1:6]
```

```
## GRanges object with 6 ranges and 20 metadata columns:
##           seqnames      ranges strand |   source       type     score     phase
##              <Rle>   <IRanges>  <Rle> | <factor>   <factor> <numeric> <integer>
##   [1] NC_000017.10 64099-76866      - |   havana ncRNA_gene        NA      <NA>
##   [2] NC_000017.10 64099-76866      - |   havana lnc_RNA           NA      <NA>
##   [3] NC_000017.10 64099-65736      - |   havana exon              NA      <NA>
##   [4] NC_000017.10 64099-76866      - |   havana ncRNA_gene        NA      <NA>
##   [5] NC_000017.10 64099-76866      - |   havana lnc_RNA           NA      <NA>
```

```
##   [6] NC_000017.10 64099-65736      - |   havana exon             NA      <NA>
##                            ID            Name     biotype      description
##                   <character>     <character> <character>      <character>
##   [1]   gene:ENSG00000280279      AC240565.2     lincRNA novel transcript
##   [2] transcript:ENST00000..  AC240565.2-201     lincRNA             <NA>
##   [3]                   <NA> ENSE00003759547        <NA>             <NA>
##   [4]   gene:ENSG00000280279      AC240565.2     lincRNA novel transcript
##   [5] transcript:ENST00000..  AC240565.2-201     lincRNA             <NA>
##   [6]                   <NA> ENSE00003759547        <NA>             <NA>
##               gene_id  logic_name     version                 Parent
##           <character> <character> <character>        <CharacterList>
##   [1] ENSG00000280279      havana           1
##   [2]             <NA>        <NA>           1   gene:ENSG00000280279
##   [3]             <NA>        <NA>           1 transcript:ENST00000..
##   [4] ENSG00000280279      havana           1
##   [5]             <NA>        <NA>           1   gene:ENSG00000280279
##   [6]             <NA>        <NA>           1 transcript:ENST00000..
##                tag   transcript_id transcript_support_level constitutive
##        <character>     <character>              <character>  <character>
##   [1]         <NA>            <NA>                     <NA>         <NA>
##   [2]        basic ENST00000623180                        5         <NA>
##   [3]         <NA>            <NA>                     <NA>            1
##   [4]         <NA>            <NA>                     <NA>         <NA>
##   [5]        basic ENST00000623180                        5         <NA>
##   [6]         <NA>            <NA>                     <NA>            1
##       ensembl_end_phase ensembl_phase         exon_id        rank
##             <character>   <character>     <character> <character>
##   [1]              <NA>          <NA>            <NA>        <NA>
##   [2]              <NA>          <NA>            <NA>        <NA>
##   [3]                -1            -1 ENSE00003759547           5
##   [4]              <NA>          <NA>            <NA>        <NA>
##   [5]              <NA>          <NA>            <NA>        <NA>
##   [6]                -1            -1 ENSE00003759547           5
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

6.4 Ploting features of genetic maps

Often, we want to see on a chromosome or genetic map where some features of interest lie in relation to others. These plots are called *chromosome plots* or *ideograms*, and we will see how `karyoploteR` can do.

```
library(karyoploteR)
library(GenomicRanges)
```

Create a `GRanage` object: five genomic ranges on chromosomes 1-5:

```
genome_df <- data.frame(
  chr = paste0("chr", 1:5),
  start = rep(1, 5),
  end = c(34964571, 22037565, 25499034, 20862711, 31270811)
)
genome_gr <- makeGRangesFromDataFrame(genome_df)
genome_gr
```

```
## GRanges object with 5 ranges and 0 metadata columns:
##       seqnames     ranges strand
##          <Rle>  <IRanges>  <Rle>
##   [1]     chr1 1-34964571      *
##   [2]     chr2 1-22037565      *
##   [3]     chr3 1-25499034      *
##   [4]     chr4 1-20862711      *
##   [5]     chr5 1-31270811      *
##   -------
##   seqinfo: 5 sequences from an unspecified genome; no seqlengths
```

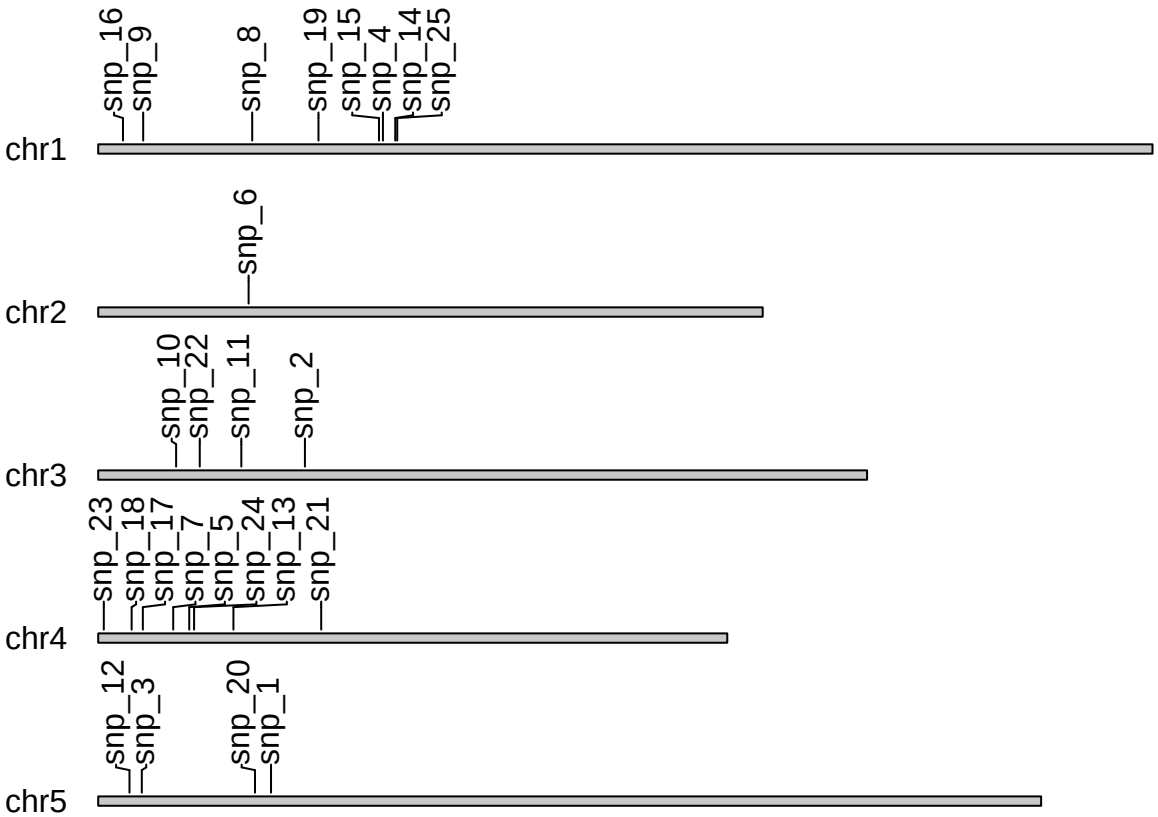
Set up SNPs positions to draw as markers

```
snp_pos <- sample(1:1e7, 25)
snps <- data.frame(
  chr = paste0("chr", sample(1:5,25, replace=TRUE)),
  start = snp_pos,
  end = snp_pos
)
snps_gr <- makeGRangesFromDataFrame(snps)
snp_labels <- paste0("snp_", 1:25)

snps_gr
```

```
## GRanges object with 25 ranges and 0 metadata columns:
##        seqnames    ranges strand
##           <Rle> <IRanges>  <Rle>
##    [1]     chr5   5731177      *
##    [2]     chr3   6855485      *
##    [3]     chr5   1445614      *
##    [4]     chr1   9442378      *
##    [5]     chr4   3018935      *
##    ...      ...       ...    ...
##   [21]     chr4   7395203      *
##   [22]     chr3   3367345      *
##   [23]     chr4    185686      *
##   [24]     chr4   3180060      *
##   [25]     chr1   9916951      *
##   -------
##   seqinfo: 5 sequences from an unspecified genome; no seqlengths
```

