Introduction to Biocondcutor tools for second-generation sequencing analysis

Héctor Corrada Bravo
based on slides developed by
James Bullard, Kasper Hansen and Margaret Taub

PASI, Guanajuato, México
May 3-4, 2010

Historical background: previous sequencing-based expression measurements

- Serial analysis of gene expression (SAGE) has been around for a while (Velculescu, et al., 1995), along with other methods such as massively parallel signature sequencing (MPSS) and others.
- SAGE data comes from selecting small "tags", one from each mRNA fragment, then concatenating them and sequencing this long series of sequence fragments using Sanger sequencing. Expression of a gene is measured by counting how many times the tag from that gene appears in the sequence.
- In general, the total number of tags sequenced per experiment is relatively small (tens of thousands).
- Many statistical methods were developed in order to account for biases in species detection, facilitate combining reads across libraries, and measure differential expression from the observed counts - more on this later.

Next Generation Sequencing: NGS

- Illumina (Solexa), "Genome Analyzer I and II"
- Roche (454), "Genome Sequencer"
- Applied Biosystems, "SOLiD"
- Helicos, "Heliscope"

There are many articles/websites comparing the different technologies, however the most prevalent platform appears to be Illumina at this point.
Publications: `http://www.illumina.com/pagesnrn.ilmn?ID=93`

Applications of sequencing technology

High throughput sequencing has been used in many applications:

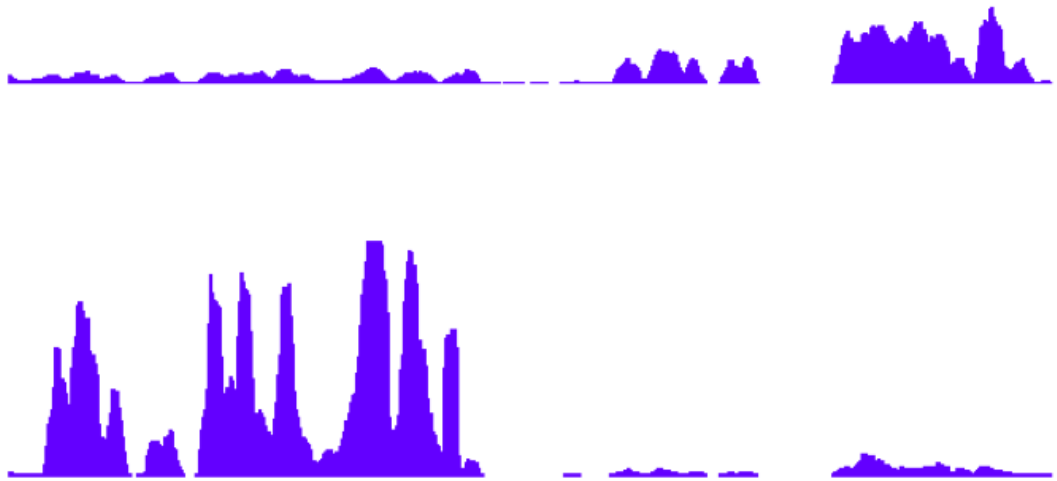
- Resequencing
- De novo genome sequencing
- Tag-based expression analysis
- RNA-Seq expression analysis
- ChIP-Seq to detect transcription factor binding sites
- Methyl-Seq to detect DNA methylation sites
- Small RNA sequencing
- Ribosome profiling
- CLIP-Seq to detect protein-RNA interactions

Steps in analysis of sequencing data

The next lectures will walk through the steps needed to go from the output of the sequencing machine to abundance measurements for regions of interest. We will focus on tools from Bioconductor, although others will be used as well. We will cover:

- Alignment
- Working with annotations (regions of interest)
- Aggregating counts to get a measure of abundance

A first quick look at the data



Data from two lanes of *D. melanogaster*, one wild-type and one where a splicing factor has been knocked down.

(Data courtesy of Brenton Graveley.)

