Sequences, Reads, Alignments, and Ranges

Marc Carlson, Valerie Obenchain, Hervé Pagès, Paul Shannon, Dan Tenenbaum, Martin Morgan[1]

June 23 – 28, 2013

[1] mtmorgan@fhcrc.org

Contents

Chapter 1

Sequencing

Exercise 1
Set up your system to do the exercises in this lab. In addition to the packages you were asked to install prior to the course, you'll need to install the Morgan2013 package and `BigData` *directory of sample data files. Do this following the instructions in the lab. The following line MUST succeed*

```
> SYSTEM_OK <- getRversion() > "3.0" && require(Morgan2013, quietly=TRUE)
> stopifnot(SYSTEM_OK)
```

In addition, some of the exercises require access to data files. These should be installed at a convenient place; the exercises assume that the files are at the location returned by

```
> path.expand("~/BigData")
```

There should be a total of 39 files

```
> stopifnot(length(dir("~/BigData", recursive=TRUE)) == 39)
```

Recent technological developments introduce high-throughput sequencing approaches. A variety of experimental protocols and analysis work flows address gene expression, regulation, and encoding of genetic variants. Experimental protocols produce a large number (tens to hundreds of millions per sample) of short (e.g., 35-100's, single or paired-end) nucleotide sequences. These are aligned to a reference or other genome. Analysis work flows use the alignments to infer levels of gene expression (RNA-seq), binding of regulatory elements to genomic locations (ChIP-seq), or prevalence of structural variants (e.g., SNPs, short indels, large-scale genomic rearrangements). Sample sizes range from minimal replication (e.g,. 2 samples per treatment group) to thousands of individuals.

1.1 Technologies

The most common 'second generation' technologies readily available to labs are

- Illumina single- and paired-end reads. Short ($\sim$ 100 per end) and very numerous. Flow cell, lane, bar-code.
- Roche 454. 100's of nucleotides, 100,000's of reads.
- Life Technologies SOLiD. Unique 'color space' model.
- Complete Genomics. Whole genome sequence / variants / etc as a service; end user gets derived results.

Figure 1.1 illustrates Illumina and 454 sequencing. *Bioconductor* has good support for Illumina and Roche 454 sequencing products, and for derived data such as aligned reads or called variants; use of SOLiD color space reads typically requires conversion to FASTQ files that undermine the benefit of the color space model.

All second-generation technologies rely on PCR and other techniques to generate reads from samples that represent aggregations of many DNA molecules. 'Third-generation' technologies shift to single-molecule sequencing, with relevant players including Pacific Biosciences and IonTorent. This data is not widely available, and will not be discussed further.

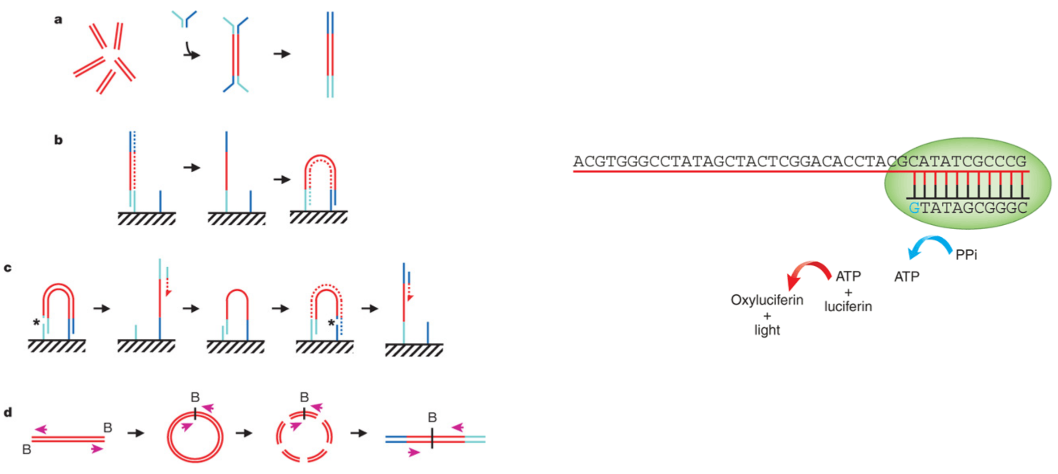

Figure 1.1: High-throughput sequencing. Left: Illumina bridge PCR [1]; mis-call errors. Right: Roche 454 [8]; homopolymer errors.

The most common data in *Bioconductor* work flows is from Illumina sequencers. Reads are either single-end or paired-end. Single-end reads represent $30- \sim 100$ nucleotides sequenced from DNA that has been sheared into ~ 300 nucleotide fragments. Paired-end reads represent $30- \sim 100$ nucleotide reads that are paired, and from both ends of the ~ 300 fragment.

Sequence data can be derived from a tremendous diversity of experiments. Some of the most common include:

RNA-seq Sequencing of reverse-complemented mRNA from the entire expressed transcriptome, typically. Used for *differential expression* studies like micro-arrays, or for *novel transcript discovery.*

DNA-seq Sequencing of whole or targeted (e.g., exome) genomic DNA. Common goals include *SNP* detection, *indel* and other structural polymorphisms, and *CNV* (copy number variation). DNA-seq is also used for *de novo* assembly, but *de novo* assembly is not an area where *Bioconductor* contributes.

ChIP-seq ChIP (chromatin immuno-precipitation) is used to enrich genomic DNA for regulatory elements, followed by sequencing and mapping of the enriched DNA to a reference genome. The initial statistical challenge is to identify regions where the mapped reads are enriched relative to a sample that did not undergo ChIP[5]; a subsequent task is to identify differential binding across a designed experiment, e.g., [7].

Metagenomics Sequencing generates sequences from samples containing multiple species, typically microbial communities sampled from niches such as the human oral cavity. Goals include inference of *species composition* (when sequencing typically targets phylogenetically informative genes such as 16S) or *metabolic contribution.*

1.2 Data

As a running example, we use the *pasilla* data set, derived from [2]. The authors investigate conservation of RNA regulation between *D. melanogaster* and mammals. Part of their study used RNAi and RNA-seq to identify exons regulated by Pasilla (*ps*), the *D. melanogaster* ortholog of mammalian NOVA1 and NOVA2. Briefly, their experiment compared gene expression as measured by RNAseq in S2-DRSC cells cultured with, or without, a 444bp dsRNA fragment corresponding to the *ps* mRNA sequence. Their assessment investigated differential exon use, but our worked example will focus on gene-level differences. For several examples we look at a subset of the *ps* data, corresponding to reads obtained from lanes of their RNA-seq experiment, and to the same reads aligned to a *D. melanogaster* reference genome. Reads were obtained from GEO and the Short Read Archive (SRA), and were aligned to the *D. melanogaster* reference genome *dm3* as described in the *pasilla* experiment data package.

At a very high level, one can envision a work flow that starts with a challenging biological question (how does *ps* influence gene and transcript regulation?). The biological question is framed in terms of

Table 1.1: Common file types (e.g., `http://genome.ucsc.edu/FAQ/FAQformat.html`) and *Bioconductor* packages used for input.

File	Description	Package
FASTQ	Unaligned sequences: identifier, sequence, and encoded quality score tuples	*ShortRead*
BAM	Aligned sequences: identifier, sequence, reference sequence name, strand position, cigar and additional tags	*Rsamtools*
GFF, GTF	Gene annotations: reference sequence name, data source, feature type, start and end positions, strand, etc.	*rtracklayer*
BED	Range-based annotation: reference sequence name, start, end coordinates.	*rtracklayer*
WIG, bigWig	'Continuous' single-nucleotide annotation.	*rtracklayer*
VCF	Called single nucleotide, indel, copy number, and structural variants, often compressed and indexed (with *Rsamtools* `bgzip`, `indexTabix`)	*VariantAnnotation*

wet-lab protocols coupled with an appropriate and all-important experimental design. There are several well-known statistical issues at this stage, common to any experimental data. What treatments are going to be applied? How many replicates will there be of each? Is there likely to be sufficient power to answer the biologically relevant question? Reality is also important at this stage, as evidenced in the *pasilla* data where, as we will see, samples were collected using different methods (single versus paired end reads) over a time when there were rapid technological changes. Such reality often introduces confounding factors that require appropriate statistical treatment in subsequent analysis.

The work flow proceeds with data generation, involving both a wet-lab (sample preparation) component and actual sequencing. It is essential to acknowledge the biases and artifacts that are introduced at each of these stages. Sample preparation involves non-trivial amounts of time and effort. Large studies are likely to have batch effects (e.g., because work was done by different lab members, or different batches of reagent). Samples might have been prepared in ways that are likely to influence down-stream analysis, e.g., using a protocol involving PCR and hence introducing opportunities for sample-specific bias. DNA isolation protocols may introduce many artifacts, e.g., non-uniform representation of reads across the length of expressed genes in RNA-seq. The sequencing reaction itself is far from bias-free, with known artifacts of called base frequency, cycle-dependent accuracy and bias, non-uniform coverage, etc. At a minimum, the researcher needs to be aware of the opportunities for bias that can be introduced during sample preparation and sequencing.

The informatics component of work flows becomes increasing important during and after sequence generation. The sequencer is often treated as a 'black box', producing short reads consisting of 10's to 100's of nucleotides and with associated quality scores. Usually, the chemistry and informatics processing pipeline are sufficiently well documented that one can arrive at an understanding of biases and quality issues that might be involved; such an understanding is likely to be particularly important when embarking on questions or using protocols that are at the fringe of standard practice (where, after all, the excitement is).