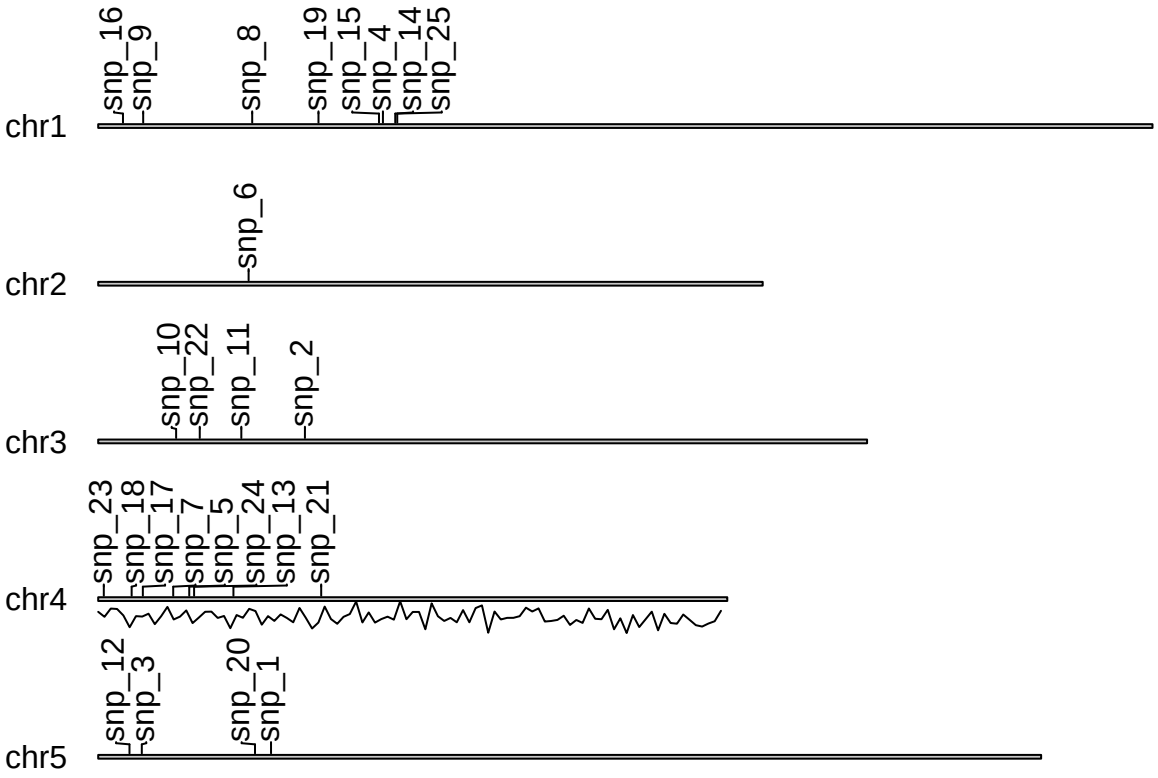
Plot SNP locations:

```
plot.params <- getDefaultPlotParams(plot.type=1)
plot.params$data1outmargin <- 600

kp <- plotKaryotype(genome=genome_gr, plot.type = 1, plot.params = plot.params)
kpPlotMarkers(kp, snps_gr, labels = snp_labels)
```



We can add some numeric data to the plot if they are available. For example, lets introduce some random plots for chr4:

```
### create random data to chr4
numeric_data <- data.frame(
  y = rnorm(100,mean = 1,sd = 0.5  ),
  chr = rep("chr4", 100),
  start = seq(1,20862711, 20862711/100),
  end = seq(1,20862711, 20862711/100)
)

numeric_data_gr <- makeGRangesFromDataFrame(numeric_data)

plot.params <- getDefaultPlotParams(plot.type=2)
plot.params$data1outmargin <- 800
plot.params$data2outmargin <- 800
plot.params$topmargin <- 800

kp <- plotKaryotype(genome=genome_gr, plot.type = 2, plot.params = plot.params)
kpPlotMarkers(kp, snps_gr, labels = snp_labels)
kpLines(kp, numeric_data_gr, y = numeric_data$y, data.panel=2)
```



6.5 Estimating the copy number at a locus of interest

We often want to know if a locus has been duplicated or its **copy number** has increased. Our approach is to use DNA-seq read coverage after alignment to estimate a background level of coverage and then inspect the coverage on the region of interest. The ratio of coverage give an estimate of the copy number of the region.

```
library(csaw) # Bioconductor package for find CNV
```

Get counts in windows across the hg17 genome:

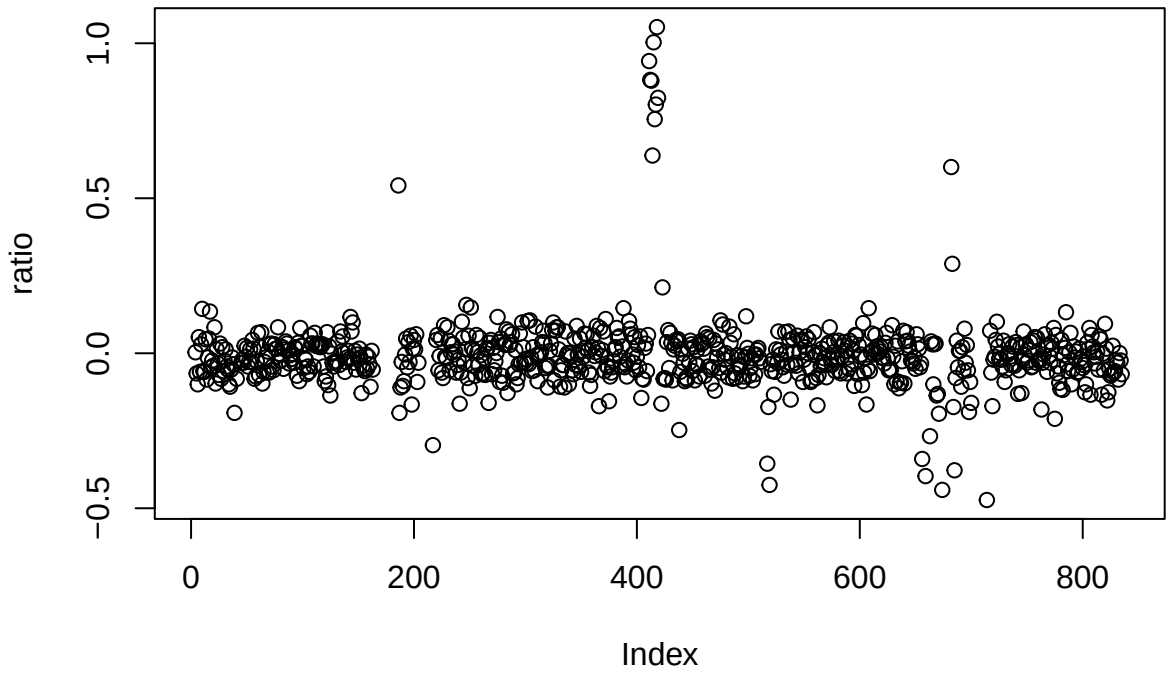
```
whole_genome <- csaw::windowCounts(
  file.path(getwd(), "hg17_snps.bam"), # the bam file contains counts
  bin = TRUE,
  filter = 0,
  width = 100, # the width of the window
  param = csaw::readParam(
    minq = 20, # minimum mapping quality
    dedup = TRUE,
    pe = "both" # paired-end or single-end or both
  )
)
colnames(whole_genome) <- c("h17")
```

Extract the data from `SummarizedExperiment` object. Set a threshold and make lower counts to NA.

```
counts <- assay(whole_genome)[,1]

min_count <- quantile(counts, 0.1)[[1]]
counts[counts < min_count] <- NA # lowest counts to NA
```

Calculate the mean coverage and ratio in each window to that mean coverage, and inspect the ratio vector with a plot.

```
n <- length(counts)
doubled_windows <- 10

left_pad <- floor( (n/2) - doubled_windows )
right_pad <- n - left_pad -doubled_windows
multiplier <- c(rep(1, left_pad ), rep(2,doubled_windows), rep(1, right_pad) )
counts <- counts * multiplier

mean_cov <- mean(counts, na.rm=TRUE)

ratio <- matrix(log2(counts / mean_cov), ncol = 1)
plot(ratio)
```



Build a `SummarisedExperiment` with new data

```
se <- SummarizedExperiment(assays=list(ratio), rowRanges= rowRanges(whole_genome),
                           colData = c("CoverageRatio"))
```

Create a region of interest and extract the coverage from it

```
region_of_interest <- GRanges(
  seqnames = c("NC_000017.10"),
  IRanges(c(40700), width = c(1500) )
```

```
)

overlap_hits <- findOverlaps(region_of_interest, se)
data_in_region <- se[subjectHits(overlap_hits)]
assay(data_in_region)
```

```
##              [,1]
##  [1,]  0.01725283
##  [2,]  0.03128239
##  [3,] -0.05748994
##  [4,]  0.05893873
##  [5,]  0.94251006
##  [6,]  0.88186246
##  [7,]  0.87927929
##  [8,]  0.63780103
##  [9,]  1.00308550
## [10,]  0.75515798
## [11,]  0.80228189
## [12,]  1.05207419
## [13,]  0.82393626
## [14,]          NA
## [15,]          NA
## [16,] -0.16269298
```

In the output, we see the region has roughly a $\log 2$ raio of 1 (two-fold) relative to the background, we can interpret as a copy number of 2.