Overview of Bioconductor NGS Packages

- IRanges : provides low-level containers used by many of the other Bioconductor packages; useful for working with annotation
- Biostrings : provides complicated string processing methods, emphasis is placed on speed, additionally provides a short read alignment tool: matchPDict; useful for mapping
- ShortRead : provides functionality for processing NGS experiments, namely reading in various aligned formats as well as lower-level Illumina experiments; useful for file I/O, mapping
- BSgenome : provides tools for the ability to deal with genomes in R, typically one uses a particular genome package, such as: BSgenome.Scerevisiae.UCSC.sacCer1; useful for mapping
- GenomeGraphs : provides plotting functionality attached to live annotation from Ensembl; useful for visualization, data quality assessment, results analysis
- biomaRt : provides programmatic access to Ensembl and other biomart databases; useful for working with annotation
- chipSeq : provides tools for dealing with Chip-seq experiments

Overview of Bioconductor NGS Packages

- RTrackLayer : provides access to various genome browsers; useful for examining results

Mapping

These slides will discuss mapping of sequence data as well as the Biostrings and ShortRead packages.

- Alignment and tools (mostly external to R).
- Biostrings overview.
- Alignment tools in Biostrings.
- Mapping data and reading it into R.

Alignment input: FASTQ files

FASTQ files represent a common "end-point" from the various sequencing platforms (i.e. NCBI short read archive, `http://www.ncbi.nlm.nih.gov/Traces/sra/sra.cgi`). Qualities are encoded in ASCII and depending on the platform have slightly different meanings/ranges and encoding. More details can be found at: `http://en.wikipedia.org/wiki/FASTQ_format`

```
@GA-EAS46_2_209DG:6:1:890:752
TTCTCTTAAGTCTTCTAGTTCTCTTCTTTCTCCACT
+GA-EAS46_2_209DG:6:1:890:752
hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
@GA-EAS46_2_209DG:6:1:905:558
TCTGGCTTAACTTCTTCTTTTTTTTCTTCTTCTTCT
+GA-EAS46_2_209DG:6:1:905:558
hhhfhhhhhhhhhhhhhhhhhhhhhhehhGhhJh
```

Alignment Tools

The number of short read aligners have exploded, but a couple tools have emerged as the *de facto* standards.

- Eland: Illumina's aligner, quality aware, fast, paired end capable
- MAQ: Good SNP caller, quality aware, paired end capable
- Bowtie: Super fast, offers different alignment strategies, paired end capable
- BWA: Fast, indel support, paired ends, qualities
- NovoAlign: MAQ like speed, many features.

A superior overview of the different aligners is available at: `http://www.sanger.ac.uk/Users/lh3/NGSalign.shtml` Additionally, a comparison of two of the best aligners can be found here: `http://www.massgenomics.org/2009/07/maq-bwa-and-bowtie-compared.html`

Common Alignment Strategies

- Use qualities (default for Bowtie and MAQ)
- Perfect match, no repeats (Strict / Lenient)
- Mismatches, no repeats
- Paired end data (PET) (harder for RNA-Seq)

The standard Illumina protocol yields unstranded reads.

Most aligners are evaluated in terms of "how many reads are mapped". Is this the right objective?

Watch out for the output of the program; there are many different conventions (0-based, what happens to read hitting the reverse strand, etc.)

SAM/BAM Formats

SAM (BAM) is a new general format for storing mapped reads. It is developed as part of the 1000 Genomes project and is quickly becoming a kind of standard. Some alignment tools output this format directly, otherwise there are scripts in `samtools` for doing it for most popular aligners. Details at `http://samtools.sourceforge.net/index.shtml`

A comment

There are no great comprehensive tools for analyzing deep-sequence data.
It will involve a fair amount of coding and gluing together various tools.

We will introduce some tools from Bioconductor that can be useful.

I use a lot of shell scripting.

Biostrings overview

- A package for working with large (biological) strings.
- Two main types of objects: A really long single string (think chromosome) or a set of shorter strings (think reads or genes). BString vs. BStringSet.
- These classes are implemented *efficiently* minimizing copy and memory loading/unloading.
- The BSgenome contains some infrastructure for whole genomes.
- Methods for dealing with biological data, including basic manipulation (complementation, translation, etc.), string searching (exact/inexact matching, Smith-Waterman, PWMs).
- Fairly complicated class structure.

> *I'm sorry to say that, at least for me, this has become hopelessly confusing, and I imagine that many other users feel the same. – Simon Anders on bioc-sig-sequencing*

Computing on genomes is not trivial. The approach to a given computation is important.

Strings in Biostrings

- BString (general), DNAString, RNAString AAString (Amino Acid), all examples of XString.
- complement, reverse, reverseComplement, translate, for the classes where "it makes sense".
- Convert to and from a standard R character string.
- Constructor: `DNAString("ACGGGGG")`.
- Support for IUPAC alphabet.
- Subsetting subseq, using the SEW format (two out of the three start/end/width). Efficient.
- StringSets are collections of Strings, like DNAStringSet.

