An introduction to *R* and *Bioconductor* for the analysis of high-throughput sequencing data

Pascal MARTIN[*]

[*]Pascal.Martin@inra.fr
Twitter: @PgpMartin
Github: pgpmartin

October 16, 2018

Contents

1 Introduction

1.1 What is *R*/*Bioconductor*?

R is a language and environment for statistical computing and graphics. The latest news and updates can be found on the *R* website. More than 6000 packages are available on the Comprehensive R Archive Network (CRAN) to extend the functionalities of *R*. *R* is increasingly used in different fields, including life science [1].
With more than 2000 packages, *Bioconductor* is the largest project associated to *R*. It is dedicated to bioinformatics [2] and proposes 4 types of packages:

- Software packages are classical *R* packages containing new functions
- Annotation Data packages contain biological annotations wrapped in convenient *R* objects
- Experiment Data packages contain Experimental data
- Workflow packages were introduced more recently and contain examples of workflows based on *Bioconductor* packages

The *Bioconductor* website is a great resource to get the latest (frequent) updates and to find help and tutorials.
One of the most notable advantage of *R* and *Bioconductor* is the vast amount of learning resources available. In addition to package vignettes (i.e. documentation), the *Bioconductor* workflows and Courses and Conferences section are great resources.

1.2 Core and specialized packages

An enormous amount of work has been done by *Bioconductor* developpers to provide efficient tools for the analysis of genomic data (see for example this excellent overview of the use of *Bioconductor* in the context of genomic data [3]).
Some packages provide a core infrastructure for genomics by defining key classes / containers and the associated accessors. Others provide complete pipelines or integrated solutions for a general task. These packages allow to perform the main steps involved in the analysis of genomic data:

- Work with sequences and strings: *Biostrings*, *BSgenome* and BSgenome* annotation packages
- Manipulate raw FASTQ data files: *ShortRead*
- Align reads on a reference: *Rsubread*, *QuasR*
- Manipulate aligned reads: *Rsamtools*, *GenomicAlignments*
- Work with genomic ranges: *IRanges*, *GenomicRanges*
- Access annotations: *AnnotationDbi*, *OrganismDbi*, *GenomicFeatures*, etc.
- Importing/exporting tracks: *rtracklayer*, *AnnotationHub*, *biomaRt*, etc.
- Visualize genomic data: *ggbio*, *Gviz*, *Sushi*, etc.

Other packages are more focused on specific applications, such as:

- RNA-seq and differential analysis: *limma*, *DESeq2*, *edgeR*, *DEXseq*, *spliceR*, *rnaSeqMap*, etc.
- ChIP-seq: *ChIPQC*, *chipseq*, *NarrowPeaks*, *DiffBind*, *MMDiff*, *epigenomix*, *jmosaics*, etc.
- DNA methylation: *bsseq*, *BiSeq*, *methylumi*, *minfi*, *Rnbeads*, etc.
- ATAC-seq: *ATACseqQC*, *esATAC*, etc.
- CAGE-seq: *TSSi*, *CAGEr*, etc.
- DNAse-seq: *DNaseR*, etc.
- MNase-seq: *PING*, etc.
- 3C/4C/Hi-C: *r3Cseq*, *FourCSeq*, *HiTC*, *InteractionSet*, *GenomicInterations*, etc.

Some packages such as *QuasR* or *systemPipeR* aim at providing complete pipelines to be run from the *R* environment.

1.3 Content and usage of this document

This document presents several packages that constitute the core *Bioconductor* tools dedicated to the analysis of functional genomics data. The chapters correspond to general tasks commonly undertaken in genomic studies and for each task the main functions from the core packages are illustrated. 'R code' is easily recognized by the grey background and text outputs are color-coded and preceded by the ## symbols. There is a progression in the document and objects created in a chapter are often used in subsequent chapters. It is thus necessary to save your R session when you make a pause in the document:

```
save.image("MySession.RData")
```

and to load it back when you start again:

```
load("MySession.RData")
```

This document does not document core packages to perform genetic studies (i.e. analyze SNPs, CNVs, GWAS, etc.) because I am less familiar with them. This might be a future development of this document.
This document is largely inspired and borrrows a lot from the documents, courses and vignettes written by others. I am grateful to them for sharing their knowledge and skills. I hope this document will also be useful to others and I invite the reader to embrace the attitude of sharing and respect that generally characterizes the *R* and *Bioconductor* community (#rstats, @Bioconductor).

2 Manipulating strings and sequences in Bioconductor

In this section, we will provide examples on the use of the following packages:

- *Biostrings*

- *Rsamtools*
- *BSgenome*
- *BSgenome.Dmelanogaster.UCSC.dm3*

2.1 Containers and accessors

First, we will see how to import sequences from a fasta file. We will use a file provided with the *Biostrings* package which contains the sequences 2000 bases upstream of the annotated transcription start sites (TSS) for the Drosophila melanogaster genome.

Get the file path:

```
dm3_upstream_filepath = system.file("extdata",
                                    "dm3_upstream2000.fa.gz",
                                    package="Biostrings")
```

Import the sequences as a DNAStringSet:

```
dm3_upstream = readDNAStringSet(dm3_upstream_filepath)
dm3_upstream

##   A DNAStringSet instance of length 26454
##         width seq                                        names
##     [1]  2000 GTTGGTGGCCCACCAGTGCC...AGTTTACCGGTTGCACGGT NM_078863_up_2000...
##     [2]  2000 TTATTTATGTAGGCGCCCGT...ACGGAAAGTCATCCTCGAT NM_001201794_up_2...
##     [3]  2000 TTATTTATGTAGGCGCCCGT...ACGGAAAGTCATCCTCGAT NM_001201795_up_2...
##     [4]  2000 TTATTTATGTAGGCGCCCGT...ACGGAAAGTCATCCTCGAT NM_001201796_up_2...
##     [5]  2000 TTATTTATGTAGGCGCCCGT...ACGGAAAGTCATCCTCGAT NM_001201797_up_2...
##     ...   ... ...
## [26450]  2000 ATTTACAAGACTAATAAAGA...AATTAAATTTCAATAAAAC NM_001111010_up_2...
## [26451]  2000 GATATACGAAGGACGACCTG...GTTTGAGTTGTTATATATT NM_001015258_up_2...
## [26452]  2000 GATATACGAAGGACGACCTG...GTTTGAGTTGTTATATATT NM_001110997_up_2...
## [26453]  2000 GATATACGAAGGACGACCTG...GTTTGAGTTGTTATATATT NM_001276245_up_2...
## [26454]  2000 CGTATGTATTAGTTAACTCT...AAAGTGTAAGAACAAATTG NM_001015497_up_2...
```

Create a random sequence:

```
randomSeq = DNAString(paste(sample(DNA_ALPHABET[1:4],
                                   size=24,
                                   replace=TRUE),
                            collapse=""))
randomSeq

##   24-letter "DNAString" instance
## seq: GTCTAAGTAAGGCTACAGTGTCCG
```

Load the whole Drosophila genome sequence:

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

```
## Fly genome:
## # organism: Drosophila melanogaster (Fly)
## # provider: UCSC
## # provider version: dm3
## # release date: Apr. 2006
## # release name: BDGP Release 5
## # 15 sequences:
## #   chr2L     chr2R     chr3L     chr3R     chr4      chrX      chrU
## #   chrM      chr2LHet  chr2RHet  chr3LHet  chr3RHet  chrXHet   chrYHet
## #   chrUextra
## # (use 'seqnames()' to see all the sequence names, use the '$' or '[['
## # operator to access a given sequence)

names(Dmelanogaster)

##  [1] "chr2L"     "chr2R"     "chr3L"     "chr3R"     "chr4"      "chrX"
##  [7] "chrU"      "chrM"      "chr2LHet"  "chr2RHet"  "chr3LHet"  "chr3RHet"
## [13] "chrXHet"   "chrYHet"   "chrUextra"

Dmelanogaster$chr2L

##   23011544-letter "DNAString" instance
## seq: CGACAATGCACGACAGAGGAAGCAGAACAGATATT...ATATTTGCAAATTTTGATGAACCCCCCTTTCAAA
```