7 Genome-wide association studies (GWAS)

Genome-wide association studies (GWAS) of genotype and phenotypes finds the presence of gentic variants in many samples of high-throughput sequencing data. GWAS is a genomic analysis of gentic variants in different individuals to see *any particular variant is associated with a trait in a large population.*

There are various approaches for doing GWAS, which rely on gathering data on vaiants in particular samples and working out each sample's genotype before cross-referencing with the phenotype in some way or other. We will use the `GWAS` function from `rrBLUP` package.

The goal of GWAS, is to test for association between the frequency of each of hundreds of thousands of common variants and a given phenotype, that exceed a conservative genome-wide threshold for association and then test these for evidence of replication.

Linear regression models for GWAS can be written as follows:

$$Y = W\alpha + X\beta + g + e$$

where, for each individual, Y is a vector of phenotype values, W is a matrix of covariates including an intercept term, α is a corresponding vector of effect sizes, Xs is a vector of genotype values for all individuals at SNPs, βs is the corresponding fixed effect size of genetic variants (also known as the SNP effect size), g is a random effect that captures the polygenic effect of other SNPs, e is a random effect of residual errors.

7.1 Phenotype and genotype associations with GWAS

```
library(VariantAnnotation)
library(rrBLUP)
set.seed(1234) # for reproducibility
```

Get the VCF (Variant Call Format) file

```
vcf <- readVcf("small_sample.vcf", "hg19")
header(vcf)
```

```
## class: VCFHeader
## samples(3): NA00001 NA00002 NA00003
## meta(3): fileformat reference contig
## fixed(1): FILTER
## info(3): DP AF DB
## geno(2): GT DP
```

Extract the genotype, sample, and marker position information

```
samples <- samples(header(vcf)) # take a sample
samples
```

```
## [1] "NA00001" "NA00002" "NA00003"
```

```
chrom <- as.character(seqnames(rowRanges(vcf)))
chrom
```

```
## [1] "20" "20" "20"
```

```
pos <- as.numeric(start(rowRanges(vcf)))
pos
```

```
## [1]   14370   17330 1230237
```

```
gts <- geno(vcf)$GT
gts
```

```
##                NA00001 NA00002 NA00003
## rs6054257      "0/0"   "0/1"   "1/1"
## 20:17330_T/A   "0/0"   "0/1"   "0/0"
## 20:1230237_T/G "0/0"   "0/0"   "1/0"
```

```
markers <- rownames(gts)
```

Convert VCF genotypes into the convention used by the GWAS function:

```
convert <- function(v){
  v <- gsub("0/0", 1, v) # homozygous
  v <- gsub("0/1", 0, v) # Heterozygous
  v <- gsub("1/0", 0, v) # Heterozygous
  v <- gsub("1/1",-1, v) # Homozygous
  return(v)
}
```

Call the function and convert the result into a numeric matrix to map heterozygous and homozygous annotations to that used in GWAS.

```
gt_char<- apply(gts, convert, MARGIN = 2)

genotype_matrix <- matrix(as.numeric(gt_char), nrow(gt_char) )
colnames(genotype_matrix)<- samples
genotype_matrix
```

```
##      NA00001 NA00002 NA00003
## [1,]       1       0      -1
## [2,]       1       0       1
## [3,]       1       1       0
```

Build a dataframe describing the variant

```
variant_info <- data.frame(marker = markers, chrom = chrom, pos = pos)
variant_info
```

```
##           marker chrom     pos
## 1      rs6054257    20   14370
## 2   20:17330_T/A    20   17330
## 3 20:1230237_T/G    20 1230237
```

Build a variant/genotype dataframe

```
genotypes <-  cbind(variant_info, as.data.frame(genotype_matrix))
genotypes
```

```
##           marker chrom     pos NA00001 NA00002 NA00003
## 1      rs6054257    20   14370       1       0      -1
## 2   20:17330_T/A    20   17330       1       0       1
## 3 20:1230237_T/G    20 1230237       1       1       0
```

Build a phenotype dataframe by assigning random values

```
phenotypes <- data.frame(
  line = samples,
  score = rnorm(length(samples)) # generate random numbers as phenotypes
)
phenotypes
```

```
##      line      score
## 1 NA00001 -1.2070657
## 2 NA00002  0.2774292
## 3 NA00003  1.0844412
```

Run GWAS to map genotypes and phenotypes:

```
GWAS(phenotypes, genotypes, plot=FALSE)
```

```
## [1] "GWAS for trait: score"
## [1] "Variance components estimated. Testing markers."
```

```
##           marker chrom     pos     score
## 1      rs6054257    20   14370 0.3010543
## 2   20:17330_T/A    20   17330 0.3010057
## 3 20:1230237_T/G    20 1230237 0.1655498
```

Returns a data frame where the first three columns are the marker name, chromosome, and position, and subsequent columns are the marker scores (p-value) for the traits.

7.2 Steps in GWAS analysis

Here the GWAS analysis is carried out using the R package '`rrBLUP`

```
library("rrBLUP")
```

Load the example data. The simulated data sets used here are an example modified from wheat dataset; with 932 phenotypic yield data observations, and 329 genotypic data with 3629 markers, which are mapped.

```
load("GWAS_Data.RData") # load "pheno", "geno", and "map"
```

```
dim(pheno)
```

```
## [1] 932   3
```

```
head(pheno)### View phenotypic data.
```

```
##      GID  ENV    Yield
## 1 Oat179 Env1 6317.606
## 2 Oat130 Env2 6335.475
## 3 Oat303 Env4 7259.274
## 4 Oat270 Env1 6916.124
## 5 Oat202 Env4 6845.943
## 6 Oat233 Env3 5750.001
```

```
str(pheno) ### Display variables structure. GID and ENV needs to be factors.
```

```
## 'data.frame':    932 obs. of  3 variables:
##  $ GID  : Factor w/ 330 levels "Oat1","Oat10",..: 89 36 228 191 116 150 205 25 153 264 ...
##  $ ENV  : Factor w/ 4 levels "Env1","Env2",..: 1 2 4 1 4 3 2 2 3 2 ...
##  $ Yield: num  6318 6335 7259 6916 6846 ...
```

Genotypic data contains 329 lines and 3629 markers. The map file contains 3354 markers and three variables.

```
dim(geno)
```

```
## [1]  329 3629
```

```
geno[1:5,1:5] ### View genotypic data.
```

```
##      Marker1 Marker2 Marker3 Marker4 Marker5
## Oat1      NA       1       1       1       1
## Oat2      -1      NA       1       1       1
## Oat3      -1       1       1       1      NA
## Oat4      -1      NA      -1       1       1
## Oat5      -1      NA      -1       1       1
```