BSgenomes

As examples, we will use whole genomes. Use available.genomes to get available genomes. Long package names, but always a shorter object name.

```
> library(BSgenome.Scerevisiae.UCSC.sacCer1)
> Scerevisiae
> Scerevisiae[[1]]
> Scerevisiae[["chr1"]]
```

A BSgenome may also have *masks*. We will ignore this for now.

Biostrings, more

- alphabetFrequency, oligonucleotideFrequency and others.

```
> alphabetFrequency(Scerevisiae[["chr1"]])
> oligonucleotideFrequency(Scerevisiae[["chr1"]],
+     width = 3)
```

- chartr for character translation ("make all As into Cs").
- Various IO functions (also ShortRead).

Views

A view is a set of substrings of an XString, stored and manipulated very efficiently. Example: to store exon sequences, one can just store the genomic location of the exons.

```
> Views(Scerevisiae[["chr1"]], start = c(300,
+     400, 500), width = 50)
```

A view is associated with a *subject*. Internally, they are essentially IRanges, so they are very fast to compute with. Views can in many cases be treated exactly as other strings. There are methods such as narrow, trim, gaps, restrict.

Matching in Biostrings

There are various ways of *matching* or *aligning* strings to each other. We use matching to denote searching for an exact match or possibly a match with a certain number of mismatches. Alignment denotes a more general strategy, e.g. Smith-Waterman.

- matchPattern / countPattern: match 1 sequence to 1 sequence.
- vmatchPattern / vcountPattern: match 1 sequence to many sequences.
- pairwiseAlignment: align many sequences to 1 sequence.
- matchPDict / countPDict: match many sequences to 1 sequence. ("dict" indicates that the many sequences are preprocessed into a dictionary).
- matchPWM, trimLRpattern

The different functions are optimized for different situations. They also use different algorithms, which has a big impact. Especially pairwiseAlignment is flexible and therefore has a complicated syntax.

Example: mapping probes to a genome

Get a list of Scerevisiae probes from the "yeast2" Affymetrix array.

```
> library(yeast2probe)
> ids <- scan("s_pombe.msk", skip = 2,
+     what = list(probeset = character(0),
+         junk = character(0)))$probeset
> probes <- yeast2probe$sequence[yeast2probe$Probe.Set.Name %in%
+     ids]
> probes <- DNAStringSet(probes)
```

Mapping

```
> require(BSgenome.Scerevisiae.UCSC.sacCer1)
> dict0 <- PDict(probes)
> dict0.r <- PDict(reverseComplement(probes))
> hits <- matchPDict(dict0, Scerevisiae[[1]])
> table(countIndex(hits))
> table(countPDict(dict0, Scerevisiae[[1]]))
> allhits <- countPDict(dict0, Scerevisiae[[1]]) +
+     countPDict(dict0.r, Scerevisiae[[1]])
> table(allhits)
```

ShortRead

ShortRead contains tools for

- Work with Illumina output.
- Generate QA report on Illumina files.
- Read a variety of short read data formats.

Data

We will use data from (Lee, Hansen, Bullard, Dudoit, and Sherlock (2008) PLoS Genetics).

We are considering a wild-type and a mutant strain of yeast. The two strains were sequenced using an Illumina Genome Analyzer.

We are using a subset of the data: 100K reads from a single lane for one strain.

Reading the unaligned data

```
> require(ShortRead)
> fq <- readFastq(".", "mut_1_f.fastq.gz$")
> fq
```