The first real data seen by users are *fastq* files (Table 1.1). These files are often simple text files consisting of many millions of records, and are described in greater detail in Section 2.2. The center performing the sequencing typically vets results for quality, but these quality measures are really about the performance of their machines. It is very important to assess quality with respect to the experiment being undertaken – Are the numbers of reads consistent across samples? Are GC content and other observable aspects of the reads consistent with expectation? Are there anomalies in the sequence results that reflect primers or other reagents used during sample preparation? Are well-known artifacts of the experimental protocol evident in the reads in hand?

The next step in many work flows involves alignment of reads to a reference genome. There are many aligners available, including BWA [4], Bowtie / Bowtie2 [3], and GSNAP; merits of these are discussed in the literature. *Bioconductor* packages 'wrapping' these tools are increasingly common (e.g., *Rbowtie*, *gmapR*; *cummeRbund* for parsing output of the *cufflinks* transcript discovery pathway). There are also alignment algorithms implemented in *Bioconductor* (e.g., `matchPDict` in the *Biostrings* package, and the

Rsubread package); `matchPDict` is particularly useful for flexible alignment of moderately sized subsets of data. Most main-stream aligners produce output in 'SAM' or 'BAM' (binary alignment) format. BAM files are the primary starting point for many analyses, and their manipulation and use in *Bioconductor* is introduced in Section 3.2.

Common analyses often use well-established third-party tools for initial stages of the analysis; some of these have *Bioconductor* counterparts that are particularly useful when the question under investigation does not meet the assumptions of other facilities. Some common work flows (a more comprehensive list is available on the SeqAnswers wiki[1]) include:

ChIP-seq ChIP-seq experiments typically use DNA sequencing to identify regions of genomic DNA enriched in prepared samples relative to controls. A central task is thus to identify peaks, with common tools including *MACS* and *PeakRanger*.

RNA-seq In addition to the aligners mentioned above, RNA-seq for differential expression might use the *HTSeq*[2] python tools for counting reads within regions of interest (e.g., known genes) or a pipeline such as the *bowtie* (basic alignment) / *tophat* (splice junction mapper) / *cufflinks* (esimated isoform abundance) (e.g., [3]) or *RSEM*[4] suite of tools for estimating transcript abundance.

DNA-seq especially variant calling can be facilitated by software such as the GATK[5] toolkit.

There are many *R* packages that replace or augment the analyses outlined above.

Programs such as those outlined the previous paragraph often rely on information about gene or other structure as input, or produce information about chromosomal locations of interesting features. The GTF and BED file formats are common representations of this information. Representing these files as *R* data structures is often facilitated by the *rtracklayer* package. We explore these files in the lab on gene and genome annotation. Variants are very commonly represented in VCF (Variant Call Format) files; these are explored in the lab on variants.

1.3 *Bioconductor* for sequence analysis

Table 1.2 enumerates some of the packages available for sequence analysis. The table includes packages for representing sequence-related data (e.g., *GenomicRanges*, *Biostrings*), as well as domain-specific analysis such as RNA-seq (e.g., *edgeR*, *DESeq2*), ChIP-seq (e.g,. *ChIPpeakAnno*, *DiffBind*), variants (e.g., *VariantAnnotation*, *VariantTools*, and SNPs and copy number variation (e.g., *genoset*, *ggtools*).

[1] http://seqanswers.com/wiki/RNA-Seq
[2] http://www-huber.embl.de/users/anders/HTSeq/doc/overview.html
[3] http://bowtie-bio.sourceforge.net/index.shtml
[4] http://deweylab.biostat.wisc.edu/rsem/
[5] http://www.broadinstitute.org/gatk/

Table 1.2: Selected *Bioconductor* packages for high-throughput sequence analysis.

Concept	Packages
Data representation	*IRanges*, *GenomicRanges*, *GenomicFeatures*, *Biostrings*, *BSgenome*, *girafe*.
Input / output	*ShortRead* (fastq), *Rsamtools* (bam), *rtracklayer* (gff, wig, bed), *VariantAnnotation* (vcf), *R453Plus1Toolbox* (454).
Annotation	*GenomicFeatures*, *ChIPpeakAnno*, *VariantAnnotation*.
Alignment	*gmapR*, *Rsubread*, *Biostrings*.
Visualization	*ggbio*, *Gviz*.
Quality assessment	*qrqc*, *seqbias*, *ReQON*, *htSeqTools*, *TEQC*, *Rolexa*, *ShortRead*.
RNA-seq	*BitSeq*, *cqn*, *cummeRbund*, *DESeq2*, *DEXSeq*, *EDASeq*, *edgeR*, *gage*, *goseq*, *iASeq*, *tweeDEseq*.
ChIP-seq, etc.	*BayesPeak*, *baySeq*, *ChIPpeakAnno*, *chipseq*, *ChIPseqR*, *ChIPsim*, *CSAR*, *DiffBind*, *MEDIPS*, *mosaics*, *NarrowPeaks*, *nucleR*, *PICS*, *PING*, *REDseq*, *Repitools*, *TSSi*.
Variants	*VariantAnnotation*, *VariantTools*, *gmapR*
SNPs	*snpStats*, *GWASTools*, *hapFabia*, *GGtools*
Copy number	*cn.mops*, *genoset*, *CNAnorm*, *exomeCopy*, *seqmentSeq*.
Motifs	*MotifDb*, *BCRANK*, *cosmo*, *cosmoGUI*, *MotIV*, *seqLogo*, *rGADEM*.
3C, etc.	*HiTC*, *r3Cseq*.
Microbiome	*phyloseq*, *DirichletMultinomial*, *clstutils*, *manta*, *mcaGUI*.
Work flows	*QuasR*, *easyRNASeq*, *ArrayExpressHTS*, *Genominator*, *oneChannelGUI*, *rnaSeqMap*.
Database	*SRAdb*.

Chapter 2

Strings and Reads

2.1 DNA (and other) Strings with the *Biostrings* package

Table 2.1 lists select *Bioconductor* packages for representing strings and reads. The *Biostrings* package provides tools for working with sequences. The essential data structures are *DNAString* and *DNAStringSet*, for working with one or multiple DNA sequences. The *Biostrings* package contains additional classes for representing amino acid and general biological strings. The *BSgenome* and related packages (e.g., *BSgenome.Dmelanogaster.UCSC.dm3*) are used to represent whole-genome sequences. Table 2.2 summarizes common operations; The following exercise explores these packages.

Exercise 2
The objective of this exercise is to calculate the GC content of the exons of a single gene. We jump into the middle of some of the data structures common in Bioconductor; these are introduced more thoroughly in later exercises..

Load the BSgenome.Dmelanogaster.UCSC.dm3 data package, containing the UCSC representation of D. melanogaster *genome assembly dm3. Discover the content of the package by evaluating* `Dmelanogaster`*.*

Load the Morgan2013 package, and evaluate the command `data(ex)` *to load an example of a GRangesList object. the GRangesList represents coordinates of exons in the* D. melanogaster *genome, grouped by gene.*

Look at `ex[1]`*. These are the genomic coordinates of the first gene in the* `ex` *object. Load the* D. melanogaster *chromosome that this gene is on by subsetting the* `Dmelanogaster` *object.*

Use `Views` *to create views on to the chromosome that span the start and end coordinates of all exons in the first gene; the start and end coordinates are accessed with* `start(ex[[1]])` *and similar.*

Develop a function `gcFunction` *to calculate GC content. Use this to calculate the GC content in each of the exons.*

Solution: Here we load the *D. melanogaster* genome, select a single chromosome, and create `Views` that reflect the ranges of the `FBgn0002183`.

```
> library(BSgenome.Dmelanogaster.UCSC.dm3)
> Dmelanogaster

Fly genome
|
```

Table 2.1: Selected *Bioconductor* packages for representing strings and reads.

Package	Description
Biostrings	Classes (e.g., *DNAStringSet*) and methods (e.g., `alphabetFrequency`, `pairwiseAlignment`) for representing and manipulating DNA and other biological sequences.
BSgenome	Representation and manipulation of large (e.g., whole-genome) sequences.
ShortRead	I/O and manipulation of FASTQ files.

Table 2.2: Operations on strings in the *Biostrings* package.