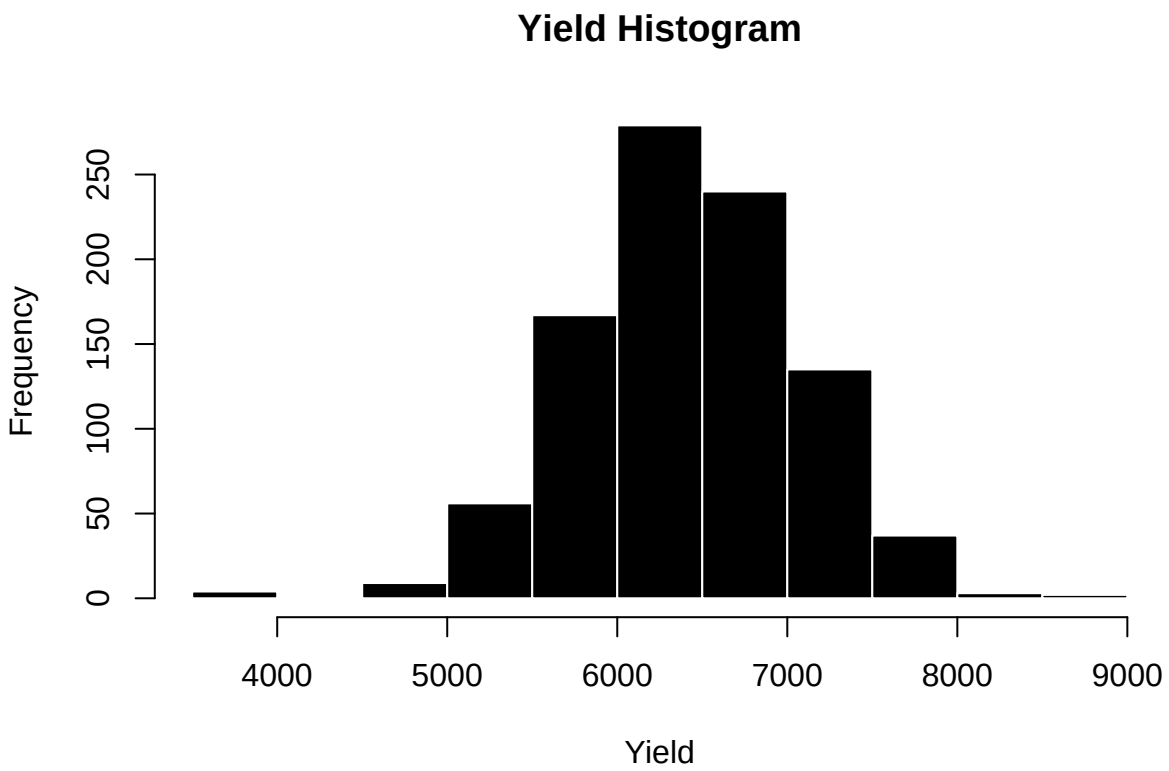
```
map[1:5,1:3] ### View Map data.
```

```
##      Markers chrom   loc
## 1  Marker607    1A 16730
## 2 Marker2900    1A 16730
## 3 Marker1316    1A 25338
## 4 Marker2297    1A 26595
## 5 Marker1895    1A 27071
```

7.2.1 Removing outliers and missing data from phenotypes

Checking normality of the data. In theory residuals needs to be checked but in general if data are normal, residuals are normal. Data seems pretty normal with some outliers.

```
hist(pheno$Yield, col="black",xlab="Yield",ylab="Frequency",
     border="white",breaks=10,main="Yield Histogram")
```



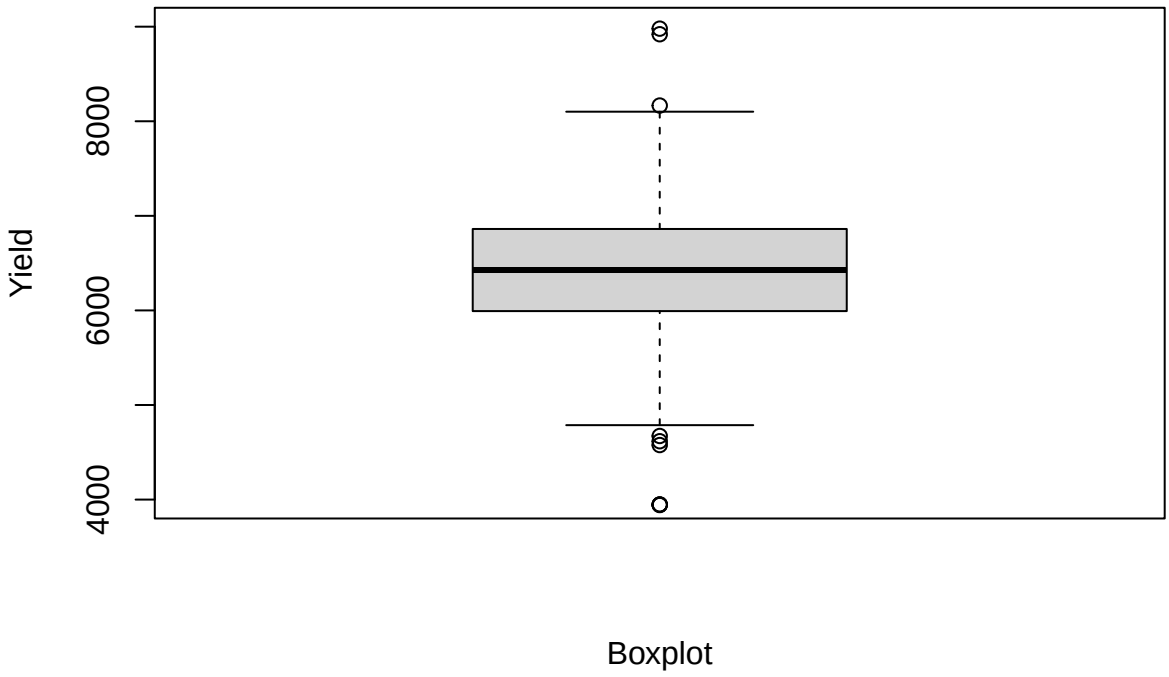
Shapiro-Wilk test indicates that normality condition is not met.

```
shapiro.test(pheno$Yield)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  pheno$Yield
## W = 0.99392, p-value = 0.0007821
```

Let´s remove the outliers and check the normality again.

```
boxplot.yield<- boxplot(pheno$Yield,xlab="Boxplot",ylab="Yield",ylim=c(4000,9000))
```




```
outliers <- boxplot.yield$out
outliers #10 outliers
```

```
##  [1] 8979.334 3944.849 3947.001 3948.641 4576.506 3942.469 8166.286 4616.835
##  [9] 4672.310 8920.055
```

```
pheno <- pheno[-which(pheno$Yield%in%outliers),] #removing outliers.
shapiro.test(pheno$Yield)# After removing outliers, Shapiro-Wilk test indicates Normality.
```

```
##
##  Shapiro-Wilk normality test
##
## data:  pheno$Yield
## W = 0.99719, p-value = 0.1093
```

```
pheno <- na.omit(pheno)## To remove all posible misiing data.
```

7.2.2 Filter data based on amount of genotypes

In this simple function, we remove individuals with missing data, markers with a certain % of missing, and heterozygous calls (IF there are a high proportion of heterozygous, it can indicate a problem with the marker, because oat is an inbreed species)

```
filter.fun <- function(geno,IM,MM,H){
  #Remove individuals with more than a % missing data
  individual.missing <- apply(geno,1,function(x){
    return(length(which(is.na(x)))/ncol(geno))
  })
```

```
  #length(which(individual.missing>0.40)) #will tell you how many
  #individulas needs to be removed with 20% missing.
  #Remove markers with % missing data
  marker.missing <- apply(geno,2,function(x)
  {return(length(which(is.na(x)))/nrow(geno))

  })
  length(which(marker.missing>0.6))

  #Remove markers herteozygous calls more than %.
  heteroz <- apply(geno,1,function(x){
    return(length(which(x==0))/length(!is.na(x)))
  })

  filter1 <- geno[which(individual.missing<IM), which(marker.missing<MM)]
  filter2 <- filter1[,(heteroz<H)]

  return(filter2)
}
geno[1:10,1:10]
```

```
##       Marker1 Marker2 Marker3 Marker4 Marker5 Marker6 Marker7 Marker8 Marker9
## Oat1       NA       1       1       1       1       1      -1      -1      -1
## Oat2       -1      NA       1       1       1       1      -1      -1      -1
## Oat3       -1       1       1       1      NA       1      -1      NA      -1
## Oat4       -1      NA      -1       1       1       1      -1      -1       1
## Oat5       -1      NA      -1       1       1      NA      -1      -1       1
## Oat6       -1       1       1      NA      -1      NA      -1       1      -1
## Oat7       NA       1      -1      -1      -1      NA       1       1      -1
## Oat8       -1      NA      -1      -1      NA      NA      NA      -1       1
## Oat9       -1       1       1      -1      -1       1      -1      -1      -1
## Oat10      -1      NA       1       1       1      NA      -1       1      -1
##       Marker10
## Oat1        NA
## Oat2         1
## Oat3         1
## Oat4        NA
## Oat5        NA
## Oat6        NA
## Oat7        NA
## Oat8        NA
## Oat9        NA
## Oat10       NA
```

```
geno.filtered <- filter.fun(geno[,1:3629],0.4,0.60,0.02)
geno.filtered[1:5,1:5];dim(geno.filtered)
```

```
##      Marker1 Marker2 Marker3 Marker4 Marker5
## Oat1      NA       1       1       1       1
## Oat2      -1      NA       1       1       1
## Oat3      -1       1       1       1      NA
## Oat4      -1      NA      -1       1       1
## Oat5      -1      NA      -1       1       1
```

```
## [1]  328 3268
```

7.2.3 Imputation of genotypes

The main idea behind imputation is to predict (or 'impute') the missing data based upon the observed data. Here, `A.mat` function from 'rrBLUP' package is used for imputation by using either means or EM algorithm.