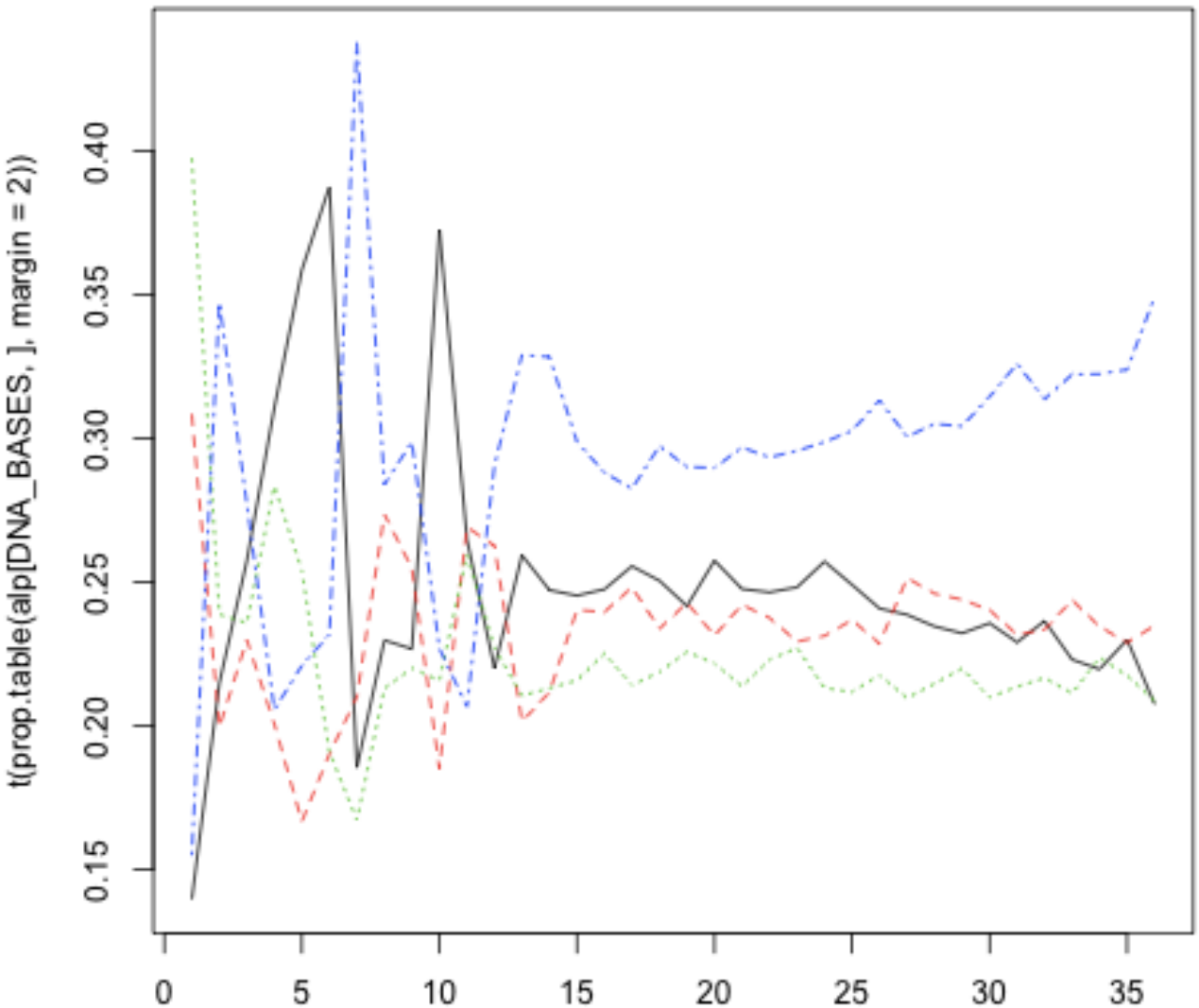
There are various simple accessor functions for this object, especially sread and quality.