	Function	Description
Access	`length`, `names`	Number and names of sequences
	`[`, `head`, `tail`, `rev`	Subset, first, last, or reverse sequences
	`c`	Concatenate two or more objects
	`width`, `nchar`	Number of letters of each sequence
	`Views`	Light-weight sub-sequences of a sequence
Compare	`==`, `!=`, `match`, `%in%`	Element-wise comparison
	`duplicated`, `unique`	Analog to `duplicated` and `unique` on character vectors
	`sort`, `order`	Locale-independent sort, order
	`split`, `relist`	Split or relist objects to, e.g., *DNAStringSetList*
Edit	`subseq`, `subseq<-`	Extract or replace sub-sequences in a set of sequences
	`reverse`, `complement`	Reverse, complement, or reverse-complement DNA
	`reverseComplement`	
	`translate`	Translate DNA to Amino Acid sequences
	`chartr`	Translate between letters
	`replaceLetterAt`	Replace letters at a set of positions by new letters
	`trimLRPatterns`	Trim or find flanking patterns
Count	`alphabetFrequency`	Tabulate letter occurrence
	`letterFrequency`	
	`letterFrequencyInSlidingView`	
	`consensusMatrix`	Nucleotide $\times$ position summary of letter counts
	`dinucleotideFrequency`	2-mer, 3-mer, and k-mer counting
	`trinucleotideFrequency`	
	`oligonucleotideFrequency`	
	`nucleotideFrequencyAt`	Nucleotide counts at fixed sequence positions
Match	`matchPattern`, `countPattern`	Short patterns in one or many (`v*`) sequences
	`vmatchPattern`, `vcountPattern`	
	`matchPDict`, `countPDict`	Short patterns in one or many (`v*`) sequences (mismatch only)
	`whichPDict`, `vcountPDict`	
	`vwhichPDict`	
	`pairwiseAlignment`	Needleman-Wunsch, Smith-Waterman, etc. pairwise alignment
	`matchPWM`, `countPWM`	Occurrences of a position weight matrix
	`matchProbePair`	Find left or right flanking patterns
	`findPalindromes`	Palindromic regions in a sequence. Also `findComplementedPalindromes`
	`stringDist`	Levenshtein, Hamming, or pairwise alignment scores
I/0	`readDNAStringSet`	FASTA (or sequence only from FASTQ). Also `readBStringSet`, `readRNAStringSet`, `readAAStringSet`
	`writeXStringSet`	
	`writePairwiseAlignments`	Write `pairwiseAlignment` as "pair" format
	`readDNAMultipleAlignment`	Multiple alignments (FASTA, "stockholm", or "clustal"). Also `readRNAMultipleAlignment`, `readAAMultipleAlignment`
	`write.phylip`	

```
| organism: Drosophila melanogaster (Fly)
| provider: UCSC
| provider version: dm3
| release date: Apr. 2006
| release name: BDGP Release 5
|
| single sequences (see '?seqnames'):
|   chr2L      chr2R      chr3L      chr3R      chr4       chrX       chrU
|   chrM       chr2LHet   chr2RHet   chr3LHet   chr3RHet   chrXHet    chrYHet
|   chrUextra
|
| multiple sequences (see '?mseqnames'):
|   upstream1000  upstream2000  upstream5000
|
| (use the '$' or '[[' operator to access a given sequence)
```

```
> library(Morgan2013)
> data(ex)
> ex[1]
```

```
GRangesList of length 1:
$FBgn0002183
GRanges with 9 ranges and 0 metadata columns:
      seqnames             ranges strand
         <Rle>          <IRanges>  <Rle>
  [1]    chr3L [1871574, 1871917]      -
  [2]    chr3L [1872354, 1872470]      -
  [3]    chr3L [1872582, 1872735]      -
  [4]    chr3L [1872800, 1873062]      -
  [5]    chr3L [1873117, 1873983]      -
  [6]    chr3L [1874041, 1875218]      -
  [7]    chr3L [1875287, 1875586]      -
  [8]    chr3L [1875652, 1875915]      -
  [9]    chr3L [1876110, 1876336]      -

---
seqlengths:
     chr2L  chr2LHet     chr2R  chr2RHet ...    chrXHet    chrYHet       chrM
  23011544    368872  21146708   3288761 ...     204112     347038      19517
```

```
> nm <- "chr3L"
> chr <- Dmelanogaster[[nm]]
> v <- Views(chr, start=start(ex[[1]]), end=end(ex[[1]]))
```

Here is the `gcFunction` helper function to calculate GC content:

```
> gcFunction
```

```
function (x)
{
    alf <- alphabetFrequency(x, as.prob = TRUE)
    rowSums(alf[, c("G", "C")])
}
<environment: namespace:Morgan2013>
```

The `gcFunction` is really straight-forward: it invokes the function `alphabetFrequency` from the *Biostrings* package. This returns a simple matrix of exon × nucleotide probabilities. The row sums of the `G` and `C` columns of this matrix are the GC contents of each exon.

The subject GC content is

```
> subjectGC <- gcFunction(v)
```

2.2 Reads and the *ShortRead* package

Short read formats Illumina technologies generate sequences by measuring incorporation of florescent nucleotides over successive PCR cycles. These sequencers produce output in a variety of formats, but *FASTQ* is ubiquitous. Each read is represented by a record of four components:

```
@SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
GTTTTGTCCAAGTTCTGGTAGCTGAATCCTGGGGCGC
+SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
```

The first and third lines (beginning with `@` and `+` respectively) are unique identifiers. The identifier produced by the sequencer typically includes a machine id followed by colon-separated information on the lane, tile, x, and y coordinate of the read. The example illustrated here also includes the SRA accession number, added when the data was submitted to the archive. The machine identifier could potentially be used to extract information about batch effects. The spatial coordinates (lane, tile, x, y) are often used to identify optical duplicates; spatial coordinates can also be used during quality assessment to identify artifacts of sequencing, e.g., uneven amplification across the flow cell, though these spatial effects are rarely pursued.

The second and fourth lines of the FASTQ record are the nucleotides and qualities of each cycle in the read. This information is given in 5' to 3' orientation as seen by the sequencer. A letter `N` in the sequence is used to signify bases that the sequencer was not able to call. The fourth line of the FASTQ record encodes the quality (confidence) of the corresponding base call. The quality score is encoded following one of several conventions, with the general notion being that letters later in the visible ASCII alphabet

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijk
lmnopqrstuvwxyz{|}~
```

are of higher quality; this is developed further below. Both the sequence and quality scores may span multiple lines.

Technologies other than Illumina use different formats to represent sequences. Roche 454 sequence data is generated by 'flowing' labeled nucleotides over samples, with greater intensity corresponding to longer runs of A, C, G, or T. This data is represented as a series of 'flow grams' (a kind of run-length encoding of the read) in Standard Flowgram Format (SFF). The *Bioconductor* package *R453Plus1Toolbox* has facilities for parsing SFF files, but after quality control steps the data are frequently represented (with some loss of information) as FASTQ.

Short reads in *R* FASTQ files can be read in to *R* using the `readFastq` function from the *ShortRead* package. Use this function by providing the path to a FASTQ file. There are sample data files available in the `/BigData` directory, each consisting of 1 million reads from a lane of the Pasilla data set.

```
> library(ShortRead)
> bigdata <- "~/BigData"  ## e.g., "C:/Users/mtmorgan/"
> fastqDir <- file.path(bigdata, "fastq")
> fastqFiles <- dir(fastqDir, full=TRUE)
> fq <- readFastq(fastqFiles[1])
> fq

class: ShortReadQ
length: 1000000 reads; width: 37 cycles
```

The data are represented as an object of class *ShortReadQ*.

```
> head(sread(fq), 3)
```

```
  A DNAStringSet instance of length 3
    width seq
[1]    37 GTTTTGTCCAAGTTCTGGTAGCTGAATCCTGGGGCGC
[2]    37 GTTGTCGCATTCCTTACTCTCATTCGGGAATTCTGTT
[3]    37 GAATTTTTTGAGAGCGAAATGATAGCCGATGCCCTGA
```

```
> head(quality(fq), 3)
```

```
class: FastqQuality
quality:
  A BStringSet instance of length 3
    width seq
[1]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
[2]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
[3]    37 IIIIIIIIIIIIIIIIIIIIII'IIIIIGBIIII2I+
```

```
> head(id(fq), 3)
```

```
  A BStringSet instance of length 3
    width seq
[1]    58 SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
[2]    57 SRR031724.2 HWI-EAS299_4_30M2BAAXX:5:1:937:1157 length=37
[3]    58 SRR031724.4 HWI-EAS299_4_30M2BAAXX:5:1:1443:1122 length=37
```

The *ShortReadQ* class illustrates *class inheritance*. It extends the *ShortRead* class

```
> getClass("ShortReadQ")
```

```
Class "ShortReadQ" [package "ShortRead"]

Slots:

Name:       quality        sread           id
Class: QualityScore DNAStringSet   BStringSet

Extends:
Class "ShortRead", directly
Class ".ShortReadBase", by class "ShortRead", distance 2

Known Subclasses: "AlignedRead"
```

Methods defined on *ShortRead* are available for *ShortReadQ*.

```
> showMethods(class="ShortRead", where=search())
```

For instance, the `width` can be used to demonstrate that all reads consist of 37 nucleotides.

```
> table(width(fq))
```

```
     37
1000000
```

The `alphabetByCycle` function summarizes use of nucleotides at each cycle in a (equal width) *ShortReadQ* or *DNAStringSet* instance.