`rrBLP` program will make imputation. For the simplicity , we impute using the mean but EM algorithm can be also used. `rrBLUP` also allows to remove markers depending on the minor allele frequency (MAF), in our example we remove those markers with MAF less than 0.05.

```
Imputation <- A.mat(geno.filtered,impute.method="EM",return.imputed=T,min.MAF=0.05)
```

```
## [1] "A.mat converging:"
## [1] 0.0431
## [1] 0.00647
```

```
K.mat <- Imputation$A ### KINSHIP matrix
K.mat[1:5,1:5] ## view Kinship
```

```
##           Oat1         Oat2       Oat3        Oat4         Oat5
## Oat1 1.8191561  0.220454719  0.1009423  0.15218203  0.177660657
## Oat2 0.2204547  2.111943356  0.1266988  0.02978199 -0.007214392
## Oat3 0.1009423  0.126698834  1.8000414 -0.12678093 -0.126589635
## Oat4 0.1521820  0.029781989 -0.1267809  1.87520005  1.803271696
## Oat5 0.1776607 -0.007214392 -0.1265896  1.80327170  1.873595678
```

```
geno.gwas <- Imputation$imputed #NEW geno data.
geno.gwas[1:5,1:5] ## view geno
```

```
##       Marker1 Marker3 Marker4   Marker5 Marker7
## Oat1 -1.44066       1       1 1.0000000      -1
## Oat2 -1.00000       1       1 1.0000000      -1
## Oat3 -1.00000       1       1 0.6753843      -1
## Oat4 -1.00000      -1       1 1.0000000      -1
## Oat5 -1.00000      -1       1 1.0000000      -1
```

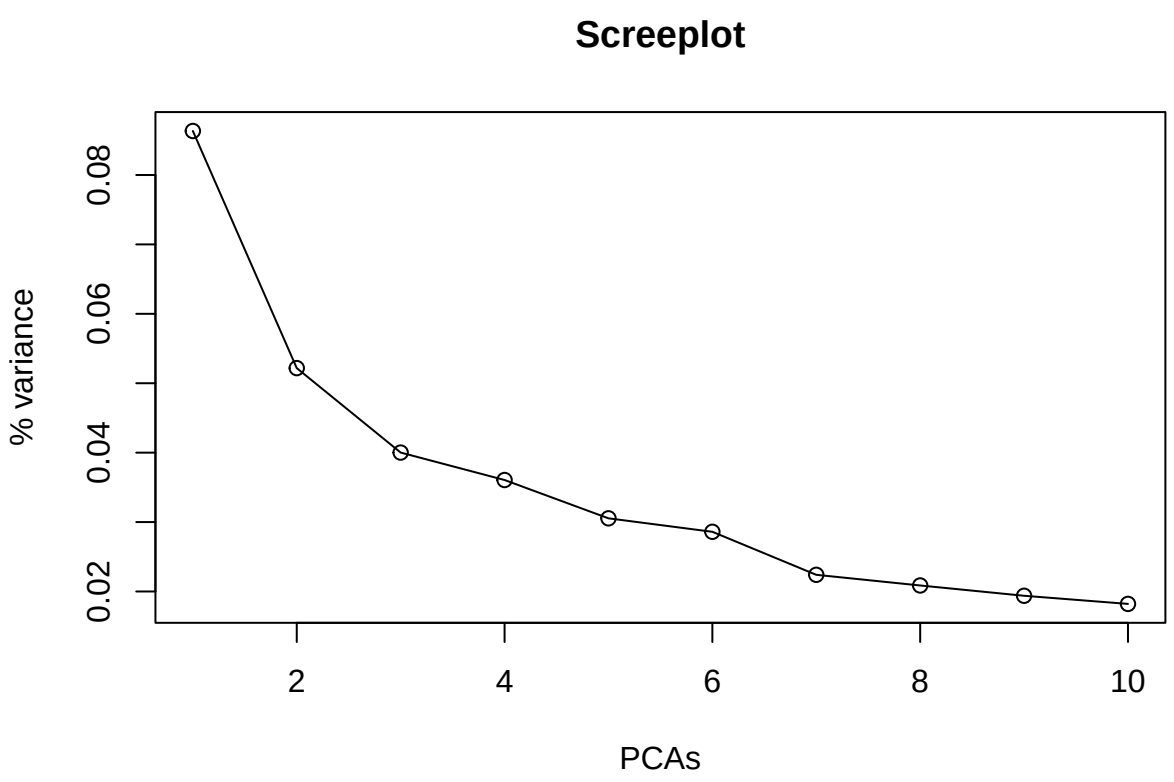
7.2.4 Checking population structure effects

One crucial step in GWAS analysis is to study the population structure. The main reason to perform this study is that, as a consequence of having different population genetic histories, distinct subpopulations could have differences in allele frequencies for many polymorphisms throughout the genome.

```
## Principal components analysis

geno.scale <- scale(geno.gwas,center=T,scale=F) # Data needs to be center.
svdgeno <- svd(geno.scale)
PCA <- geno.scale%*%svdgeno$v #Principal components
PCA[1:5,1:5]
```

```
##            [,1]       [,2]      [,3]       [,4]        [,5]
## Oat1  -6.976549  -9.824339  5.290272  -1.184019  5.00415366
## Oat2  -8.745900 -10.545057  4.730541  -8.834242 -0.02504027
## Oat3  -6.292199   3.412671  4.774038   6.657838  8.93925754
## Oat4 -19.524147   7.689386 -7.963162 -13.644082 -6.19429306
## Oat5 -19.334593   8.011715 -7.624697 -13.033030 -6.53379527
```

```
## Screeplot to visualize the proportion of variance explained by PCA
plot(round((svdgeno$d)^2/sum((svdgeno$d)^2),d=7)[1:10],type="o",main="Screeplot",
     xlab="PCAs",ylab="% variance")
```