A brief look at the unaligned data

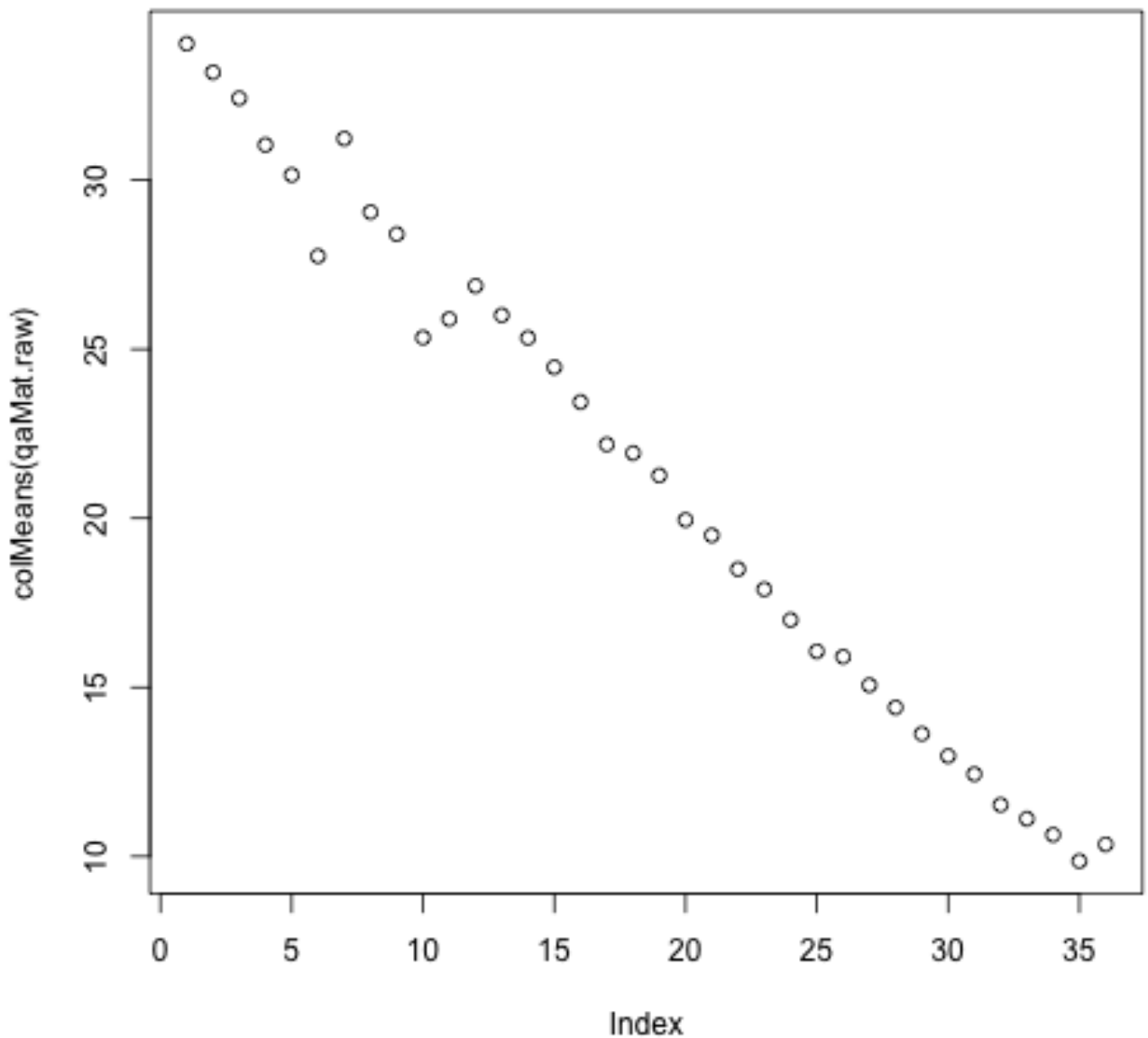
Using alphabetByCycle and as(, "matrix") (which creates a big read times cycle matrix), we can do

```
> alp <- alphabetByCycle(sread(fq))
> matplot(t(prop.table(alp[DNA_BASES,
+     ], margin = 2)), type = "l")
> qaMat.raw <- as(quality(fq), "matrix")
> plot(colMeans(qaMat.raw))
```

Alphabet By Cycle plot



Mean Quality per Cycle



Reading in the aligned data

We read data aligned with `bowtie` into `R` as a AlignedRead object

```
> aligned <- readAligned(".", pattern = "\\.bowtie.gz$",
+     type = "Bowtie")
> aligned
```

Getting abundance data for regions of interest

- We want to get our data in a form where we can measure abundance for regions of interest (e.g. transcript abundance in genes).
- We have already mapped our reads to our reference genome, so we know what position they originated from.
- Next, we need to work with the genome annotation and produce region-level summaries.
- We start by importing the annotation into R so that we can work with it there.

Working with annotation

We need to obtain annotation for our organism. Using Bioconductor this can be done with the biomaRt package. This package allows us to directly download information from Ensembl using the biomaRt interface.

```
> require(biomaRt)
> mart <- useMart("ensembl", "scerevisiae_gene_ensembl")
> yeastAnno <- getBM(c("ensembl_gene_id",
+     "chromosome_name", "start_position",
+     "end_position", "strand", "gene_biotype"),
+     mart = mart)

> head(yeastAnno, 2)
```

Data Representation

Currently, we have the data stored where each line in our data file represents a unique read. Often we want to count the number of reads that overlap or start at a given position. One easy way to do this is to use the pileup function from ShortRead.