Accessors:

```
dm3_upstream[[5]]

##   2000-letter "DNAString" instance
## seq: TTATTTATGTAGGCGCCCGTTCCCGCAGCCAAAGC...AATTAATCGATAGATACGGAAAGTCATCCTCGAT

toString(dm3_upstream[[5]][2:30])

## [1] "TATTTATGTAGGCGCCCGTTCCCGCAGCC"

subseq(dm3_upstream[[5]],start=2,end=30)

##   29-letter "DNAString" instance
## seq: TATTTATGTAGGCGCCCGTTCCCGCAGCC

Views(dm3_upstream[[5]],start=c(1,11,21),end=c(10,20,30))

##   Views on a 2000-letter DNAString subject
## subject: TTATTTATGTAGGCGCCCGTTCCCGCAGCCAAA...TTAATCGATAGATACGGAAAGTCATCCTCGAT
## views:
##     start end width
## [1]     1  10    10 [TTATTTATGT]
## [2]    11  20    10 [AGGCGCCCGT]
## [3]    21  30    10 [TCCCGCAGCC]
```

Views objects are used to store a sequence together with ranges defined on this sequence. The ranges are said to represent *views* onto the sequence (see paragraph 3.4 for details on these objects).

Working with large FASTA files

The *Rsamtools* package (presented in paragraph 5) also provides interesting functions to work on large indexed FASTA files (e.g. containing a whole genome sequence). The short example below illustrates how to extract from a FASTA file a set of sequences defined by a *GRanges* (see paragraph 3.5 for details on *GRanges*):

```
## library(Rsamtools)
indFaEx_path=system.file("extdata","ce2dict1.fa",package="Rsamtools")
indFaEx=FaFile(indFaEx_path)

getSeq(indFaEx,
       GRanges(c("pattern01:3-10",
                 "pattern04:10-24")))

##   A DNAStringSet instance of length 2
##     width seq                                               names
## [1]     8 GAAACTAG                                          pattern01
## [2]    15 TTGTTGCAAATTTGA                                   pattern04
```

2.2 Sequence analysis and masks

Reverse complement of a sequence (see also the `reverse` and `complement` functions):

```
reverseComplement(dm3_upstream[[5]])

##   2000-letter "DNAString" instance
## seq: ATCGAGGATGACTTTCCGTATCTATCGATTAATTC...CTTTGGCTGCGGGAACGGGCGCCTACATAAATAA
```

Count the occurence of each base:

```
alphabetFrequency(dm3_upstream[1:2],baseOnly=TRUE,as.prob=TRUE)

##          A     C      G      T other
## [1,] 0.323 0.191 0.1875 0.2985     0
## [2,] 0.300 0.207 0.2230 0.2700     0
```

Get the GC content of a sequence:

```
letterFrequency(Dmelanogaster$chr2L,"CG",as.prob=TRUE)

##     C|G
## 0.41835
```

Masked versions of BSgenome packages are generally available:

```
library("BSgenome.Dmelanogaster.UCSC.dm3.masked")
```

Activate/deactivate the masks:

```
maskedgenome <- BSgenome.Dmelanogaster.UCSC.dm3.masked #A MaskedBSgenome object
chrU <- maskedgenome$chrU #A MaskedDNAString object
active(masks(chrU)) #Only some masks are active

## AGAPS   AMB    RM   TRF
##  TRUE  TRUE FALSE FALSE

active(masks(chrU)) <- TRUE #turn on all masks
chrUmask=injectHardMask(chrU) #Replaces the masked nucleotides by "+"
as(chrU,"XStringViews") #Get the unmasked regions

##   Views on a 10049037-letter DNAString subject
## subject: TGCGTGCTACCACATCATGCAGTTTTCAAAGAA...AGCGCCTTTTTACGACCAACTGAGCGTACCAG
## views:
##           start      end width
##    [1]      3295     3309    15 [GTGAGGCATCACAAC]
##    [2]      3537     3858   322 [CTGAAATTACGTTATAATTTA...ACTAATTTGCGGAATTCGAC]
##    [3]      4410     5172   763 [AGCGCGCAAGCAAGAGAGGGA...TATGTCGGTGAAATATTAAT]
##    [4]      5904     5991    88 [CAGGTGCCCTTCCAAAGCAAA...TTTGCACTGGATAAGACAAG]
##    [5]      6109     6393   285 [CAAATTTGTAGAGGGGTGAGT...AAGCAACGCACCTCGACGTG]
##    ...       ...      ...   ... ...
## [8109] 10039126 10039126     1 [A]
## [8110] 10039540 10040228   689 [CAGCCATTTATTCTTATTTTC...ATTTTTAGTCGTCAGCGTTG]
## [8111] 10042946 10043532   587 [TATATAGAATATATTCGCCAA...ATTGGCAGGACAAGGCACAC]
## [8112] 10045835 10046264   430 [CTGTTTCCGTTGATTCCCGTT...TTTGCAAATTGAGCTCTAAA]
## [8113] 10046809 10046835    27 [TTTTATGGTTCGGTCAATTGTTTGGAT]
```

Hard masking changes the sequence itself while soft masking does not, as illustrated when extracting `Views` (see paragraph 3.4):

```
toString(Views(Dmelanogaster$chrU,start=1714848, width=12))

## [1] "GAGAGAGAGAGA"

toString(Views(chrU,start=1714848, width=12))

## Warning in Views(chrU, start = 1714848, width = 12):  masks were dropped

## [1] "GAGAGAGAGAGA"

toString(Views(chrUmask,start=1714848, width=12))

## [1] "++++++++++++"
```

Some functions take the masks into account without requiring hard masking. For example `alphabetFrequency` or, as shown below, the `matchPattern` function (see paragraph 2.3 for details on this function):

```
length(matchPattern('GAGAGAGAGAGA',maskedgenome$chrU))

## [1] 104

length(matchPattern('GAGAGAGAGAGA',chrU))

## [1] 8

length(matchPattern('GAGAGAGAGAGA',chrUmask))

## [1] 8

active(masks(chrU))=F #deactivate all masks
length(matchPattern('GAGAGAGAGAGA',chrU))

## [1] 104
```

```
active(masks(chrU))['RM']=T #activate only RepeatMasker
length(matchPattern('GAGAGAGAGAGA',chrU))

## [1] 14
```

2.3 Motifs and pattern matching

Sequence motifs or patterns are typically used to represent the DNA regions bound by transcription factors (TFs) and other regulatory proteins. There are several packages in *Bioconductor* to search for and identify such patterns in strings and sequences. Pattern matching is frequently used in ChIP-seq experiments to search e.g. for binding sites of transcription factors. A complete *Bioconductor* workflow illustrates the use of pattern matching for the identification of transcription factor binding sites.
Here, we illustrate the use of the following packages:

- *MotifDb*
- *seqLogo*
- *motifStack*
- *TFBSTools*

Other packages of interest include *rGADEM*, *BCRANK* and *motifRG* for de novo motif discovery, *MotIV* to compare motifs to databases such as JASPAR or *motifbreakR* to evaluate the impact of SNPs on TF binding sites.

2.3.1 Searching and plotting motifs.

The *MotifDb* package allows to query a collection of DNA motifs aggregated from different databases.