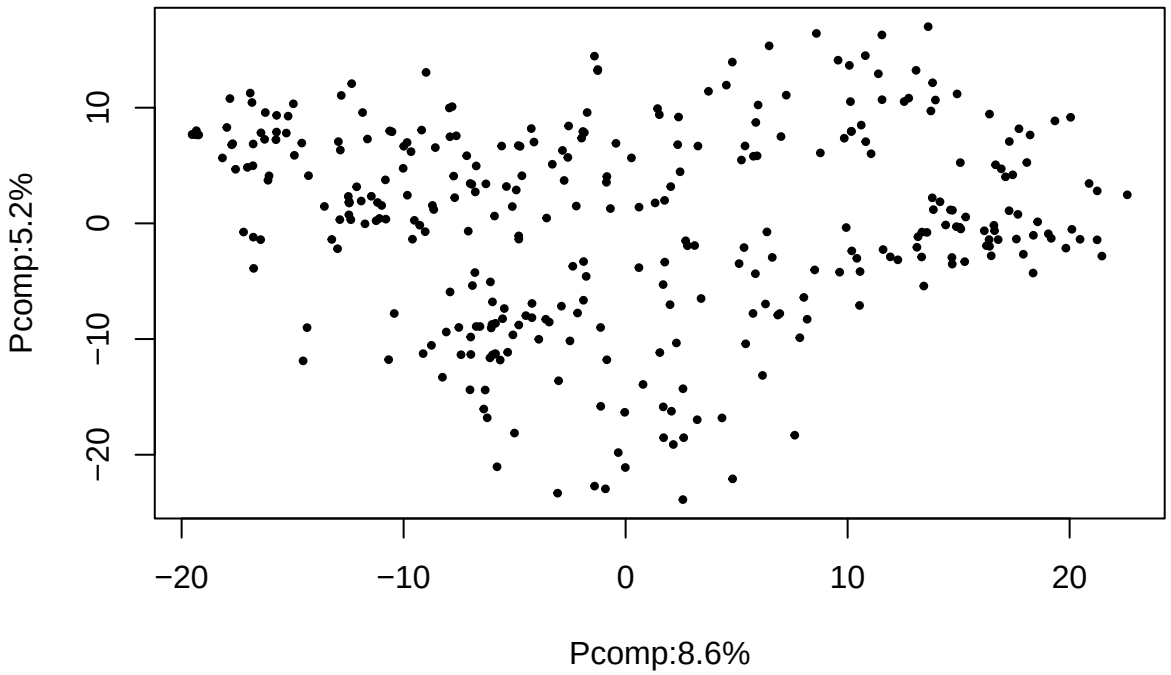

```
##Proportion of variance explained by PCA1 and PCA2
PCA1 <- 100*round((svdgeno$d[1])^2/sum((svdgeno$d)^2),d=3); PCA1
```

```
## [1] 8.6
```

```
PCA2 <- 100*round((svdgeno$d[2])^2/sum((svdgeno$d)^2),d=3); PCA2
```

```
## [1] 5.2
```

```
### Plotting Principal components.
plot(PCA[,1],PCA[,2],xlab=paste("Pcomp:",PCA1,"%",sep=""),ylab=paste("Pcomp:",PCA2,"%",sep=""),
     pch=20,cex=0.7)
```

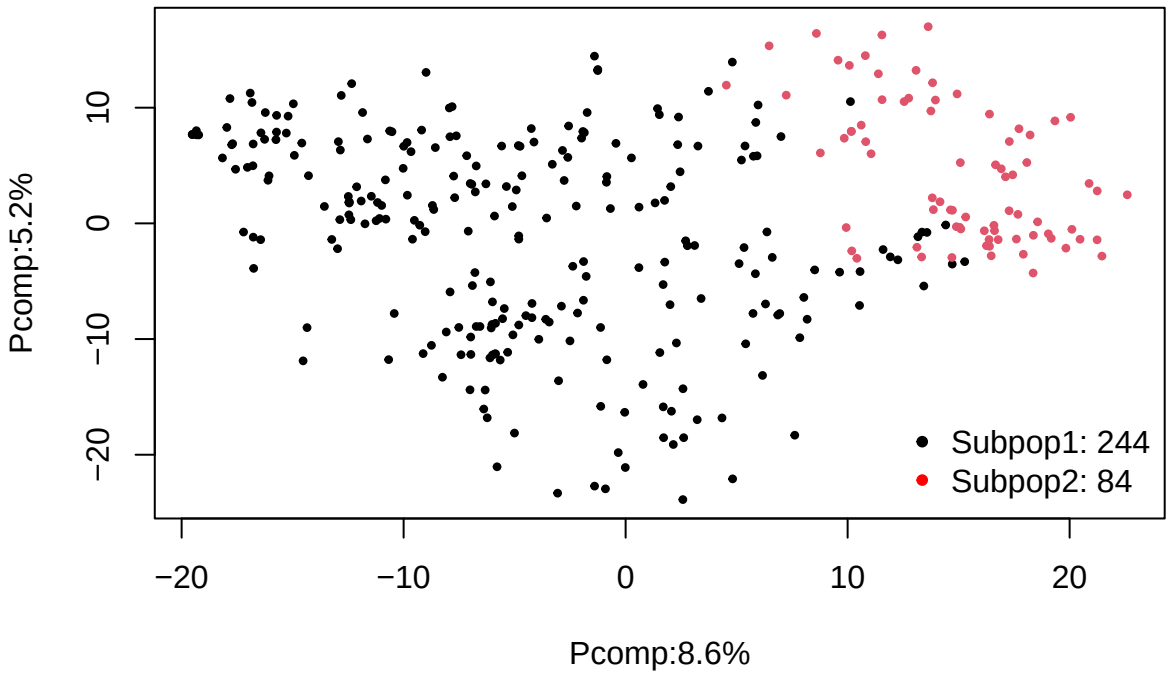


If there is a hint of polymophism, you can check with PCA after clustering

```
### Plotting depending on clustering.
Eucl <- dist(geno.gwas) ###Euclinean distance
fit <- hclust(Eucl,method="ward.D2")### Ward criterion makes clusters with same size.
groups2 <- cutree(fit,k=2) ### Selecting two clusters.
table(groups2)# Number of individuals per cluster.
```

```
## groups2
##   1   2
## 251  77
```

```
plot(PCA[,1],PCA[,2],xlab=paste("Pcomp:",PCA1,"%",sep=""),ylab=paste("Pcomp:",PCA2,"%",sep=""),
     pch=20,cex=0.7,col=groups2)
legend("bottomright",c("Subpop1: 244","Subpop2: 84"),pch=20,col=(c("black","red")),
       lty=0,bty="n",cex=1)
```



7.2.5 Matching phenotype and genotype

The following code prepares the files to be used on GWAS function. In order to do the analysis the same genotypes needs to be on phenotype and genotypes files. A matching function between files needs to be performed.

```
pheno=pheno[pheno$GID%in%rownames(geno.gwas),]
pheno$GID<-factor(as.character(pheno$GID), levels=rownames(geno.gwas)) #to assure same levels on both f
pheno <- pheno[order(pheno$GID),]

##Creating file for GWAS function from rrBLUP package
X<-model.matrix(~-1+ENV, data=pheno)
pheno.gwas <- data.frame(GID=pheno$GID, X, Yield=pheno$Yield)
head(pheno.gwas)
```

```
##      GID ENVEnv1 ENVEnv2 ENVEnv3 ENVEnv4    Yield
## 77  Oat1       0       0       1       0 6126.514
## 81  Oat1       1       0       0       0 6613.542
## 330 Oat1       0       0       0       1 5984.449
## 417 Oat1       0       1       0       0 6855.848
## 441 Oat2       0       1       0       0 5510.549
## 442 Oat2       1       0       0       0 5711.477
```

```
# after imputation, we have to align genotype and phenotype data again
geno.gwas <- geno.gwas[rownames(geno.gwas)%in%pheno.gwas$GID,]
pheno.gwas <- pheno.gwas[pheno.gwas$GID%in%rownames(geno.gwas),]
geno.gwas <- geno.gwas[rownames(geno.gwas)%in%rownames(K.mat),]
K.mat <- K.mat[rownames(K.mat)%in%rownames(geno.gwas),colnames(K.mat)%in%rownames(geno.gwas)]
pheno.gwas <- pheno.gwas[pheno.gwas$GID%in%rownames(K.mat),]
```

```
# we need to do the same for 'map' data
geno.gwas<-geno.gwas[,match(map$Markers,colnames(geno.gwas))]
geno.gwas <- geno.gwas[,colnames(geno.gwas)%in%map$Markers]
map <- map[map$Markers%in%colnames(geno.gwas),]
head(map)
```

```
##       Markers chrom   loc
## 4  Marker2297    1A 26595
## 9  Marker3125    1A 35232
## 10 Marker2100    1A 35653
## 13 Marker1797    1A 51943
## 14 Marker3191    1A 51943
## 16 Marker1403    1A 56393
```

```
geno.gwas2<- data.frame(mark=colnames(geno.gwas),chr=map$chrom,loc=map$loc,t(geno.gwas))
dim(geno.gwas2)
```

```
## [1] 1759  331
```

```
colnames(geno.gwas2)[4:ncol(geno.gwas2)] <- rownames(geno.gwas)

head(pheno.gwas)
```

```
##      GID ENVEnv1 ENVEnv2 ENVEnv3 ENVEnv4    Yield
## 77  Oat1       0       0       1       0 6126.514
## 81  Oat1       1       0       0       0 6613.542
## 330 Oat1       0       0       0       1 5984.449
## 417 Oat1       0       1       0       0 6855.848
## 441 Oat2       0       1       0       0 5510.549
## 442 Oat2       1       0       0       0 5711.477
```

```
geno.gwas2[1:6,1:6]
```

```
##                  mark chr   loc Oat1       Oat2 Oat3
## Marker2297 Marker2297  1A 26595   -1 -1.0000000   -1
## Marker3125 Marker3125  1A 35232    1 -1.0000000   -1
## Marker2100 Marker2100  1A 35653   -1  1.0000000    1
## Marker1797 Marker1797  1A 51943    1 -0.4259651   -1
## Marker3191 Marker3191  1A 51943   -1  1.0000000    1
## Marker1403 Marker1403  1A 56393   -1  1.0000000    1
```