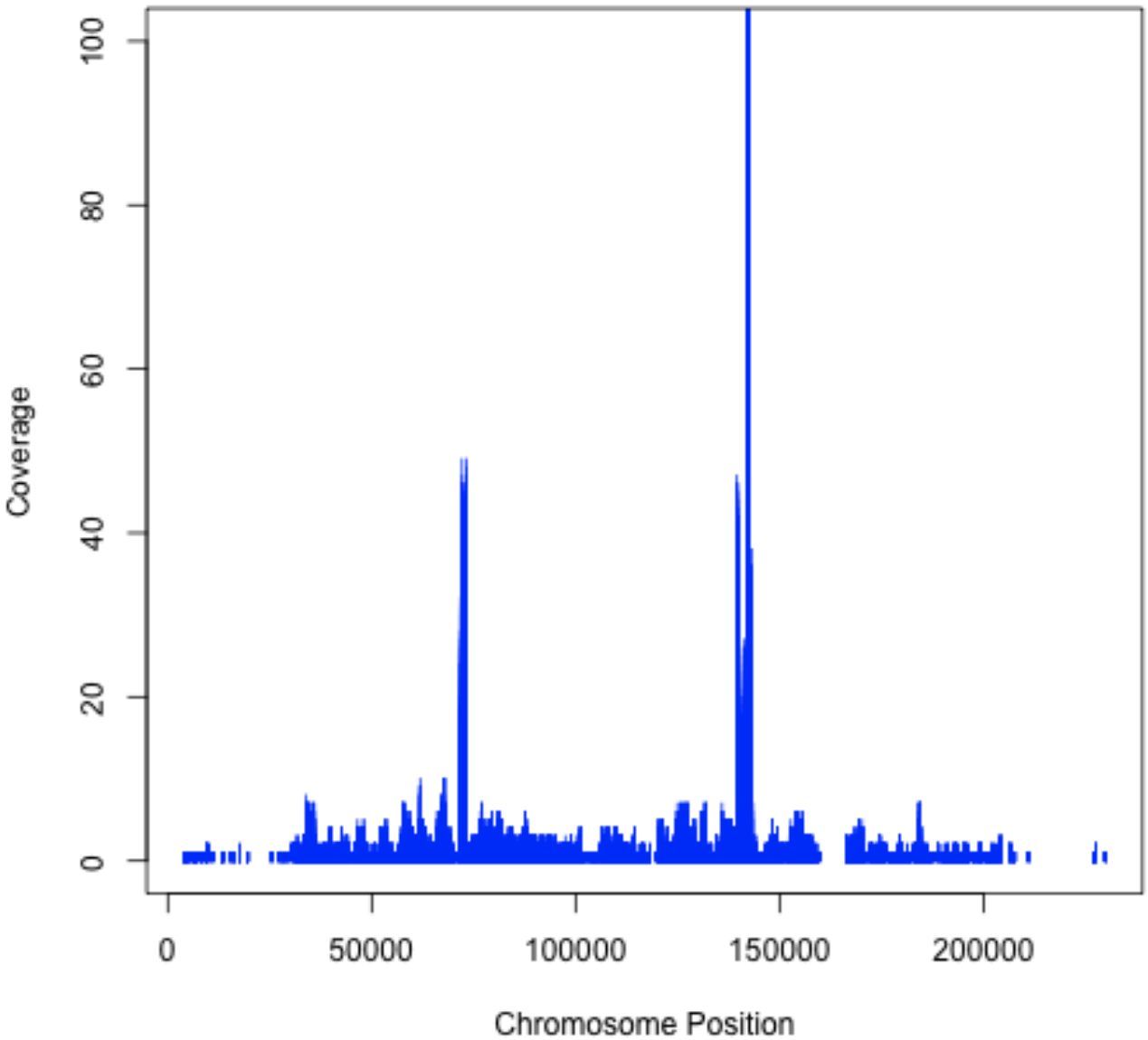
```
> keep <- chromosome(aligned) ==
+     "Scchr01"
> ll <- seqlengths(Scerevisiae)
> names(ll) <- levels(chromosome(aligned))
> cov <- coverage(aligned[keep],
+     width = ll)
```

- What type of object is returned?