We search these databases for the response element of the ecdysone receptor (EcR):

```
EcRMotifs=MotifDb::query(MotifDb,"EcR")
EcRMotifs

## MotifDb object of length 3
## | Created from downloaded public sources: 2013-Aug-30
## | 3 position frequency matrices from 3 sources:
## |    FlyFactorSurvey:    1
## |        JASPAR_2014:    1
## |         jaspar2016:    1
## | 1 organism/s
## |      Dmelanogaster:    3
## Dmelanogaster-FlyFactorSurvey-EcR_SANGER_5_FBgn0000546
## Dmelanogaster-JASPAR_2014-EcR::usp-MA0534.1
## Dmelanogaster-jaspar2016-EcR::usp-MA0534.1

EcRMotifs[[1]]

##           1 2 3 4          5 6         7   8
## A 0.6428571 0 0 1 0.00000000 0 0.0000000 0.0
## C 0.0000000 0 0 0 0.92857143 1 0.1428571 0.5
## G 0.0000000 0 1 0 0.07142857 0 0.0000000 0.0
## T 0.3571429 1 0 0 0.00000000 0 0.8571429 0.5
```

The motif is given as a Position Frequency Matrix (PFM), which, can be converted to a Position-weight matrix using the information present in the metadata of the object (number of sequences used to define the motif: see `elementMetadata`).

The package *seqLogo* allows to easily plot sequence logos:

Figure 1 shows EcR motif:

```
seqLogo::seqLogo(EcRMotifs[[1]])
```

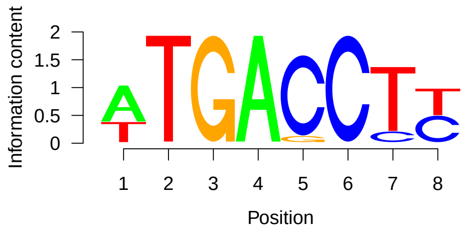


Figure 1: Sequence logo for EcR motif

Figure 2 its reverse complement:

```
seqLogo::seqLogo(reverseComplement(EcRMotifs[[1]]))
```

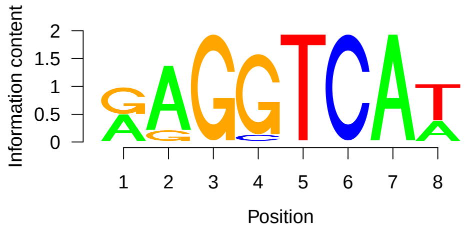


Figure 2: Sequence logo for EcR motif reverse complement

And Figure 3 the logo for the response element obtained from JASPAR:

```
seqLogo::seqLogo(EcRMotifs[[2]])
```

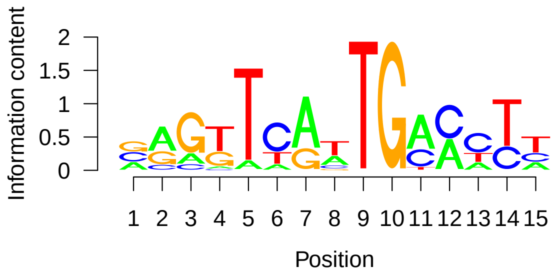


Figure 3: Sequence logo for EcR:Usp heterodimer

The second motif (from JASPAR) represented in Figure 3 is a binding site for the heterodimer composed of the ecdysone receptor (EcR) and its binding partner Ultraspiracle protein (Usp). This imperfect palindromic ecdysone response element is an inverted repeat of the consensus motif AGGTCA separated by 1 nucleotide (IR1).

The package *motifStack* provides additional plotting functionalities to plot multiple sequence logos, as illustrated in Figure 4:

```
#Format the PFMs:
pfms <- as.list(EcRMotifs)
names(pfms) <- gsub("Dmelanogaster-", "", names(pfms))
pfms <- lapply(names(pfms),
               function(x) {new("pfm",
                                mat=pfms[[x]],
                                name = x)})


#Plot
motifStack(pfms, layout="tree")
```

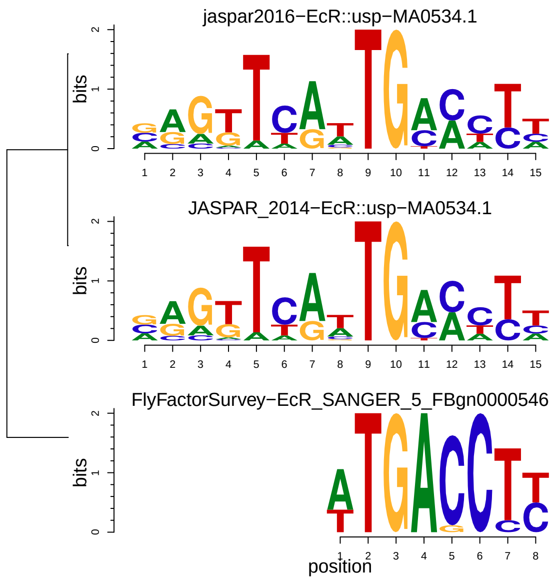


Figure 4: **Sequence logos for EcR motifs using motifStack**

For those using *ggplot2* for plotting, there is also the excellent *ggseqlogo* package.

2.3.2 Scanning a sequence with a Position-weight matrix.

A Position weight matrix (PWM) can be used to scan one or multiple "subject" sequences to search the location of the corresponding motif. At each position, the "subject" sequence is given a score based on the values present in the PWM.

We select the JASPAR2014 motif:

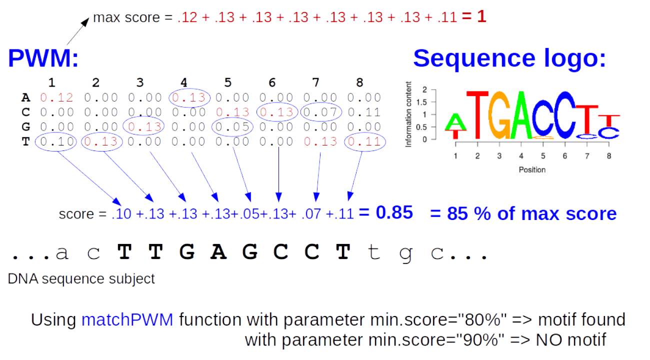

max score = .12 + .13 + .13 + .13 + .13 + .13 + .13 + .11 = 1
PWM:
Sequence logo:
1 2 3 4 5 6 7 8
A 0.12 0.00 0.00 0.13 0.00 0.00 0.00 0.00
C 0.00 0.00 0.00 0.00 0.13 0.13 0.07 0.11
G 0.00 0.00 0.13 0.00 0.05 0.00 0.00 0.00
T 0.10 0.13 0.00 0.00 0.00 0.00 0.13 0.11
Information content
Position
score = .10 +.13 +.13 +.13+.05+.13+ .07 +.11 = 0.85
= 85 % of max score
...a c T T G A G C C T t g c...
DNA sequence subject
Using matchPWM function with parameter min.score="80%" => motif found
with parameter min.score="90%" => NO motif

Figure 5: **Principle of sequence scoring with a PWM**

```r
EcrJASP <- EcRMotifs[[2]]
```

MotifDb returns a PFM with columns summing to 1. The number of sequences used to build the PFM are available in the `elementMetadata` or `mcols` of the object. We use this information to obtain a PWM

```r
nseq <- as.integer(mcols(EcRMotifs[2])$sequenceCount)
ecrpfm <- apply(round(nseq * EcrJASP,0), 2, as.integer)
rownames(ecrpfm) <- rownames(EcrJASP)
EcrJASP <- PWM(ecrpfm)
```

Here, we use the default background nucleotide frequencies (25% for each A, T, G and C) but this should be adapted depending on the genome studied.