```
> abc <- alphabetByCycle(sread(fq))
> abc[1:4, 1:8]
```

```
        cycle
alphabet   [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]
       A  78194 153156 200468 230120 283083 322913 162766 220205
       C 439302 265338 362839 251434 203787 220855 253245 287010
       G 397671 270342 258739 356003 301640 247090 227811 246684
       T  84833 311164 177954 162443 211490 209142 356178 246101
```

FASTQ files are getting larger. A very common reason for looking at data at this early stage in the processing pipeline is to explore sequence quality. In these circumstances it is often not necessary to parse the entire FASTQ file. Instead create a representative sample

```
> sampler <- FastqSampler(fastqFiles[1], 1000000)
> yield(sampler) # sample of 1000000 reads
```

```
class: ShortReadQ
length: 1000000 reads; width: 37 cycles
```

A second common scenario is to pre-process reads, e.g., trimming low-quality tails, adapter sequences, or artifacts of sample preparation. The *FastqStreamer* class can be used to 'stream' over the fastq files in chunks, processing each chunk independently.

ShortRead contains facilities for quality assessment of FASTQ files. Here we generate a report from a sample of 1 million reads from each of our files and display it in a web browser

```
> qas0 <- Map(function(fl, nm) {
+     fq <- FastqSampler(fl)
+     qa(yield(fq), nm)
+ }, fastqFiles,
+    sub("_subset.fastq", "", basename(fastqFiles)))
> qas <- do.call(rbind, qas0)
> rpt <- report(qas, dest=tempfile())
> browseURL(rpt)
```

A report from a larger subset of the experiment is available

```
> rpt <- system.file("GSM461176_81_qa_report", "index.html",
+                    package="Morgan2013")
> browseURL(rpt)
```

Exercise 3
Use the file path `bigdata` *and the* `file.path` *and* `dir` *functions to locate the fastq file from [2] (the file was obtained as described in the pasilla experiment data package).*

Input the fastq files using `readFastq` *from the ShortRead package.*

Use `alphabetFrequency` *to summarize the GC content of all reads (hint: use the* `sread` *accessor to extract the reads, and the* `collapse=TRUE` *argument to the* `alphabetFrequency` *function). Using the helper function* `gcFunction` *defined in the Morgan2013 package, draw a histogram of the distribution of GC frequencies across reads.*

Use `alphabetByCycle` *to summarize the frequency of each nucleotide, at each cycle. Plot the results using* `matplot`*, from the graphics package.*

As an advanced exercise, and if on Mac or Linux, use the parallel package and `mclapply` *to read and summarize the GC content of reads in two files in parallel.*

Solution: Discovery:

```
> dir(bigdata)
```

```
[1] "CompleteGenomics" "TERT"             "bam"              "fastq"
```

```
> fls <- dir(file.path(bigdata, "fastq"), full=TRUE)
```

Input:

```
> fq <- readFastq(fls[1])
```

GC content:

```
> alf0 <- alphabetFrequency(sread(fq), as.prob=TRUE, collapse=TRUE)
> sum(alf0[c("G", "C")])
```

```
[1] 0.5457237
```

A histogram of the GC content of individual reads is obtained with

```
> gc <- gcFunction(sread(fq))
> hist(gc)
```

Alphabet by cycle:

```
> abc <- alphabetByCycle(sread(fq))
> matplot(t(abc[c("A", "C", "G", "T"),]), type="l")
```

Advanced (Mac, Linux only): processing on multiple cores.

```
> library(parallel)
> gc0 <- mclapply(fls, function(fl) {
+   fq <- readFastq(fl)
+   gc <- gcFunction(sread(fq))
+   table(cut(gc, seq(0, 1, .05)))
+ })
> ## simplify list of length 2 to 2-D array
> gc <- simplify2array(gc0)
> matplot(gc, type="s")
```

Exercise 4
Use `quality` to extract the quality scores of the short reads. Interpret the encoding qualitatively.

Convert the quality scores to a numeric matrix, using `as()`. Inspect the numeric matrix (e.g., using `dim`) and understand what it represents.

Use `colMeans` to summarize the average quality score by cycle. Use `plot` to visualize this.

Solution:

```
> head(quality(fq))
```

```
class: FastqQuality
quality:
  A BStringSet instance of length 6
    width seq
[1]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
[2]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
[3]    37 IIIIIIIIIIIIIIIIIIIIII'IIIIIGBIIII2I+
[4]    37 IIIIIIIIIIIIIIIIIIIIIIII,II*E,&4HI++B
[5]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII&.$
[6]    37 III.IIIIIIIIIIIIIIIIIII%IIE(-EIH<IIII
```

```
> qual <- as(quality(fq), "matrix")
> dim(qual)
```

```
[1] 1000000      37
```

```
> plot(colMeans(qual), type="b")
```

Exercise 5
As an independent exercise when an internet connection is available, visit the qrqc landing page and explore the package vignette. Use the qrqc package (you may need to install this) to generate base and average quality plots for the data, like those in the report generated by ShortRead.

Chapter 3

Ranges and Alignments

Ranges describe both features of interest (e.g., genes, exons, promoters) and reads aligned to the genome. *Bioconductor* has very powerful facilities for working with ranges, some of which are summarized in Table 3.1.

3.1 Ranges and the *GenomicRanges* package

Next-generation sequencing data consists of a large number of short reads. These are, typically, aligned to a reference genome. Basic operations are performed on the alignment, asking e.g., how many reads are aligned in a genomic range defined by nucleotide coordinates (e.g., in the exons of a gene), or how many nucleotides from all the aligned reads cover a set of genomic coordinates. How is this type of data, the aligned reads and the reference genome, to be represented in *R* in a way that allows for effective computation?

The *IRanges*, *GenomicRanges*, and *GenomicFeatures* *Bioconductor* packages (Table 3.1) provide the essential infrastructure for these operations; we start with the *GRanges* class, defined in *GenomicRanges*.

GRanges Instances of *GRanges* are used to specify genomic coordinates. Suppose we wish to represent two *D. melanogaster* genes. The first is located on the positive strand of chromosome 3R, from position 19967117 to 19973212. The second is on the minus strand of the X chromosome, with 'left-most' base at 18962306, and right-most base at 18962925. The coordinates are *1-based* (i.e., the first nucleotide on a chromosome is numbered 1, rather than 0), *left-most* (i.e., reads on the minus strand are defined to 'start' at the left-most coordinate, rather than the 5' coordinate), and *closed* (the start and end coordinates are included in the range; a range with identical start and end coordinates has width 1, a 0-width range is represented by the special construct where the end coordinate is one less than the start coordinate).

A complete definition of these genes as *GRanges* is:

Table 3.1: Selected *Bioconductor* packages for representing and manipulating ranges.

Package	Description
IRanges	Defines important classes (e.g., *IRanges*, *Rle*) and methods (e.g., `findOverlaps`, `countOverlaps`) for representing and manipulating ranges of consecutive values. Also introduces *DataFrame*, *SimpleList* and other classes tailored to representing very large data.
GenomicRanges	Range-based classes tailored to sequence representation (e.g., *GRanges*, *GRangesList*), with information about strand and sequence name.
GenomicFeatures	Foundation for manipulating data bases of genomic ranges, e.g., representing coordinates and organization of exons and transcripts of known genes.
AnnotationHub	Access many common large-data resources (e.g., Encode tracks; Ensembl GTF files) as *GRanges* instances.

```
> genes <- GRanges(seqnames=c("3R", "X"),
+                  ranges=IRanges(
+                      start=c(19967117, 18962306),
+                      end=c(19973212, 18962925)),
+                  strand=c("+", "-"),
+                  seqlengths=c(`3R`=27905053L, `X`=22422827L))
```

The components of a *GRanges* object are defined as vectors, e.g., of `seqnames`, much as one would define a *data.frame*. The start and end coordinates are grouped into an *IRanges* instance. The optional `seqlengths` argument specifies the maximum size of each sequence, in this case the lengths of chromosomes 3R and X in the 'dm2' build of the *D. melanogaster* genome. This data is displayed as

```
> genes

GRanges with 2 ranges and 0 metadata columns:
      seqnames               ranges strand
         <Rle>            <IRanges>  <Rle>
  [1]       3R [19967117, 19973212]      +
  [2]        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827
```

For the curious, the gene coordinates and sequence lengths are derived from the *org.Dm.eg.db* package for genes with Flybase identifiers FBgn0039155 and FBgn0085359, using the annotation facilities described in the lab on gene and genome annotation.

The *GRanges* class has many useful methods defined on it. Consult the help page

```
> ?GRanges
```

and package vignettes (especially 'An Introduction to *GenomicRanges*')

```
> vignette(package="GenomicRanges")
```

for a comprehensive introduction. A *GRanges* instance can be subset, with accessors for getting and updating information.

```
> genes[2]

GRanges with 1 range and 0 metadata columns:
      seqnames               ranges strand
         <Rle>            <IRanges>  <Rle>
  [1]        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827

> strand(genes)

factor-Rle of length 2 with 2 runs
  Lengths: 1 1
  Values : + -
Levels(3): + - *

> width(genes)

[1] 6096  620
```

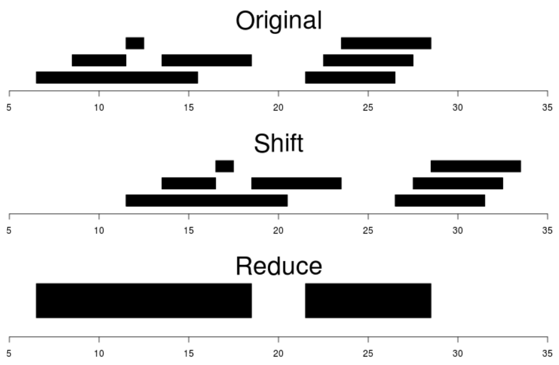

Figure 3.1: Ranges

```
> length(genes)