Plotting coverage

```
> cov <- as.numeric(cov)
> keep <- cov > 0
> Indices <- (1:length(cov))[keep]
> plot(Indices, cov[keep], col = "blue",
+     type = "h", ylim = c(0, 100),
+     xlab = "Chromosome Position",
+     ylab = "Coverage")
```

Plotting Coverage



Associating Annotation to Reads

At an abstract level, we want to apply a function that associates the aligned read data to the relevant regions of annotation and then counts the number of reads that fall into each gene. This will be our measure of gene expression.

First, we need to determine in which range each read lands. At the end of the day, we want a data structure that looks more or less like this:

	mut_1	mut_2	wt_1	wt_2
YHR055C	0	0	0	0
YPR161C	38	39	35	34
YOL138C	31	33	40	26
YDR395W	55	52	47	47
YGR129W	29	26	5	5
YPR165W	189	180	151	180

IRanges

- An IRange is a set of integer ranges: $\{1-4, 7-10, 45-89\}$. The ranges can be overlapping or not, and are stored as (start, end, width).
- Every range can be specified using 2 out of (start, end, width).
- This has proved to be a very general class of objects. Examples: gene (annotation in general), DNA reads.
- A NormalIRanges is a minimal representation (in some sense): not empty, non-overlapping, non-empty gap, ordered. They are important for computational reasons.
- Many methods for IRanges are very fast. Some computations can be sped up dramatically by converting the underlying objects into IRanges, do the computation and then convert back.

Some examples of IRanges functions

In our case, the integer ranges we are interested in are the base positions between the start and end position of the yeast genes. But the package is much more general and can be used for other kinds of analysis as well. As a simple example:

```
> ir1 <- IRanges(start = 1:5, end = 6:10)
> ir2 <- IRanges(start = 4:12, width = 5)
> union(ir1, ir2)
> intersect(ir1, ir2)
> findOverlaps(ir1, ir2)
> as.table(findOverlaps(ir1, ir2))
```

Some more details on overlap

The function findOverlaps can be confusing at first. The syntax is `findOverlaps(query,subject)`, and you can think of the result as a binary matrix, with `length(query)` rows and `length(subject)` columns. The return value you see is simply a "sparse" representation of this matrix, giving only the coordinates of the non-zero elements.

Additionally, overlap can be called on two `RangesList` objects of the same length. The resulting object is a list of objects of class `RangesMatching`.

Using IRanges to represent our annotation

First, we need to match the chromosome names as Ensembl uses roman numerals and the Bowtie index used names.

```
> cNames <- levels(chromosome(aligned[[1]]))
> names(cNames) <- c(as.character(as.roman(1:16)),
+     NA)
> yeastAnno$chr <- cNames[yeastAnno$chr]
> yeastAnno <- yeastAnno[!is.na(yeastAnno$chr),
+     ]
```

Now, we construct a set of IRanges which represent our genes. Since the IRanges class only includes a start, a stop and a width, we need one IRanges object for each chromosome. In the last step, we construct a RangesList object, which will be useful for our next steps.

Using IRanges to represent our annotation

```
> annoByChr <- split(yeastAnno, yeastAnno$chr)
> iL <- lapply(annoByChr, function(d) {
+     IRanges(start = d$start_position,
+         end = d$end_position)
+ })
> iL <- do.call(RangesList, iL)
> iL
```

Representing our reads as IRanges

Next, we convert our aligned reads into IRanges as well. Again, we need a separate IRanges object for each chromosome, and they are arranged into a RangesList.

```
> alnAsRanges <- as(aligned, "RangesList")
> alnAsRanges
```

Counting reads that fall in our genes

Finally, we use the as.table and countOverlaps in order to compute the counts within each gene (removing reads from the mitochondrial genome). Unfortunately, this operation doesn't yet do the right thing with names, so we have to add the names back at the end; not elegant.

```
> alnAsRanges <- alnAsRanges[-length(alnAsRanges)]
> oCounts <- lapply(countOverlaps(alnAsRanges,
+     iL), as.table)
> geneNames <- split(yeastAnno$ensembl_gene_id,
+     yeastAnno$chr)[-c(1, 7)]
> o <- order(as.integer(as.roman(names(geneNames))))
> geneNames <- geneNames[o]
> oCounts <- mapply(FUN = function(cnts,
+     nms) {
+     names(cnts) <- nms
+     cnts
+ }, oCounts, geneNames)
> lapply(oCounts, head)
```