Search for this motif on both strands of chromosome 2L:

```r
EcRJASP_2L=matchPWM(EcrJASP,
                    Dmelanogaster$chr2L,
                    min.score='90%')
EcRJASP_2L_rev=matchPWM(reverseComplement(EcrJASP),
                        Dmelanogaster$chr2L,
                        min.score='90%')
#One way to merge the results:
EcRJASP_2L_all <- Views(Dmelanogaster$chr2L,
                        union(ranges(EcRJASP_2L),
                              ranges(EcRJASP_2L_rev)))
```

We can use the `matchPWM` function to search a motif directly in a `BSgenome` object. In this case, the search is automatically done on both strands and the function returns a convenient `GRanges object` (see 3.5 below).

```r
EcRJASP_all <- matchPWM(EcrJASP, Dmelanogaster, min.score="90%")
EcRJASP_all

## GRanges object with 70653 ranges and 2 metadata columns:
##            seqnames               ranges strand |     score          string
##               <Rle>            <IRanges>  <Rle> | <numeric>  <DNAStringSet>
##        [1]    chr2L       [ 8094,  8108]      + | 0.9106767 AAAGTCAGTGAAACC
##        [2]    chr2L       [ 8746,  8760]      + | 0.9058075 GAGGTCATTAACTTT
##       [3]     chr2L       [17627, 17641]      + | 0.9241086 CCCTTTAATGAACTA
##       [4]     chr2L       [19791, 19805]      + | 0.9028672 AAAATAAATGACCCC
##       [5]     chr2L       [20432, 20446]      + | 0.9241086 CCCTTTAATGAACTA
##       ...       ...                  ...    ... .       ...             ...
##   [70649] chrUextra [28702900, 28702914]      - | 0.9049213 AAGGACATTGTAATC
##   [70650] chrUextra [28715402, 28715416]      - | 0.9422644 ACATTCAATGCACTT
##   [70651] chrUextra [28795481, 28795495]      - | 0.9079439 AGGGTTATTGTCTCA
##   [70652] chrUextra [28808899, 28808913]      - | 0.9079439 AGGGTTATTGTCTCA
##   [70653] chrUextra [28834409, 28834423]      - | 0.9011531 CCAATCATTGCCTTA
##   -------
##   seqinfo: 15 sequences from an unspecified genome
```

2.3.3 TFBSTools

TFBS stands for Transcription Factor Binding Site. The *TFBSTools* package [4] includes several features to manipulate and analyze sequence motifs or potential TFBS:

- Containers for motifs (`XMatrix`), collections of motifs (`XMatrixList`), sets of motifs obtained with MEME (`MotifSet`) or from a motifSearch on a sequence (`SiteSet`)
- Containers and graphical representations of the most recent TFFM (Transcription Factor Flexible Model) motif representations
- Functions to convert between PWM (position-weight matrix), PFM (position frequency matrix) and ICM (information content matrix)
- Functions to compare motifs as matrices (PFMs/PWMs) or with IUPAC strings
- Functions to scan sequences or whole genome (`searchSeq`) and to scan aligned sequences (`searchAln`) or genomes (`searchPairBSgenome`)
- Functions to query the JASPAR database, using *Bioconductor* packages such as *JASPAR2018*
- Wrapper function for the de novo motif dicovery tool MEME

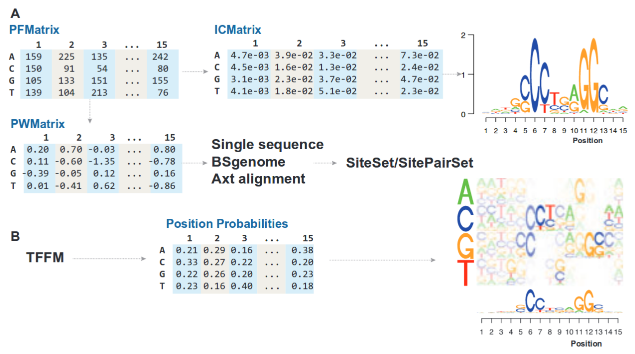


Figure 6: Common workflow and classes in *TFBSTools*
Taken from the TFBSTools paper

We search for the EcR:Usp binding motif in JASPAR2018 database:

```
EcR2018 <- TFBSTools::getMatrixByID(JASPAR2018, "MA0534")
```

Figure 7 shows EcR motif obtained from JASPAR2018:

```
TFBSTools::seqLogo(toICM(EcR2018))
```

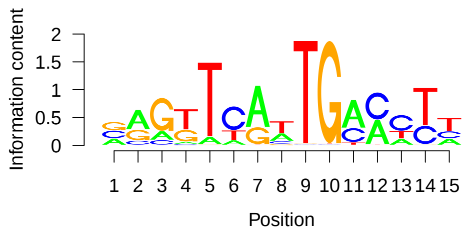


Figure 7: Sequence logo for EcR motif (JASPAR2018)

Compare EcR PWM with the PWM for a human nuclear receptor ESR1, the Drosophila CTCF or a randomly permuted PWM

```
#hESR1:
ESR1 <- TFBSTools::getMatrixByID(JASPAR2018, "MA0112")
PWMSimilarity(toPWM(EcR2018), toPWM(ESR1), method = "Pearson")

## [1] 0.5350265

#dCTCF
CTCF <- TFBSTools::getMatrixByID(JASPAR2018, "MA0531")
PWMSimilarity(toPWM(EcR2018), toPWM(CTCF), method = "Pearson")

## [1] 0.4987823

#random permutation:
PWMSimilarity(toPWM(EcR2018), toPWM(permuteMatrix(EcR2018)), method = "Pearson")

## [1] 0.3187625
```

Search for EcR motif on chr2L:

```
EcR2018_on2L <- searchSeq(toPWM(EcR2018), Dmelanogaster$chr2L, min.score="90%")
```

`searchPairBSgenome` is another interesting function that searches for a motif in regions that are conserved between 2 genomes given as `BSgenome`.

2.3.4 Scanning sequences with strings.

Scanning multiple sequences with one string.
A pattern or motif can also be represented as a character string possibly using the IUPAC ambiguity code. Here, for simplicity, we will represent ambiguities as ’N’.

To perform pattern matching, we define a consensus sequence for the IR1 ecdysone response element:

```
EcR_IR1_cons=consensusString(EcrJASP,ambiguityMap="N")
EcR_IR1_cons

## [1] "NAGTTCATTGACCTT"

EcR_IR1_cons=substring(EcR_IR1_cons,first=2)
EcR_IR1_cons

## [1] "AGTTCATTGACCTT"
```

Similarly, we can build consensus sequences for the EcR itself or for its binding partner Ultraspiracle (Usp):

```
EcR_cons=substring(consensusString(reverseComplement(EcRMotifs[[1]]),
                                   ambiguityMap="N"),
                   first=2)
Usp_cons=substring(consensusString(MotifDb::query(MotifDb,"Usp")[[5]],
                                   ambiguityMap="N"),
                   first=1,
                   last=7)
```

Now, we search for the EcR consensus in chromosome 2L:

```
EcR_on_2L=matchPattern(EcR_cons,Dmelanogaster$chr2L)
EcR_on_2L_rev=matchPattern(reverseComplement(DNAString(EcR_cons)),
                           Dmelanogaster$chr2L)
EcR_on_2L_all=Views(Dmelanogaster$chr2L,union(ranges(EcR_on_2L),
                                              ranges(EcR_on_2L_rev)))
```

Note that we have found a perfect palindrome:

```
EcR_on_2L_all[width(EcR_on_2L_all)!=7]

##   Views on a 23011544-letter DNAString subject
## subject: CGACAATGCACGACAGAGGAAGCAGAACAGATA...ATTTGCAAATTTTGATGAACCCCCCTTTCAAA
## views:
##       start     end width
## [1] 7208722 7208733    12 [AGGTCATGACCT]
```

Scanning multiple sequences with one string.
It is also possible to search for a single pattern in several subject sequences using the `vmatchPattern`.
Search for the EcR pattern in TSS upstream sequences:

```
EcR_on_up=vmatchPattern(EcR_cons, dm3_upstream)
```

Get the number of matches per subject element:

```
nmatch_per_seq = elementNROWS(EcR_on_up)
table(nmatch_per_seq)

## nmatch_per_seq
##     0     1     2     3
## 24765  1638    48     3
```