[1] 2

> names(genes) <- c("FBgn0039155", "FBgn0085359")
> genes  # now with names

GRanges with 2 ranges and 0 metadata columns:
              seqnames                 ranges strand
                 <Rle>              <IRanges>  <Rle>
  FBgn0039155       3R [19967117, 19973212]      +
  FBgn0085359        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827
```

`strand` returns the strand information in a compact representation called a *run-length encoding*, this is introduced in greater detail below. The 'names' could have been specified when the instance was constructed; once named, the *GRanges* instance can be subset by name like a regular vector.

As the `GRanges` function suggests, the *GRanges* class extends the *IRanges* class by adding information about `seqnames`, `strand`, and other information particularly relevant to representing ranges that are on genomes. The *IRanges* class and related data structures (e.g., *RangedData*) are meant as a more general description of ranges defined in an arbitrary space. Many methods implemented on the *GRanges* class are 'aware' of the consequences of genomic location, for instance treating ranges on the minus strand differently (reflecting the 5' orientation imposed by DNA) from ranges on the plus strand.

Operations on ranges The *GRanges* class has many useful methods from the *IRanges* class; some of these methods are illustrated here. We use *IRanges* to illustrate these operations to avoid complexities associated with strand and seqnames, but the operations are comparable on *GRanges*. We begin with a simple set of ranges:

```
> ir <- IRanges(start=c(7, 9, 12, 14, 22:24),
+               end=c(15, 11, 12, 18, 26, 27, 28))
```

These and some common operations are illustrated in the upper panel of Figure 3.1 and summarized in Table 3.2.

Methods on ranges can be grouped as follows:

Intra-range methods act on each range independently. These include `flank`, `narrow`, `reflect`, `resize`, `restrict`, and `shift`, among others. An illustration is `shift`, which translates each range by the amount specified by the `shift` argument. Positive values shift to the right, negative to the left;

`shift` can be a vector, with each element of the vector shifting the corresponding element of the *IRanges* instance. Here we shift all ranges to the right by 5, with the result illustrated in the middle panel of Figure 3.1.

```
> shift(ir, 5)
```

```
IRanges of length 7
    start end width
[1]    12  20     9
[2]    14  16     3
[3]    17  17     1
[4]    19  23     5
[5]    27  31     5
[6]    28  32     5
[7]    29  33     5
```

Inter-range methods act on the collection of ranges as a whole. These include `disjoin`, `reduce`, `gaps`, and `range`. An illustration is `reduce`, which reduces overlapping ranges into a single range, as illustrated in the lower panel of Figure 3.1.

```
> reduce(ir)
```

```
IRanges of length 2
    start end width
[1]     7  18    12
[2]    22  28     7
```

`coverage` is an inter-range operation that calculates how many ranges overlap individual positions. Rather than returning ranges, `coverage` returns a compressed representation (run-length encoding)

```
> coverage(ir)
```

```
integer-Rle of length 28 with 12 runs
  Lengths: 6 2 4 1 2 3 3 1 1 3 1 1
  Values : 0 1 2 1 2 1 0 1 2 3 2 1
```

The run-length encoding can be interpreted as 'a run of length 6 of nucleotides covered by 0 ranges, followed by a run of length 2 of nucleotides covered by 1 range.

Between methods act on two (or sometimes more) *IRanges* instances. These include `intersect`, `setdiff`, `union`, `pintersect`, `psetdiff`, and `punion`. The `countOverlaps` and `findOverlaps` functions operate on two sets of ranges. `countOverlaps` takes its first argument (the `query`) and determines how many of the ranges in the second argument (the `subject`) each overlaps. The result is an integer vector with one element for each member of `query`. `findOverlaps` performs a similar operation but returns a more general matrix-like structure that identifies each pair of query / subject overlaps. Both arguments allow some flexibility in the definition of 'overlap'.

Common operations on ranges are summarized in Table 3.2.

`mcols` and `metadata` The *GRanges* class (actually, most of the data structures defined or extending those in the *IRanges* package) has two additional very useful data components. The `mcols` function allows information on each range to be stored and manipulated (e.g., subset) along with the *GRanges* instance. The element metadata is represented as a *DataFrame*, defined in *IRanges* and acting like a standard *R data.frame* but with the ability to hold more complicated data structures as columns (and with element metadata of its own, providing an enhanced alternative to the *Biobase* class *AnnotatedDataFrame*).

```
> mcols(genes) <- DataFrame(EntrezId=c("42865", "2768869"),
+                           Symbol=c("kal-1", "CG34330"))
```

`metadata` allows addition of information to the entire object. The information is in the form of a list; any data can be provided.

```
> metadata(genes) <-
+     list(CreatedBy="A. User", Date=date())
```

Table 3.2: Common operations on *IRanges*, *GRanges* and *GRangesList*.

Category	Function	Description
Accessors	`start`, `end`, `width`	Get or set the starts, ends and widths
	`names`	Get or set the names
	`mcols`, `metadata`	Get or set metadata on elements or object
	`length`	Number of ranges in the vector
	`range`	Range formed from min start and max end
Ordering	`<`, `<=`, `>`, `>=`, `==`, `!=`	Compare ranges, ordering by start then width
	`sort`, `order`, `rank`	Sort by the ordering
	`duplicated`	Find ranges with multiple instances
	`unique`	Find unique instances, removing duplicates
Arithmetic	`r + x`, `r - x`, `r * x`	Shrink or expand ranges `r` by number `x`
	`shift`	Move the ranges by specified amount
	`resize`	Change width, anchoring on start, end or mid
	`distance`	Separation between ranges (closest endpoints)
	`restrict`	Clamp ranges to within some start and end
	`flank`	Generate adjacent regions on start or end
Set operations	`reduce`	Merge overlapping and adjacent ranges
	`intersect`, `union`, `setdiff`	Set operations on reduced ranges
	`pintersect`, `punion`, `psetdiff`	Parallel set operations, on each `x[i]`, `y[i]`
	`gaps`, `pgap`	Find regions not covered by reduced ranges
	`disjoin`	Ranges formed from union of endpoints
Overlaps	`findOverlaps`	Find all overlaps for each `x` in `y`
	`countOverlaps`	Count overlaps of each `x` range in `y`
	`nearest`	Find nearest neighbors (closest endpoints)
	`precede`, `follow`	Find nearest `y` that `x` precedes or follows
	`x %in% y`	Find ranges in `x` that overlap range in `y`
Coverage	`coverage`	Count ranges covering each position
Extraction	`r[i]`	Get or set by logical or numeric index
	`r[[i]]`	Get integer sequence from `start[i]` to `end[i]`
	`subsetByOverlaps`	Subset `x` for those that overlap in `y`
	`head`, `tail`, `rev`, `rep`	Conventional R semantics
Split, combine	`split`	Split ranges by a factor into a *RangesList*
	`c`	Concatenate two or more range objects

GRangesList The `GRanges` class is extremely useful for representing simple ranges. Some next-generation sequence data and genomic features are more hierarchically structured. A gene may be represented by several exons within it. An aligned read may be represented by discontinuous ranges of alignment to a reference. The *GRangesList* class represents this type of information. It is a list-like data structure, which each element of the list itself a *GRanges* instance. The gene FBgn0039155 contains several exons, and can be represented as a list of length 1, where the element of the list contains a *GRanges* object with 7 elements (the *GRangesList* instance shown here is from a 'TranscriptDb' instance, covered in a later lab).

```
GRangesList of length 1:
$FBgn0039155
GRanges with 7 ranges and 2 metadata columns:
      seqnames               ranges strand |   exon_id   exon_name
         <Rle>            <IRanges>  <Rle> | <integer> <character>
  [1]    chr3R [19967117, 19967382]      + |     45515        <NA>
  [2]    chr3R [19970915, 19971592]      + |     45516        <NA>
  [3]    chr3R [19971652, 19971770]      + |     45517        <NA>
  [4]    chr3R [19971831, 19972024]      + |     45518        <NA>
  [5]    chr3R [19972088, 19972461]      + |     45519        <NA>
  [6]    chr3R [19972523, 19972589]      + |     45520        <NA>
  [7]    chr3R [19972918, 19973212]      + |     45521        <NA>

---
seqlengths:
    chr3R
 27905053
```

The *GRangesList* object has methods one would expect for lists (e.g., `length`, sub-setting). Many of the methods introduced for working with *IRanges* are also available, with the method applied element-wise.

The *GenomicFeatures* packages Many public resources provide annotations about genomic features. For instance, the UCSC genome browser maintains the 'knownGene' track of established exons, transcripts, and coding sequences of many model organisms. The *GenomicFeatures* package provides a way to retrieve, save, and query these resources. The underlying representation is as sqlite data bases, but the data are available in *R* as *GRangesList* objects. The following exercise explores the *GenomicFeatures* package and some of the functionality for the *IRanges* family introduced above.

Exercise 6

Load the TxDb.Dmelanogaster.UCSC.dm3.ensGene annotation package, and create an alias `txdb` *pointing to the TranscriptDb object this class defines.*

Extract all exon coordinates, organized by gene, using `exonsBy`*. What is the class of this object? How many elements are in the object? What does each element correspond to? And the elements of each element? Use* `elementLengths` *and* `table` *to summarize the number of exons in each gene, for instance, how many single-exon genes are there?*

Select just those elements corresponding to flybase gene ids FBgn0002183, FBgn0003360, FBgn0025111, and FBgn0036449. Use `reduce` *to simplify gene models, so that exons that overlap are considered 'the same'.*

Solution:

```
> library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
> txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene # alias
> ex0 <- exonsBy(txdb, "gene")
> head(table(elementLengths(ex0)))

   1    2    3    4    5    6
3182 2608 2070 1628 1133  886
```

Table 3.3: Fields in a SAM record. From http://samtools.sourceforge.net/samtools.shtml

Field	Name	Value
1	QNAME	Query (read) NAME
2	FLAG	Bitwise FLAG, e.g., strand of alignment
3	RNAME	Reference sequence NAME
4	POS	1-based leftmost POSition of sequence
5	MAPQ	MAPping Quality (Phred-scaled)
6	CIGAR	Extended CIGAR string
7	MRNM	Mate Reference sequence NaMe
8	MPOS	1-based Mate POSition
9	ISIZE	Inferred insert SIZE
10	SEQ	Query SEQuence on the reference strand
11	QUAL	Query QUALity
12+	OPT	OPTional fields, format TAG:VTYPE:VALUE

```
> ids <- c("FBgn0002183", "FBgn0003360", "FBgn0025111", "FBgn0036449")
> ex <- reduce(ex0[ids])
```

3.2 Alignments and the *Rsamtools* Package

Most down-stream analysis of short read sequences is based on reads aligned to reference genomes.

Alignment formats Most main-stream aligners produce output in SAM (text-based) or BAM format. A SAM file is a text file, with one line per aligned read, and fields separated by tabs. Here is an example of a single SAM line, split into fields.