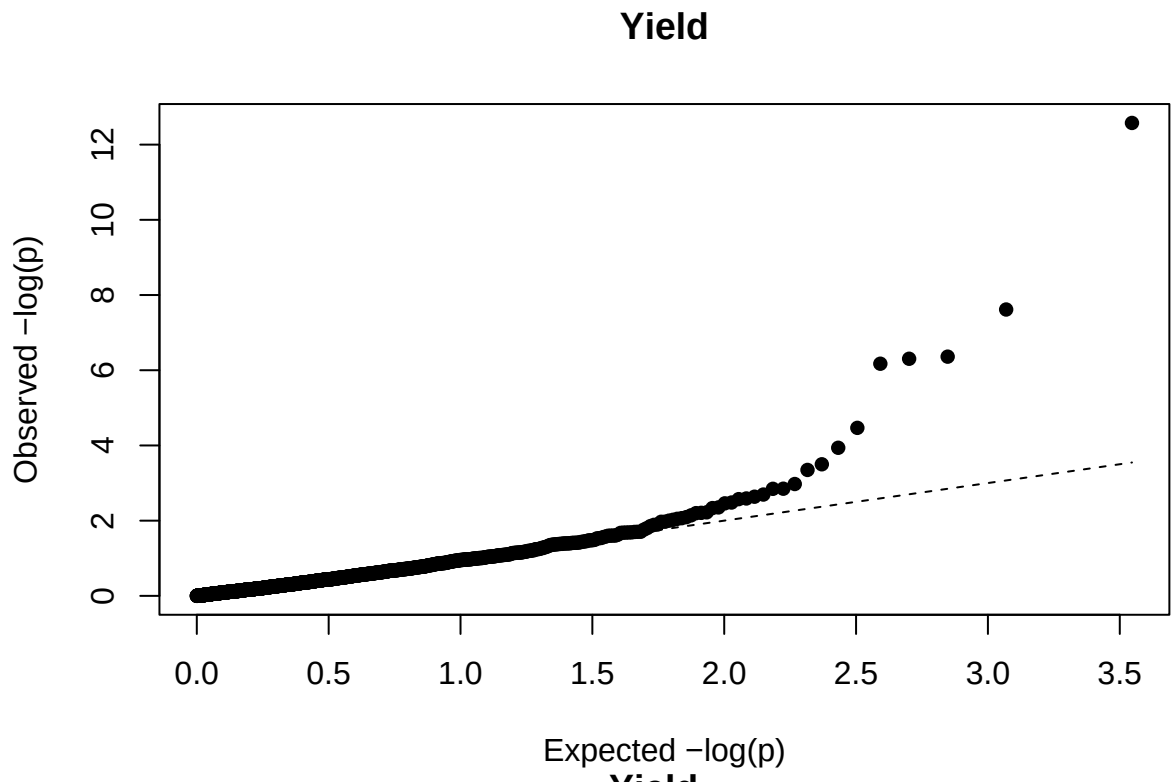
A statistically significant association between a genotypic marker and a particular trait is considered to be a proof of linkage between the phenotype and a casual locus. Generally, population structure (PS) leads to spurious associations between markers and a trait, so that a statistical approach must account for PS

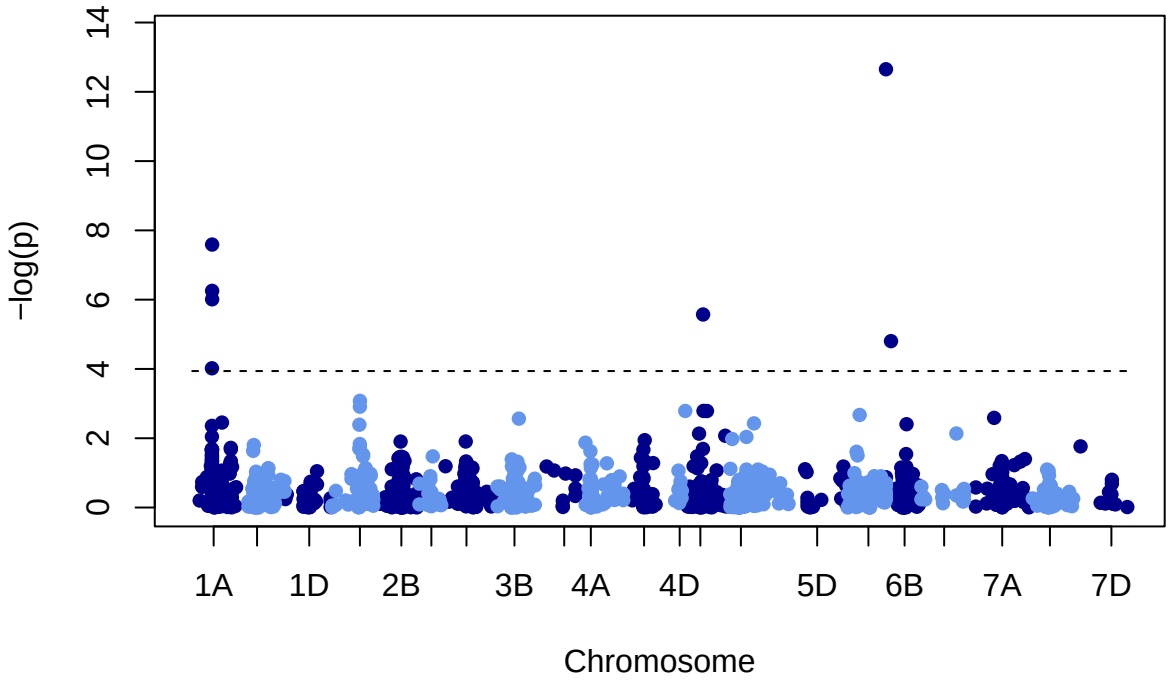
```
gwasresults<-GWAS(pheno.gwas,geno.gwas2, fixed=colnames(pheno.gwas)[2:5], K=NULL, plot=T,n.PC=0)
```

```
## [1] "GWAS for trait: Yield"
## [1] "Variance components estimated. Testing markers."
```

```
gwasresults2<-GWAS(pheno.gwas,geno.gwas2, fixed=colnames(pheno.gwas)[2:5], K=NULL, plot=T,n.PC=6)
```

```
## [1] "GWAS for trait: Yield"
## [1] "Variance components estimated. Testing markers."
```