Let's take look at one of the upstream sequence with the maximum number of matches:

```
i0=which.max(nmatch_per_seq)
Views(dm3_upstream[[i0]], EcR_on_up[[i0]])

##   Views on a 2000-letter DNAString subject
## subject: AAATATCTATTTTCTTAGTTAAGTCCCATCAAA...TTCAAAATAACTGTATTAGTGGACTTTTCTGG
## views:
##     start  end width
## [1]  1633 1639     7 [AGGTCAT]
## [2]  1809 1815     7 [AGGTCAT]
## [3]  1829 1835     7 [AGGTCAT]
```

Scanning a sequence with multiple strings.
One may also search for multiple patterns in a single subject sequence using `matchPDict`. Get all PWM matrices available for Drosophila melanogaster:

```
dm_matrices = MotifDb::query(MotifDb,"dmelanogaster")
```

Keep only the motifs that are 8bp-long and get their consensus sequences:

```
motif_ln = sapply(dm_matrices,ncol)
dm_matrices = dm_matrices[motif_ln==8]
dm_motifs=DNAStringSet(sapply(dm_matrices,consensusString,ambiguityMap="N"))
```

Search for all these motifs in chromosome 2L:

```
mot8_on_2L=matchPDict(dm_motifs,Dmelanogaster$chr2L,fixed=FALSE)
summary(elementNROWS(mot8_on_2L)) #Number of matches

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     268     867    3289   57508   13143 4815391

head(unlist(mot8_on_2L)) #first 6 matches

## IRanges object with 6 ranges and 0 metadata columns:
##                                           start       end     width
##                                       <integer> <integer> <integer>
##   Dmelanogaster-cispb_1.02-M0111_1.02       118       125         8
##   Dmelanogaster-cispb_1.02-M0111_1.02       196       203         8
##   Dmelanogaster-cispb_1.02-M0111_1.02       576       583         8
##   Dmelanogaster-cispb_1.02-M0111_1.02       654       661         8
##   Dmelanogaster-cispb_1.02-M0111_1.02      1034      1041         8
##   Dmelanogaster-cispb_1.02-M0111_1.02      1112      1119         8
```

The motif most frequently found on chromosome 2L:

```
names(dm_motifs[which.max(elementNROWS(mot8_on_2L))])

## [1] "Dmelanogaster-cispb_1.02-M5115_1.02"

toString(dm_motifs[[which.max(elementNROWS(mot8_on_2L))]])

## [1] "NNNNGNNN"
```

The motif less frequently found on chromosome 2L:

```
names(dm_motifs[which.min(elementNROWS(mot8_on_2L))])

## [1] "Dmelanogaster-cispb_1.02-M5018_1.02"

toString(dm_motifs[[which.min(elementNROWS(mot8_on_2L))]])

## [1] "CGCGCGAT"
```

Scanning multiple sequence with multiple strings.
Finally, some functions, still under development, are available to search for multiple patterns in multiple sequences.

Remove the motifs containing N bases and create a dictionary of motifs (the motifs must have the same length):

```
dm_mot8_dict=PDict(dm_motifs[sapply(dm_motifs,hasOnlyBaseLetters)])
```

Search for the motifs in TSS upstream sequences:

```
mot8_count_upstream=vcountPDict(dm_mot8_dict,dm3_upstream)
```

Number of motifs found:

```
apply(mot8_count_upstream,1,sum)

##  [1]  975  566 1133  733 1445 1295 1133  920  498  521 9411 4366  399  292
## [15] 1281 1698 1698  761  475 2536  706  706  290 6162 2327  542 3063  724
## [29]  708 5781 1579 1340  702 4384 1939 6411  473  823 4560  742  714  952
## [43]  432  432 7916 6411  414  419 1284 1103  321  321  324  324  574  574
## [57] 6411 2527 3027 2549  469  469  885 1461  532 6411  724 1284 2527 6411
## [71]  724 1284 2527  885  515 6411  724 1284 2527  885
```

Number of motif1 per upstream sequence:

```
table(mot8_count_upstream[1,])

##
##     0     1     2     3     5
## 25527   887    36     2     2
```

Number of motifs per upstream sequence:

```
nMot8_perSeq=apply(mot8_count_upstream,2,sum)
names(nMot8_perSeq)=names(dm3_upstream)
```

Plot in Figure 8 the number of upstream sequences as a function of the number of motifs:

```
nMot8_perSeq=nMot8_perSeq[nMot8_perSeq>=1]
plot(as.integer(names(table(nMot8_perSeq))),
        as.integer(table(nMot8_perSeq)),
                pch=15,type="b",log="y",col="blue",
                xlab="Number of 8bp-long motifs",
                ylab="Number of upstream sequences")
```

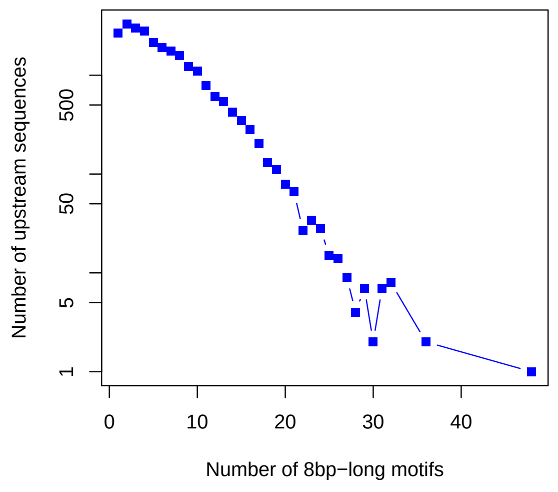


Figure 8: Motifs per upstream sequence

2.4 Sequence alignment

Sequence alignment is a common task in NGS bioinformatic pipelines. Most of the time, alignment of short or long reads is performed outside *R*, using dedicated NGS aligner.

Widely used short read aligners include:

- BWA [5, 6]
- Bowtie/Bowtie2 [7, 8]

Some aligners are also adapted to RNA-seq which often generates spliced reads:

- HiSat2 [9, 10]
- STAR [11]
- Subread [12]

There are *R* packages that provide interfaces to some of these aligners, such as *Rbowtie*, *Rbowtie2*, *QuasR* or *Rsubread*.
Here, we only briefly present classical sequence alignement tools available in *Bioconductor*, mainly via the *Biostrings* package (see the vignette of this package for further examples, including on NGS data). In NGS analysis, these tools can be used to search for adapters or primer sequences for example.

2.4.1 Pairwise alignment.

The main function `pairwiseAlignment` provides functionalities to perform global (Needleman-Wunsch), local (Smith-Waterman) and overlap (ends-free) pairwise alignments while tuning *substitution scoring* and *gap penalties*.

Align the consensus ecdysone receptor response element to a region of chromosome 2L were such a motif is present:

```
pairwiseAlignment(EcR_IR1_cons,
                  Dmelanogaster$chr2L[14067:14101],
                  type='global-local')

## Global-Local PairwiseAlignmentsSingleSubject (1 of 1)
## pattern:  [1] AGTTCATTGACCTT
## subject: [12] AATTCATTGAAATG
## score: -3.779581
```

Align the consensus motifs for EcR and Usp on a TSS upstream sequence:

```
paln_EcRUsp=pairwiseAlignment(c(EcR_cons,Usp_cons),
                              dm3_upstream[[1780]],
                              type='global-local')
paln_EcRUsp[1]

## Global-Local PairwiseAlignmentsSingleSubject (1 of 1)
## pattern:   [1] AGGTCAT
## subject: [779] AGGGCAT
## score: 5.991251

paln_EcRUsp[2]

## Global-Local PairwiseAlignmentsSingleSubject (1 of 1)
## pattern:   [1] GGGGTCA
## subject: [772] GAGGTCA
## score: 5.991251

Views(paln_EcRUsp)

##   Views on a 2000-letter DNAString subject
## subject: CCATGCACTGGCCAGCGATAGCCCCATCTATCG...AGCCGCGAAATGAGGGGAAACCGAGCTGGAGC
## views:
##     start end width
## [1]   779 785     7 [AGGGCAT]
## [2]   772 778     7 [GAGGTCA]

Views(dm3_upstream[[1780]],start=772,end=785)

##   Views on a 2000-letter DNAString subject
## subject: CCATGCACTGGCCAGCGATAGCCCCATCTATCG...AGCCGCGAAATGAGGGGAAACCGAGCTGGAGC
## views:
##     start end width
## [1]   772 785    14 [GAGGTCAAGGGCAT]
```