```
> fl <- system.file("extdata", "ex1.sam", package="Rsamtools")
> strsplit(readLines(fl, 1), "\t")[[1]]
```

```
 [1] "B7_591:4:96:693:509"
 [2] "73"
 [3] "seq1"
 [4] "1"
 [5] "99"
 [6] "36M"
 [7] "*"
 [8] "0"
 [9] "0"
[10] "CACTAGTGGCTCATTGTAAATGTGTGGTTTAACTCG"
[11] "<<<<<<<<<<<<<<<;<<<<<<<<<5<<<<<;:<;7"
[12] "MF:i:18"
[13] "Aq:i:73"
[14] "NM:i:0"
[15] "UQ:i:0"
[16] "H0:i:1"
[17] "H1:i:0"
```

Fields in a SAM file are summarized in Table 3.3. We recognize from the FASTQ file the identifier string, read sequence and quality. The alignment is to a chromosome 'seq1' starting at position 1. The strand of alignment is encoded in the 'flag' field. The alignment record also includes a measure of mapping quality, and a CIGAR string describing the nature of the alignment. In this case, the CIGAR is 36M, indicating that the alignment consisted of 36 `M`atches or mismatches, with no indels or gaps; indels (relative to the reference) are represented by I and D; gaps (e.g., from alignments spanning introns) by N.

BAM files encode the same information as SAM files, but in a format that is more efficiently parsed by software; BAM files are the primary way in which aligned reads are imported in to *R*.

Aligned reads in *R* The `readGappedAlignments` function (this is renamed `readGalignments` in the 'devel' version of *Bioconductor*) from the *GenomicRanges* package reads essential information from a BAM file in to *R*. The result is an instance of the *GappedAlignments* class. The *GappedAlignments* class has been designed to allow useful manipulation of many reads (e.g., 20 million) under moderate memory requirements (e.g., 4 GB); strategies for dealing with large data are discussed in Chaptar 4.

```
> alnFile <- system.file("extdata", "ex1.bam", package="Rsamtools")
> aln <- readGappedAlignments(alnFile)
> head(aln, 3)

GappedAlignments with 3 alignments and 0 metadata columns:
      seqnames strand       cigar    qwidth     start       end     width
         <Rle>  <Rle> <character> <integer> <integer> <integer> <integer>
  [1]     seq1      +         36M        36         1        36        36
  [2]     seq1      +         35M        35         3        37        35
  [3]     seq1      +         35M        35         5        39        35
           ngap
      <integer>
  [1]         0
  [2]         0
  [3]         0
  ---
  seqlengths:
   seq1 seq2
   1575 1584
```

The `readGappedAlignments` function takes an additional argument, `param`, allowing the user to specify regions of the BAM file (e.g., known gene coordinates) and characteristics of the alignments (e.g., 'proper' pairs, as identified by the aligner) from which to extract alignments.

A *GappedAlignments* instance is like a data frame, but with accessors as suggested by the column names. It is easy to query, e.g., the distribution of reads aligning to each strand, the width of reads, or the cigar strings

```
> table(strand(aln))

   +    -    *
1647 1624    0

> table(width(aln))

  30   31   32   33   34   35   36   38   40
   2   21    1    8   37 2804  285    1  112

> head(sort(table(cigar(aln)), decreasing=TRUE))

     35M      36M      40M      34M      33M 14M4I17M
    2804      283      112       37        6        4
```

Exercise 7

Use `bigdata`*,* `file.path` *and* `dir` *to obtain file paths to the BAM files. These are a subset of the aligned reads, overlapping just four genes.*

Input the aligned reads from one file using `readGappedAlignments`*. Explore the reads, e.g., using* `table` *or* `xtabs`*, to summarize which chromosome and strand the subset of reads is from.*

The object `ex` *is created in a later exercises, but we will use it now. It contains coordinates of four genes. Use* `countOverlaps` *to first determine the number of genes an individual read aligns to, and then the number of uniquely aligning reads overlapping each gene. Since the RNAseq protocol was not strand-sensitive, set the strand of* `aln` *to* `*`*.*

Write the sequence of steps required to calculate counts as a simple function, and calculate counts on each file. On Mac or Linux, can you easily parallelize this operation?

Solution: We discover the location of files using standard *R* commands:

```
> bigdata <- "~/BigData"
> fls <- dir(file.path(bigdata, "bam"), ".bam$", full=TRUE) #$
> names(fls) <- sub("_subset.bam", "", basename(fls))
```

Use `readGappedAlignments` to input data from one of the files, and standard *R* commands to explore the data.

```
> ## input
> aln <- readGappedAlignments(fls[1])
> xtabs( ~ seqnames + strand, as.data.frame(aln))
```

```
        strand
seqnames    +    -
   chr3L 5402 5974
   chrX  2278 2283
```

To count overlaps in regions defined in a previous exercise, load the regions.

```
> data(ex)              # from a later exercise
```

Many RNA-seq protocols are not strand aware, i.e., reads align to the plus or minus strand regardless of the strand on which the corresponding gene is encoded. Adjust the strand of the aligned reads to indicate that the strand is not known.

```
> strand(aln) <- "*"   # protocol not strand-aware
```

One important issue when counting reads is to make sure that the reference names in both the annotation and the read files are identical.

Exercise 8
Check the reference name in both the `ex` *and* `aln`*. If they are not similar, how could you correct them?*

For simplicity, we are interested in reads that align to only a single gene. Count the number of genes a read aligns to...

```
> hits <- countOverlaps(aln, ex)
> table(hits)
```

```
hits
    0     1     2
  772 15026   139
```

Subset the aligned reads to contain just those aligning once, `aln[hits == 1]`. Finally, reverse the counting operation to count the number of times each region of interest aligns to a uniquely overlapping alignment.

```
> cnt <- countOverlaps(ex, aln[hits==1])
```

A simple function for counting reads (see `GenomicRangese::summarizeOverlaps` for more realistic counting schemes) is

```
> counter <-
+     function(filePath, range)
+ {
+     aln <- readGappedAlignments(filePath)
+     strand(aln) <- "*"
+     hits <- countOverlaps(aln, range)
+     countOverlaps(range, aln[hits==1])
+ }
```

This can be applied to all files using `sapply`

```
> counts <- sapply(fls, counter, ex)
```

The counts in one BAM file are independent of counts in another BAM file. This encourages us to count reads in each BAM file in parallel, decreasing the length of time required to execute our program. On Linux and Mac OS, a straight-forward way to parallelize this operation is:

```
> if (require(parallel))
+     simplify2array(mclapply(fls, counter, ex))
```

Chapter 4

Strategies for working with large data

Bioinformatics data is now very large; it is not reasonable to expect all of a fastq or bam file, for instance, to fit into memory. How is this data to be processed? This challenge confronts us in whatever language or tool we are using. In *R* and *Bioconductor*, main approaches are to: (1) *restrict* data input to the interesting subset of the larger data set; (2) *sample* from the large data, knowing that an appropriately sized sample will accurately estimate statistics we are interested in; (3) *iterate* through large data in chunks; and (4) use *parallel* evaluation on one or several computers. Let's look at each of these approaches.

4.1 Restriction

Just because a file contains a lot of data does not mean that we are interested in all of it. In base *R*, one might use the `colClasses` argument to `read.delim` or similar function (e.g., setting some elements to `NULL`) to read only some columns of a large comma-separated value file. In addition to the obvious benefit of using less memory than if all of the file had been read in, input will be substantially faster because less computation needs to be done to coerce values from their representation in the file to their representation in *R*'s memory.

A variation on the idea of restricting data input is to organize the data on disk into a representation that facilitates restriction. In base *R*, large data might be stored in a relational data base like the *sqlite* data bases that are built in to *R* and used in the *AnnotationDbi* *Bioconductor* packages. In addition to facilitating restriction, these approaches are typically faster than parsing a plain text file, because the data base software has stored data in a way that efficiently transforms data from its on-disk to in-memory representation.

Let's use BAM files and the *Rsamtools* package to illustrate restriction. Rather than using simple text files ("sam" format), we use "bam" (binary alignment) files that have been indexed. Use of a binary format enhances data input speed, while the index facilitates restrictions that take into account genomic coordinates. The basic approach will use the `ScanBamParam` function to specify a restriction.

A BAM record can contain a lot of information, including the relatively large sequence, quality, and query name strings. Not all information in the BAM file is needed for some calculations. For instance, one could calculate coverage (number of nucleotides overlapping each reference position) using only the `rname` (reference sequence name), `pos`, and `cigar` fields. We could arrange to input just this information with