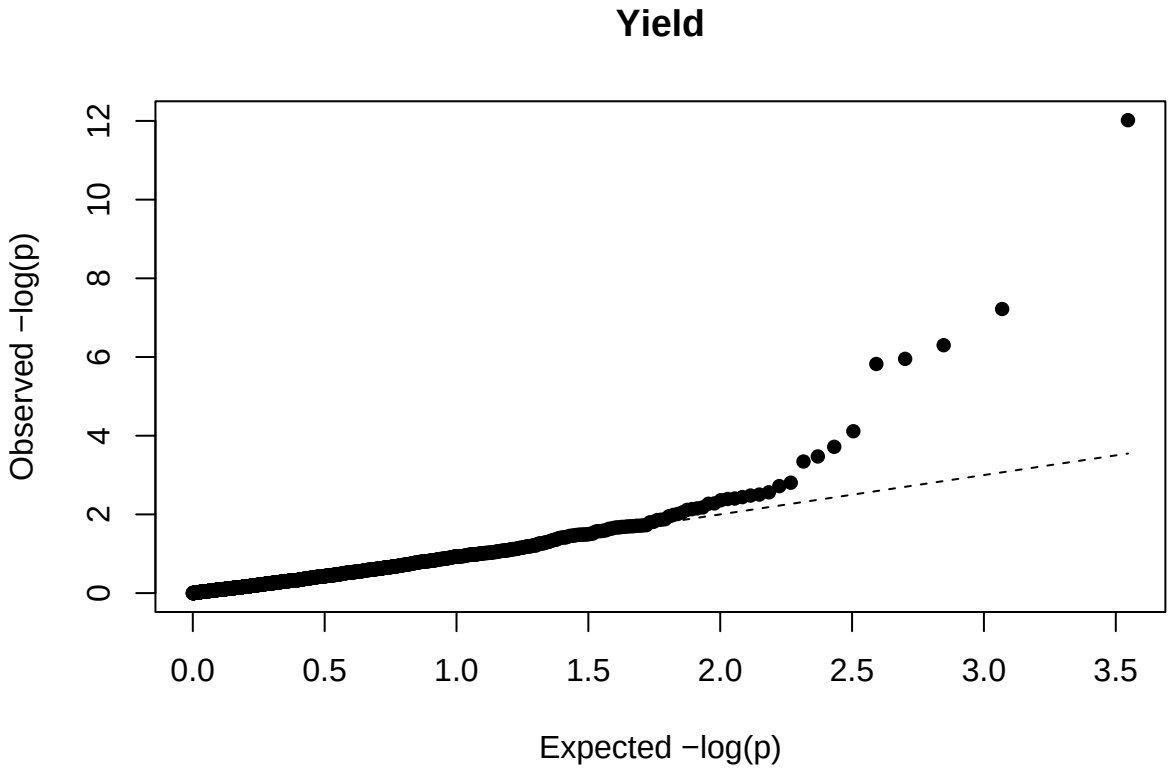


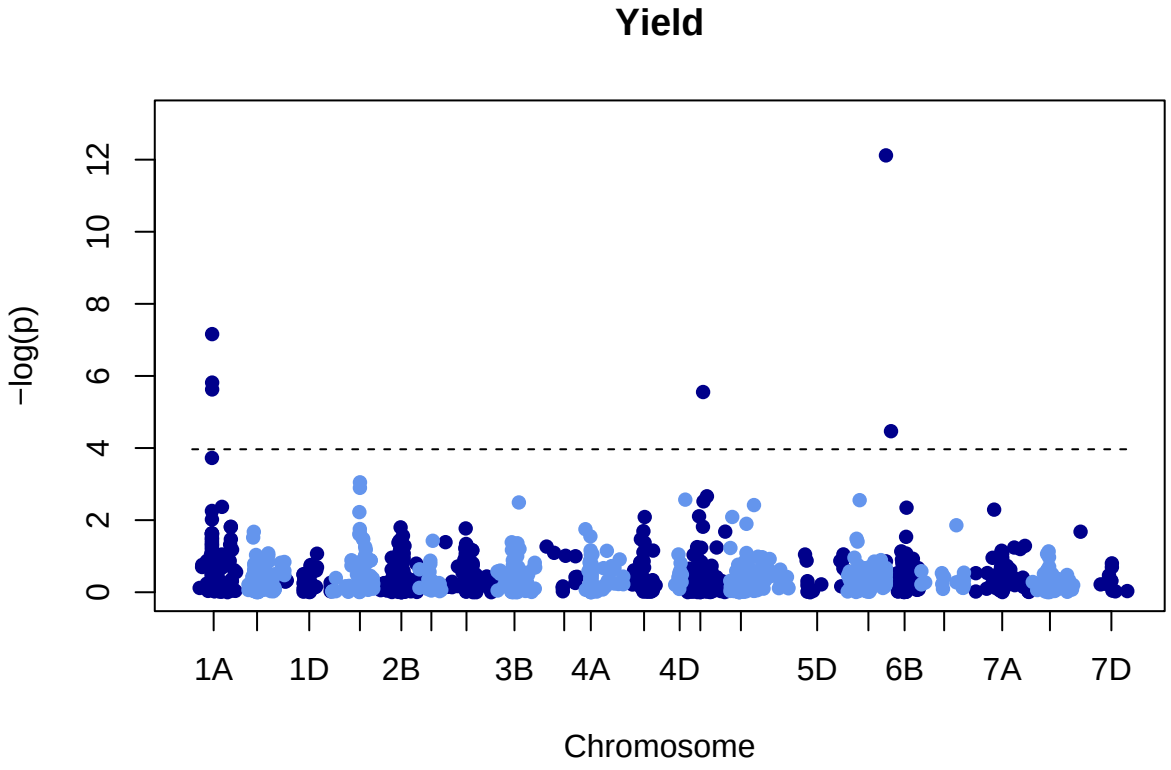
```
gwasresults3<-GWAS(pheno.gwas,geno.gwas2, fixed=colnames(pheno.gwas)[2:5], K=K.mat, plot=T,n.PC=0)
```

```
## [1] "GWAS for trait: Yield"
## [1] "Variance components estimated. Testing markers."
```

```
gwasresults4<-GWAS(pheno.gwas,geno.gwas2, fixed=colnames(pheno.gwas)[2:5], K=K.mat, plot=T,n.PC = 6)
```

```
## [1] "GWAS for trait: Yield"
## [1] "Variance components estimated. Testing markers."
```






```
#The option plot=T will produce manhattan plots and q-q plots. With the aim to provide
#another option in R, another graphs in R have been provided.
```

```
str(gwasresults)
```

```
## 'data.frame':    1759 obs. of  4 variables:
##  $ mark : chr  "Marker2297" "Marker3125" "Marker2100" "Marker1797" ...
##  $ chr  : chr  "1A" "1A" "1A" "1A" ...
##  $ loc  : int  26595 35232 35653 51943 51943 56393 66313 68829 69872 70102 ...
##  $ Yield: num  0.325 0.451 0.588 0.187 0.861 ...
```

```
#Let´s see the structure
#First 3 columns are just the information from markers and map.
#Fouth and next columns are the results form GWAS. Those values are already
#the  -log10 pvalues, so no more transformation needs to be done to plot them.
```

7.2.6 False Discovery Rate Function

```
FDR<-function(pvals, FDR){
    pvalss<-sort(pvals, decreasing=F)
    m=length(pvalss)
    cutoffs<-((1:m)/m)*FDR
    logicvec<-pvalss<=cutoffs
    postrue<-which(logicvec)
    print(postrue)
    k<-max(c(postrue,0))
```

```
    cutoff<-(((0:m)/m)*FDR)[k+1]
    return(cutoff)
}

alpha_bonferroni =-log10(0.05/length(gwasresults$Yield)) ###This is Bonferroni correcton
alpha_FDR_Yield <- -log10(FDR(10^(-gwasresults$Yield),0.05))## This is FDR cut off
```

```
## [1] 1 2 3 4 5 6 7
```

Match with hits

Species that are highly correlated with the yeild:

```
which(gwasresults$Yield>alpha_bonferroni)
```

```
## [1]   53   56   57 1054 1427
```

```
which(gwasresults$Yield>alpha_FDR_Yield)
```

```
## [1]   45   53   56   57 1054 1427 1428
```

```
which(gwasresults2$Yield>alpha_bonferroni)
```

```
## [1]   53   56   57 1054 1427 1428
```

```
which(gwasresults2$Yield>alpha_FDR_Yield)
```

```
## [1]   45   53   56   57 1054 1427 1428
```

```
which(gwasresults3$Yield>alpha_bonferroni)
```

```
## [1]   53   56   57 1054 1427
```

```
which(gwasresults3$Yield>alpha_FDR_Yield)
```

```
## [1]   45   53   56   57 1054 1427 1428
```

```
which(gwasresults4$Yield>alpha_bonferroni)
```

```
## [1]   53   56   57 1054 1427
```

```
which(gwasresults4$Yield>alpha_FDR_Yield)
```

```
## [1]   45   53   56   57 1054 1427 1428
```

8 References:

- R Bioinformatics Cookbook by Dan MacLean:
https://github.com/PacktPublishing/R-Bioinformatics-Cookbook

- Next Generation Sequencing And Data Analysis edited by Melanie Kappelmann-Fenzl:
https://link.springer.com/book/10.1007/978-3-030-62490-3

- Genome-wide Association Anaysis using R https://link.springer.com/protocol/10.1007/978-1-4939-6682-0_14