Global vs local alignment:

```
pairwiseAlignment('GTGTCAATACGACAGCAATCTG',
                  'AGTGTGAATTACAGCAAATCTCTGTT',
                  type='global')

## Global PairwiseAlignmentsSingleSubject (1 of 1)
## pattern: [1] GTGTCAATACGACAGCAA---TCTG
## subject: [2] GTGTGAATT--ACAGCAAATCTCTG
## score: -48.12697

pairwiseAlignment('GTGTCAATACGACAGCAATCTG',
                  'AGTGTGAATTACAGCAAATCTCTGTT',
                  type='local')

## Local PairwiseAlignmentsSingleSubject (1 of 1)
## pattern: [12] ACAGCAA
## subject: [11] ACAGCAA
## score: 13.87229

pairwiseAlignment('GTGTCAATACGACAGCAATCTG',
                  'AGTGTGAATTACAGCAAATCTCTGTT',
                  type='global-local')

## Global-Local PairwiseAlignmentsSingleSubject (1 of 1)
## pattern: [1] GTGTCAATACGACAGCAA-TCTG
## subject: [2] GTGTGAATT--ACAGCAAATCTC
## score: -16.008
```

Playing with gap penalties:

```
pairwiseAlignment('GTGTCAATACGACAGCAATCTG',
                  'AGTGTGAATTACAGCAAATCTCTGTTCAATTTCTG',
                  type='global')

## Global PairwiseAlignmentsSingleSubject (1 of 1)
## pattern: [1] GTGTCAATACGACAGCAA--------------TCTG
## subject: [2] GTGTGAAT--TACAGCAAATCTCTGTTCAATTTCTG
## score: -74.12697

pairwiseAlignment('GTGTCAATACGACAGCAATCTG',
                  'AGTGTGAATTACAGCAAATCTCTGTTCAATTTCTG',
                  gapExtension=-6,type='global')

## Global PairwiseAlignmentsSingleSubject (1 of 1)
## pattern: [1] GTGTCAAT-ACGACAG---------CAAT--CTG
## subject: [2] GTGTGAATTACAGCAAATCTCTGTTCAATTTCTG
## score: -105.9255

pairwiseAlignment('GTGTCAATACGACAGCAATCTG',
                  'AGTGTGAATTACAGCAAATCTCTGTTCAATTTCTG',
                  gapOpening=-60,type='global')

## Global PairwiseAlignmentsSingleSubject (1 of 1)
## pattern:  [1] GTGTCAATACGACAGCAATCTG
## subject: [14] GCAAATCTCTGTTCAATTTCTG
## score: -186.617
```

Accessors and methods:

```
paln=pairwiseAlignment(
               DNAStringSet(c('GTGTCAATACGACAGCAATCTG',
                              'TAAGGTCATAGTGT')),
               DNAString('TCGCCATAGGTCAATAGTGTGAATTACAGCAAATCTCTGTTCAATTTCTG'),
                       type='global-local')
pattern(paln)

## QualityAlignedXStringSet (1 of 2)
## [1] GTGTCAATACGACAGCAA-TCTG

subject(paln)

## QualityAlignedXStringSet (1 of 2)
## [17] GTGTGAATT--ACAGCAAATCTC

aligned(paln)

##   A DNAStringSet instance of length 2
##     width seq
## [1]    50 ----------------GTGTCAATAACAGCAA-TCTG-------------
## [2]    50 -----TAAGGTC-ATAGTGT------------------------------

Biostrings::score(paln)

## [1] -16.008003  -2.017499

pid(paln) # percentage identity

## [1] 73.91304 80.00000

compareStrings(paln) #symbolic representation of the alignment

## [1] "GTGT?AAT?++ACAGCAA-TCT?" "??AGGTC-ATAGTGT"

nedit(paln) #Levenshtein edit distance

## [1] 6 3

nmatch(paln) #also nmismatch(paln)

## [1] 17 12

insertion(paln)[[1]] #also deletion(paln)

## IRanges object with 1 range and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]        10        11         2

Views(paln)

##   Views on a 50-letter DNAString subject
## subject: TCGCCATAGGTCAATAGTGTGAATTACAGCAAATCTCTGTTCAATTTCTG
## views:
##     start end width
## [1]    17  37    21 [GTGTGAATTACAGCAAATCTC]
## [2]     6  20    15 [ATAGGTCAATAGTGT]

coverage(paln)

## integer-Rle of length 50 with 5 runs
##   Lengths:  5 11  4 17 13
##   Values :  0  1  2  1  0
```

See the 'pairwise alignment' vignette of *Biostrings* for other utilities and accessors.

The stringDist function computes the Levenshtein edit distance or pairwise alignment score for a set of strings:

```
paln[1]

## Global-Local PairwiseAlignmentsSingleSubject (1 of 1)
## pattern:  [1] GTGTCAATACGACAGCAA-TCTG
## subject: [17] GTGTGAATT--ACAGCAAATCTC
## score: -16.008

nedit(paln[1])

## [1] 6

#Can also be obtained with:
stringDist(c('GTGTCAATACGACAGCAATCTG',
             'GTGTGAATTACAGCAAATCTC'),
           method = "levenshtein")

##   1
## 2 6
```

Using substitution matrices based on evolutionary models (other matrices are available):

```
data(BLOSUM62)
paln <- pairwiseAlignment(AAString("ALAKHLYDSYIKSFPLTKAKARAILTGKTTDKS"),
                          AAString("AQQFNDIVCAMTQEDLEKFWKRCSRPFTAHM"),
                          substitutionMatrix = BLOSUM62)
```

Visualizing the alignment with the DECIPHER package

```
library(DECIPHER)
alnseqs <- c(aligned(pattern(paln)), aligned(subject(paln)))
BrowseSeqs(alnseqs)
```

The *DECIPHER* package contains numerous functions that make use of *Biostrings* architecture to perform basic or advanced tasks with sequences including:

- Manipulate and analyze sequences
- Perform multiple sequence alignment (incl. on translated sequences)
- Compare and classify sequences
- Align whole genomes and find synteny
- design PCR primers, FISH probes or oligo for microarrays

2.4.2 Multiple alignment.

Several tools and algorithms are available to perform multiple sequence alignments (e.g. Multalin, ClustalW, T-Coffee or MUSCLE).
The *Biostrings* package essentially allows to import and explore the alignments obtained with different tools. Examples are provided in the vignette of *Biostrings* and here.
Additionally, the *msa* package [13] provides access, from within *R*, to the popular multiple alignment tools ClustalW, ClustalOmega and MUSCLE.

3 Manipulating genomic ranges

So far, we have essentially manipulated sequence data. However, we have already noticed that start-end coordinates along a sequence are also useful in some situations. These start-end coordinates define a range onto the sequence. As underlined by Martin Morgan: 1) ranges allow to represent a wide array of genomic data and annotations and 2) several biological questions reflect range-based queries. *Bioconductor* implements a number of tools to manipulate and analyze ranges and specifically genomic ranges [14].
In this section, we will mainly use the following 2 packages:

- *IRanges*
- *GenomicRanges*

We will also present the packages providing annotations as genomic ranges.