```
> param <- ScanBamParam(what=c("rname", "pos", "cigar"))
```

Another common restriction is to particular genomic regions of interest. Regions of interest might be known genes in an RNAseq differential expression study, or promoter regions in a ChIP-seq study. Restrictions in genome space are specified using `GRanges` objects (typically computed from some reference source, rather than entered by hand) provided as the `which` argument to `ScanBamParam`. Here we create a

`GRanges` instance representing all exons on *D. melanogaster* chromosome 3L, and use that as a restriction to `ScanBamParam`'s `which` argument:

```
> library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
> exByGn <- exonsBy(TxDb.Dmelanogaster.UCSC.dm3.ensGene, "gene")
> seqlevels(exByGn, force=TRUE) <- "chr3L"
> gns <- unlist(range(exByGn))
> param <- ScanBamParam(which=gns)
```

The `what` and `which` restrictions can be combined with other restrictions into a single `ScanBamParam` object

```
> param <- ScanBamParam(what=c("rname", "pos", "cigar"), which=gns)
```

We can also restrict input to, e.g., paired-end reads that represent the primary (best) alignment; see the help page `?ScanBamParam` for a more complete description.

Here are some BAM files from an experiment in *D. melanogaster*; the BAM files contain a subset of aligned reads

```
> bigdata <- "~/BigData"
> bamfls <- dir(file.path(bigdata, "bam"), ".bam$", full=TRUE) #$
> names(bamfls) <- sub("_subset.bam", "", basename(bamfls))
```

We'll read in the first BAM file, but restrict input to "rname" to find how many reads map to each chromosome. We use the "low-level" function `scanBam` to input the data

```
> param <- ScanBamParam(what="rname")
> bam <- scanBam(bamfls[1], param=param)[[1]]
> table(bam$rname)

  chr2L   chr2R   chr3L   chr3R    chr4    chrM    chrX chrYHet
      0       0   11376       0       0       0    4561       0
```

Restriction is such a useful concept that many *Bioconductor* high throughput sequence analysis functions enable doing the right thing. Functions such as `coverage,BamFile-method` use restrictions to read in the specific data required for them to compute the statistic of interest. Most "higher level" functions have a `param=ScanBamParam()` argument. For instance, the primary user-friendly function for reading BAM files is `readGappedAlignments` (this is renammed to `readGAlignments` in *Bioconductor* version 2.13) reads the most useful information, allowing the user to specify additional fields if desired.

```
> gal <- readGappedAlignments(bamfls[1])
> head(gal, 3)

GappedAlignments with 3 alignments and 0 metadata columns:
      seqnames strand         cigar    qwidth     start       end     width
         <Rle>  <Rle>   <character> <integer> <integer> <integer> <integer>
  [1]    chr3L      - 20M498300N25M        45   1613818   2112162    498345
  [2]    chr3L      -  8M498300N37M        45   1613830   2112174    498345
  [3]    chr3L      -  8M498300N37M        45   1613830   2112174    498345
           ngap
      <integer>
  [1]         1
  [2]         1
  [3]         1
  ---
  seqlengths:
      chr2L    chr2R    chr3L    chr3R     chr4     chrM     chrX  chrYHet
   23011544 21146708 24543557 27905053  1351857    19517 22422827   347038

> param <- ScanBamParam(what="seq")  ## also input sequence
> gal <- readGappedAlignments(bamfls[1], param=param)
> head(mcols(gal)$seq)
```

```
  A DNAStringSet instance of length 6
    width seq
[1]    45 ATGTGTAGCGGGGGCACTAGCACCAGAAGCAAGGACAAACCCGCA
[2]    45 CGAACTAGCACCAGAATCAAGGACAAACCCGCAGTTGTCCATCAC
[3]    45 AGCTCTAGCACCAGAATCAAGGACAAACCCGCAGTTGTCCATCAC
[4]    45 GCGCGCAGGTGTGTGAGGAAGACCAGGGTGTTGTTGTCCTCGGGA
[5]    45 GCGCGCAGGTGTGTGAGGAAGACCAGGGTGTTGTTGTCCTCGTGT
[6]    45 CGCAGGTGTGTGAGGAAGACCAGGGTGTTGTTGTCCTCGAGATCG
```

4.2 Sampling

R is after all a statistical language, and it sometimes makes sense to draw inferences from a sample of large data. For instance, many quality assessment statistics summarize overall properties (e.g., GC content or per-nucleotide base quality of FASTQ reads) that do not require processing of the entire data. For these statistics to be valid, the sample from the file needs to be a random sample, rather than a sample of convenience.

There are two advantages to sampling from a FASTQ (or BAM) file. The sample uses less memory than the full data. And because less data needs to be parsed from the on-disk to in-memory representation the input is faster.

The *ShortRead* package `FastqSampler` function (see also `Rsamtools::BamSampler` visits a FASTQ file but retains only a random sample from the file. Here is our function to calculate GC content from a `DNAStringSet`:

```
> library(Morgan2013)
> gcFunction

function (x)
{
    alf <- alphabetFrequency(x, as.prob = TRUE)
    rowSums(alf[, c("G", "C")])
}
<environment: namespace:Morgan2013>
```

and a subset of a FASTQ file with 1 million reads:

```
> bigdata <- "~/BigData"
> fqfl <- dir(file.path(bigdata, "fastq"), ".fastq$", full=TRUE) #$
```

`FastqSampler` works by specifying the file name and desired sample size, e.g., 100,000 reads. This creates an object from which an independent sample can be drawn using the `yield` function.

```
> sampler <- FastqSampler(fqfl, 100000)
> fq <- yield(sampler)                    # 100,000 reads
> lattice::densityplot(gcFunction(sread(fq)), plot.points=FALSE)
> fq <- yield(sampler)                    # a different 100,000 reads
```

Generally, one would choose a sample size large enough to adequately characterize the data but not so large as to consume all (or a fraction, see section 4.4 below) of the memory. The default (1 million) is a reasonable starting point.

`FastqSampler` relies on the *R* random number generator, so the same sequence of reads can be sampled by using the same random number seed. This is a convenient way to sample the same read pairs from the FASTQ files typically used to represent paired-end reads

```
> ## NOT RUN
> set.seed(123)
> end1 <- yield(FastqSampler("end_1.fastq"))
> set.seed(123)
> end2 <- yield(FastqSampler("end_2.fastq"))
```

4.3 Iteration

Restriction may not be enough to wrestle large data down to size, and sampling may be inappropriate for the task at hand. A solution is then to iterate through the file. An example in base *R* is to open a file connection, and then read and process successive chunks of the file, e.g., reading chunks of 10000 lines

```
> ## NOT RUN
> con <- file("<hypothetical-file>.txt")
> open(f)
> while (length(x <- readLines(f, n=10000))) {
+     ## work on character vector 'x'
+ }
> close(f)
```

The pattern `length(x <- readLines(f, n=10000))` is a convenient short-hand pattern that reads the *next* 10000 lines into a variable `x` and then asks `x` it's length. `x` is created in the calling environment (so it is available for processing in the `while` loop) When there are no lines left to read, `length(x)` will evaluate to zero and the `while` loop will end.

Iteration really involves three steps: input of a 'chunk' of the data; calculation of a desired summary; and aggregation of summaries across chunks. We illustrate this with BAM files in the *Rsamtools* package; see also `FastqStreamer` in the *ShortRead* package and `TabixFile` for iterating through large VCF files (via `readVCF` in the *VariantAnnotation* package).

The first stage in iteration is to arrange for input of a chunk of data. In many programming languages one would iterate 'record-at-a-time', reading in one record, processing it, and moving to the next. *R* is not efficient when used in this fashion. Instead, we want to read in a larger chunk of data, typically as much as can comfortably fit in available memory. In *Rsamtools* for reading BAM files we arrange for this by creating a `BamFile` (or `TabixFile` for VCF) object where we specify an appropriate `yieldSize`. Here we go for a bigger BAM file

```
> library(RNAseqData.HNRNPC.bam.chr14)
> bamfl <- RNAseqData.HNRNPC.bam.chr14_BAMFILES[1]
> countBam(bamfl)

  space start end width                file records nucleotides
1    NA    NA  NA    NA ERR127306_chr14.bam  800484    57634848

> bf <- BamFile(bamfl, yieldSize=200000) # could be larger, e.g., 2-10 million
```

A `BamFile` object can be used to read data from a BAM file, e.g., using `readGappedAlignments`. Instead of reading all the data, the reading function will read just `yieldSize` records. The idea then is to open the BAM file and iterate through until no records are input. The pattern is

```
> ## initialize, e.g., for step 3 ...
> open(bf)
> while (length(gal <- readGappedAlignments(bf))) {
+     ## step 2: do work...
+     ## step 3: aggregate results...
+ }
> close(bf)
```

The second step is to perform a useful calculation on the chunk of data. This is particularly easy to do if chunks are independent of one another. For instance, a common operation is to count the number of times reads overlap regions of interest. There are functions that implement a variety of counting modes (see, e.g., `summarizeOverlaps` in the *GenomicRanges* package); here we'll go for a simple counter that arranges to tally one 'hit' each time a read overlaps (in any way) at most one range. Here is our counter function, taking as arguments a `GRanges` or `GRangesList` object representing the regions for which counts are desired, and a `GappedAlignments` object representing reads to be counted:

```
> counter <-
+     function(query, subject, ..., ignore.strand=TRUE)
+     ## query: GRanges or GRangesList
+     ## subject: GappedAlignments
+ {
+     if (ignore.strand)
+         strand(subject) <- "*"
+     hits <- countOverlaps(subject, query)
+     countOverlaps(query, subject[hits==1])
+ }
```

To set this step up a little more completely, we need to know the regions over which counting is to occur. Here we retrieve exons grouped by gene for the genome to which the BAM files were aligned:

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> query <- exonsBy(TxDb.Hsapiens.UCSC.hg19.knownGene, "gene")
```

Thus we'll add the following lines to our loop:

```
> ## initialize, e.g., for step 3 ...
> open(bf)
> while (length(gal <- readGappedAlignments(bf))) {
+     ## step 2: do work...
+     count0 <- counter(query, gal, ignore.strand=TRUE)
+     ## step 3: aggregate results...
+ }
> close(bf)
```

It is worth asking whether the 'do work' step will always be as straight-forward. The current example is easy because counting overlaps of one read does not depend on other reads, so the chunks can be processed independently of one another. *FIXME: These are paired-end reads, so counting is* not *this easy! See the lab on RNA-seq.*

The final step in iteration is to aggregate results across chunks. In our present case, `counter` always returns an integer vector with `length(query)` elements. We want to add these over chunks, and we can arrange to do this by starting with an initial vector of counts `counts <- integer(length(query))` and simply adding `count0` at each iteration. The complete iteration, packaged as a function to facilitate re-use is

```
> counter1 <-
+     function(bf, query, ...)
+ {
+     ## initialize, e.g., for step 3 ...
+     counts <- integer(length(query))
+     open(bf)
+     while (length(gal <- readGappedAlignments(bf, ...))) {
+         ## step 2: do work...
+         count0 <- counter(query, gal, ignore.strand=TRUE)
+         ## step 3: aggregate results...
+         counts <- counts + count0
+     }
+     close(bf)
+     counts
+ }
```

In action, this is invoked as

```
> bf <- BamFile(bamfl, yieldSize=500000)
> counts <- counter1(bf, query)
```

Counting is a particularly simple operation; one will often need to think carefully about how to aggregate statistics derived from individual chunks. A useful general approach is to identify *sufficient statistics* that represent a description of the data and can be easily aggregated across chunks. This is the approach taken by, for instance, the *biglm* package for fitting linear models to large data sets – the linear model is fit to successive chunks and the fit reduced to sufficient statistics, the sufficient statistics are then aggregated across chunks to arrive at an overall fit.

4.4 Parallel evaluation

Each of the preceding sections addressed memory management; what about overall performance?

The starting point is really writing efficient, vectorized code, with common strategies outlined elsewhere. Performance differences between poorly written versus well written *R* code can easily span two orders of magnitude, whereas parallel processing can only increase throughput by an amount inversely proportional to the number of processing units (e.g., CPUs) available.

The memory management techniques outlined earlier in this chapter are important in a parallel evaluation context. This is because we will typically be trying to exploit multiple processing cores on a single computer, and the cores will be competing for the same pool of shared memory. We thus want to arrange for the collection of processors to cooperate in dividing available memory between them, i.e., each processor needs to use only a fraction of total memory.

There are a number of ways in which *R* code can be made to run in parallel. The least painful and most effective will use 'multicore' functionality provided by the *parallel* package; the *parallel* package is installed by default with base *R*, and has a useful vignette [6]. Unfortunately, multicore facilities are not available on Windows computers.

Parallel evaluation on several cores of a single Linux or MacOS computer is particularly easy to achieve when the code is already vectorized. The solution on these operating systems is to use the functions `mclapply` or `pvec`. These functions allow the 'master' process to 'fork' processes for parallel evaluation on each of the cores of a single machine. The forked processes initially share memory with the master process, and only make copies of data when the forked process modifies a memory location ('copy on write' semantics). On Linux and MacOS, the `mclapply` function is meant to be a 'drop-in' replacement for `lapply`, but with iterations being evaluated on different cores. The following illustrates the use of `mclapply` for a toy vectorized function `f`:

```
> library(parallel)
> f <- function(i) {
+     cat("'f' called, length(i) = ", length(i), "\n")
+     sqrt(i)
+ }
> res0 <- mclapply(1:5, f, mc.cores=2)

'f' called, length(i) =  1
'f' called, length(i) =  1
'f' called, length(i) =  1
'f' called, length(i) =  1
'f' called, length(i) =  1
```

`pvec` is a little more subtle. It takes a vectorized function and distributes computation of different chunks of the vector across cores.

```
> res1 <- pvec(1:5, f, mc.cores=2)

'f' called, length(i) =  3
'f' called, length(i) =  2

> identical(unlist(res0), res1)

[1] TRUE
```

Both `mclapply` and `pvec` functions allow the user to specify the number of cores used, and how the data are divided into chunks.

The *parallel* package does not support fork-like behavior on Windows, where users need to more explicitly create a cluster of *R* workers and arrange for each to have the same data loaded into memory; similarly, parallel evaluation across computers (e.g., in a cluster) require more elaborate efforts to coordinate workers; this is typically done using `lapply`-like functions provided by the *parallel* package but specialized for simple ('snow') or more robust ('MPI') communication protocols between workers.

Data movement and random numbers are two important additional considerations in parallel evaluation. Moving data to and from cores to the manager can be expensive, so strategies that minimize explicit movement (e.g., passing file names or data base queries rather than *R* objects read from files; reducing data on the worker before transmitting results to the manager) are important. Random numbers need to be synchronized across cores to avoid generating the same sequences on each 'independent' computation.

How might parallel evaluation be used in *Bioconductor* work flows? One approach when working with BAM files exploits the fact that data are often organized with one sample per BAM file. Suppose we are interested in running our iterating counter `counter1` over several BAM files. We could do this by creating a `BamFileList` with appropriate `yieldSize`

```
> fls <- RNAseqData.HNRNPC.bam.chr14_BAMFILES  # 8 BAM files
> bamfls <- BamFileList(fls, yieldSize=500000) # use a larger yieldSize
```

and using an `lapply` to take each file in turn and performing the count.

```
> counts <- simplify2array(lapply(bamfls, counter1, query))
```

The parallel equivalent of this is simply (note the change from `lapply` to `mclapply`)

```
> options(mc.cores=detectCores())         # use all cores
> counts <- simplify2array(mclapply(bamfls, counter1, query))
> head(counts[rowSums(counts) != 0,], 3)
```

```
          ERR127306 ERR127307 ERR127308 ERR127309 ERR127302 ERR127303 ERR127304
10001           207       277       217       249       303       335       362
100128927       572       629       597       483       379       319       388
100129075        62        70        56        63        34        51        88
          ERR127305
10001           300
100128927       435
100129075        79
```

Note that `names(bamfls)` and `names(query)` are returned as row and column names of the data, reducing chance of mistaken labelling.

4.5 And. . .

The *foreach* package can be useful for parallel evaluation written using coding styles more like `for` loops rather than `lapply`. The *iterator* package is an abstraction that simplifies the notion of iterating over objects. The *Rmpi* package provides access to MPI, a structured environment for calculation on clusters. The *pbdR* formalism[1] is especially useful for well-structured distributed matrix computations.

The *MatrixEQTL* package is an amazing example implementing high performance algorithms on large data; the corresponding publication [9] is well worth studying.

Bioconductor provides an Amazon machine instance with MPI and *Rmpi* installed[2]. This can be an effective way to gain access to large computing resources.

[1]https://rdav.nics.tennessee.edu/2012/09/pbdr/
[2]http://bioconductor.org/help/bioconductor-cloud-ami/

References

[1] D. Bentley, S. Balasubramanian, H. Swerdlow, G. Smith, J. Milton, C. Brown, K. Hall, D. Evers, C. Barnes, H. Bignell, et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, 2008.

[2] A. N. Brooks, L. Yang, M. O. Duff, K. D. Hansen, J. W. Park, S. Dudoit, S. E. Brenner, and B. R. Graveley. Conservation of an RNA regulatory map between Drosophila and mammals. *Genome Research*, pages 193–202, 2011.

[3] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, 10:R25, 2009.

[4] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26:589–595, Mar 2010.

[5] P. J. Park. ChIP-seq: advantages and challenges of a maturing technology. *Nat. Rev. Genet.*, 10:669–680, Oct 2009. [PubMed Central:PMC3191340] [DOI:10.1038/nrg2641] [PubMed:19736561].

[6] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2013. ISBN 3-900051-07-0.

[7] C. Ross-Innes, R. Stark, A. Teschendorff, K. Holmes, H. Ali, M. Dunning, G. Brown, O. Gojis, I. Ellis, A. Green, et al. Differential oestrogen receptor binding is associated with clinical outcome in breast cancer. *Nature*, 481(7381):389–393, 2012.

[8] J. Rothberg and J. Leamon. The development and impact of 454 sequencing. *Nature biotechnology*, 26(10):1117–1124, 2008.

[9] A. A. Shabalin. Matrix eqtl: ultra fast eqtl analysis via large matrix operations. *Bioinformatics*, 28(10):1353–1358, 2012.