3.1 IRanges and accessors

Definition.
An *IRanges* object is defined as follow:

- defined by 2 vectors out of start, end, width (SEW ; $end = start + width - 1$)
- closed intervals (i.e. include end points)
- zero-width convention: $width \geq 0$; $end = start - 1 \Leftrightarrow width = 0$
- can be named

A simple IRanges:

```
eg = IRanges(start = c(1, 10, 20),
              end = c(4, 10, 19),
              names = c("A", "B", "C"))
eg

## IRanges object with 3 ranges and 0 metadata columns:
##         start       end     width
##     <integer> <integer> <integer>
##   A         1         4         4
##   B        10        10         1
##   C        20        19         0
```

A bigger IRanges:

```
set.seed(123) #For reproducibility
start = floor(runif(10000, 1, 1000))
end = start + floor(runif(10000, 0, 100))
ir = IRanges(start, end)
ir

## IRanges object with 10000 ranges and 0 metadata columns:
##               start       end     width
##           <integer> <integer> <integer>
##       [1]       288       319        32
##       [2]       788       820        33
##       [3]       409       496        88
##       [4]       883       915        33
##       [5]       940       952        13
##       ...       ...       ...       ...
##    [9996]       466       493        28
##    [9997]       899       984        86
##    [9998]       114       125        12
##    [9999]       571       596        26
##   [10000]       900       966        67
```

Accessors and methods.
IRanges accessors:

```
length(ir)

## [1] 10000

ir[1:4]

## IRanges object with 4 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]       288       319        32
##   [2]       788       820        33
##   [3]       409       496        88
##   [4]       883       915        33

start(ir[1:4])

## [1] 288 788 409 883

width(ir[1:4])

## [1] 32 33 88 33

names(eg)

## [1] "A" "B" "C"
```

Other useful methods for *IRanges*

```
c(ir[1:2],ir[5:6]) #combining

## IRanges object with 4 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]       288       319        32
##   [2]       788       820        33
##   [3]       940       952        13
##   [4]        46        81        36

sort(ir[1:4])

## IRanges object with 4 ranges and 0 metadata columns:
##           start       end     width
##         <integer> <integer> <integer>
##   [1]       288       319        32
##   [2]       409       496        88
##   [3]       788       820        33
##   [4]       883       915        33
```

```
rank(ir[1:4],ties="first")
```

```
## [1] 1 3 2 4
```

```
mid(ir[1:4]) # midpoints
```

```
## [1] 303 804 452 899
```

```
tile(ir[1:2],n=2) #returns an IRangesList (see below)
```

```
## IRangesList of length 2
## [[1]]
## IRanges object with 2 ranges and 0 metadata columns:
##             start       end     width
##         <integer> <integer> <integer>
##   [1]       288       303        16
##   [2]       304       319        16
##
## [[2]]
## IRanges object with 2 ranges and 0 metadata columns:
##             start       end     width
##         <integer> <integer> <integer>
##   [1]       788       803        16
##   [2]       804       820        17
```

```
ir[[1]]
```

```
##  [1] 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305
## [19] 306 307 308 309 310 311 312 313 314 315 316 317 318 319
```

```
as.integer(ir[1]) #equivalent but works on multiple ranges
```

```
##  [1] 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305
## [19] 306 307 308 309 310 311 312 313 314 315 316 317 318 319
```

```
unlist(ir[1]) #also equivalent but names can be added
```

```
##  [1] 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305
## [19] 306 307 308 309 310 311 312 313 314 315 316 317 318 319
```

```
rep(ir[1:2],each=2)
```

```
## IRanges object with 4 ranges and 0 metadata columns:
##             start       end     width
##         <integer> <integer> <integer>
##   [1]       288       319        32
##   [2]       288       319        32
##   [3]       788       820        33
##   [4]       788       820        33
```

```
isNormal(ir[1:4])
```

```
## [1] FALSE
```

```
isNormal(sort(ir[1:4])) #see ?'Ranges-class' for Normality definition

## [1] TRUE

isDisjoint(ir[1:4])

## [1] TRUE

match(ir[1:4],ir[4:1]) #see ?'Ranges-comparison' for Ranges comparison methods

## [1] 4 3 2 1

ir[1:4]>ir[4:1]

## [1] FALSE  TRUE FALSE  TRUE
```

Other methods creating IRanges:

```
as(c(2:10,8,90:100),"IRanges") #from a vector of integers

## IRanges object with 3 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         2        10         9
##   [2]         8         8         1
##   [3]        90       100        11

successiveIRanges(width=rep(10,5),gap=10)

## IRanges object with 5 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         1        10        10
##   [2]        21        30        10
##   [3]        41        50        10
##   [4]        61        70        10
##   [5]        81        90        10

whichAsIRanges(c(19, 5, 0, 8, 5)>=5) #transforms a logical vector in IRanges

## NormalIRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         1         2         2
##   [2]         4         5         2
```

IRangesList.
It can be convenient to group ranges in a list (e.g. exons grouped by genes). *IRangesList* objects serve this purpose. Accessors and functions for *IRanges* generally work on *IRangesList* objects.

```
irl=split(ir,width(ir)) # an IRangesList
irl[[1]]

## IRanges object with 96 ranges and 0 metadata columns:
##            start       end     width
##        <integer> <integer> <integer>
##    [1]       321       321         1
##    [2]       600       600         1
##    [3]       184       184         1
##    [4]       297       297         1
##    [5]       276       276         1
##    ...       ...       ...       ...
##   [92]       188       188         1
##   [93]       308       308         1
##   [94]       289       289         1
##   [95]       936       936         1
##   [96]       669       669         1

start(irl)

## IntegerList of length 100
## [["1"]] 321 600 184 297 276 816 87 729 407 ... 858 52 85 188 308 289 936 669
## [["2"]] 915 576 706 235 678 647 451 138 ... 638 66 979 740 300 869 433 645
## [["3"]] 457 415 336 774 487 787 587 352 ... 264 60 607 292 709 418 552 102
## [["4"]] 253 75 429 4 785 24 464 869 433 ... 827 240 175 459 130 146 681 267
## [["5"]] 498 812 977 120 991 583 959 931 532 ... 157 871 538 209 8 37 39 443
## [["6"]] 372 298 241 61 363 351 847 37 916 ... 910 929 656 71 839 223 34 602
## [["7"]] 753 198 674 584 850 250 962 14 ... 457 175 731 758 953 212 551 153
## [["8"]] 895 803 784 358 366 536 530 323 ... 350 959 592 651 487 406 895 53
## [["9"]] 318 478 991 747 769 626 332 436 ... 114 546 928 69 415 219 568 546
## [["10"]] 467 674 225 194 464 87 494 364 ... 706 267 938 259 685 700 435 937
## ...
## <90 more elements>

head(elementNROWS(irl))

##   1   2   3   4   5   6
##  96  83 108  95  84 110
```

3.2 Intra-and inter-range operations

3.2.1 Intra-range operations

These operations apply on each range of an *IRanges* object:

```
ir[1:2]

## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]       288       319        32
##   [2]       788       820        33

shift(ir[1:2],shift=10)

## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
## [1] 298 329 32
## [2] 798 830 33
```

```
resize(ir[1:2],width=100,fix="start")
```

```
## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]       288       387       100
##   [2]       788       887       100
```

```
flank(ir[1:2],width=100,start=T)
```

```
## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]       188       287       100
##   [2]       688       787       100
```

```
narrow(ir[1:2],start=1,width=30) #here 'start' is relative
```

```
## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]       288       317        30
##   [2]       788       817        30
```

```
ir[1:2]+10
```

```
## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]       278       329        52
##   [2]       778       830        53
```

```
ir[1:2]-10
```

```
## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]       298       309        12
##   [2]       798       810        13
```

```
ir[1:2]+c(0,10)
```

```
## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]       288       319        32
##   [2]       778       830        53
```

```
ir[1:4]*-10 ; ir[1:4]*10 # acts like a centered zoom
```

```
## IRanges object with 4 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]       144       463       320
##   [2]       639       968       330
##   [3]        13       892       880
##   [4]       734      1063       330
## IRanges object with 4 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]       302       304         3
##   [2]       803       805         3
##   [3]       449       456         8
##   [4]       898       900         3
```

```
ir[1:2]*c(1,2) #zoom second range by 2X
```

```
## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]       288       319        32
##   [2]       796       811        16
```

See `help('intra-range-methods',package="IRanges")` for other methods

3.2.2 Inter-range operations

Function to plot ranges (adapted from the *IRanges* vignette):

```
plotRanges <- function(x,
                       xlim = x,
                       main = deparse(substitute(x)),
                       col = "black", sep = 0.5,
                       cextitle=1, cexaxis=1, xaxis=T,...)
{
  height <- 1
  if (is(xlim, "Ranges"))
    xlim <- c(min(start(xlim)), max(end(xlim)))
  bins <- disjointBins(IRanges(start(x), end(x) + 1))
  plot.new()
  plot.window(xlim, c(0, max(bins)*(height + sep)))
  ybottom <- bins * (sep + height) - height
  rect(start(x)-0.5, ybottom, end(x)+0.5, ybottom + height, col = col, ...)
  if (xaxis)
    (axis(1,cex.axis=cexaxis,padj=1))
  title(main,cex.main=cextitle)
}
```

These inter-range operations, called *endomorphisms*, apply on a set of ranges and return a set of ranges:

```
#select some ranges in [1:100]:
irs=ir[which(start(ir)<=100 &
               end(ir)<=100)[c(3:4,8,14,17:18)]]
irs
```

```
## IRanges object with 6 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]        25        44        20
##   [2]        46        60        15
##   [3]        53        65        13
##   [4]         9        33        25
##   [5]         1        50        50
##   [6]        87        96        10
```

```
reduce(irs)
```

```
## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         1        65        65
##   [2]        87        96        10
```

```
disjoin(irs)
```

```
## IRanges object with 10 ranges and 0 metadata columns:
##            start       end     width
##        <integer> <integer> <integer>
##    [1]         1         8         8
##    [2]         9        24        16
##    [3]        25        33         9
##    [4]        34        44        11
##    [5]        45        45         1
##    [6]        46        50         5
##    [7]        51        52         2
##    [8]        53        60         8
##    [9]        61        65         5
##   [10]        87        96        10
```

```
gaps(irs)
```

```
## IRanges object with 1 range and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]        66        86        21
```

```
coverage(irs)
```

```
## integer-Rle of length 96 with 11 runs
##   Lengths:  8 16  9 11  1  5  2  8  5 21 10
##   Values :  1  2  3  2  1  2  1  2  1  0  1
```

See `help('inter-range-methods',package="IRanges")` for other methods.

They are illustrated in Figure 9 using:

```
par(mfrow=c(5,1))
plotRanges(irs,xlim=c(0,100),xaxis=T, cextitle=2, cexaxis=1.5)
plotRanges(reduce(irs),xlim=c(0,100),xaxis=T, cextitle=2, cexaxis=1.5)
plotRanges(disjoin(irs),xlim=c(0,100),xaxis=T, cextitle=2, cexaxis=1.5)
plotRanges(gaps(irs),xlim=c(0,100),xaxis=T, cextitle=2, cexaxis=1.5)
plot(1:100,c(coverage(irs),rep(0,4)),type="l",axes=F,xlab="",ylab="",lwd=3)
title(main="coverage",cex.main=2)
axis(side=2,lwd=2,cex.axis=2,at=0:3,labels=0:3)
axis(1,lwd=2,cex.axis=2,padj=1)
```

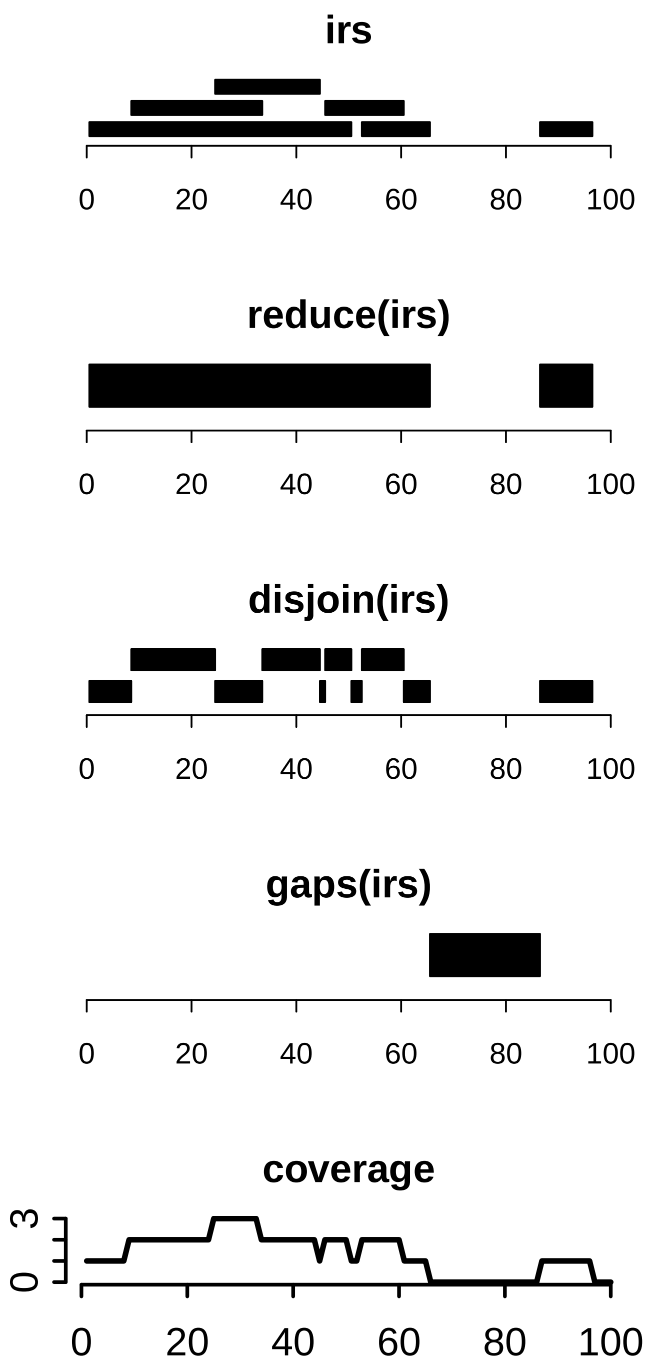


Figure 9: Inter-range operations

3.2.3 Set operations

See `help('setops-methods',package="IRanges")` for details. The `union` and `intersect` functions for *IRanges*:

```
union(irs[1:3],irs[4:6])

## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         1        65        65
##   [2]        87        96        10

intersect(irs[1:3],irs[4:6])

## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]        25        44        20
##   [2]        46        50         5
```

Illustrated in Figure 10:

```
par(mfrow=c(4,1))
 plotRanges(irs[1:3], xlim=c(0,100), xaxis=T,
            cextitle=2, cexaxis=1.5)
 plotRanges(irs[4:6], xlim=c(0,100), xaxis=T,
            cextitle=2, cexaxis=1.5)
 plotRanges(union(irs[1:3], irs[4:6]), xlim=c(0,100), xaxis=T,
            cextitle=2, cexaxis=1.5)
 plotRanges(intersect(irs[1:3], irs[4:6]), xlim=c(0,100), xaxis=T,
            cextitle=2, cexaxis=1.5)
```

The `setdiff` function for asymmetric differences:

```
setdiff(irs[1:3],irs[4:6])

## IRanges object with 1 range and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]        51        65        15

setdiff(irs[4:6],irs[1:3])

## IRanges object with 3 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         1        24        24
##   [2]        45        45         1
##   [3]        87        96        10
```

Illustrated in Figure 11:

```
par(mfrow=c(4,1))
 plotRanges(irs[1:3],xlim=c(0,100),xaxis=T, cextitle=2, cexaxis=1.5)
 plotRanges(irs[4:6],xlim=c(0,100),xaxis=T, cextitle=2, cexaxis=1.5)
```

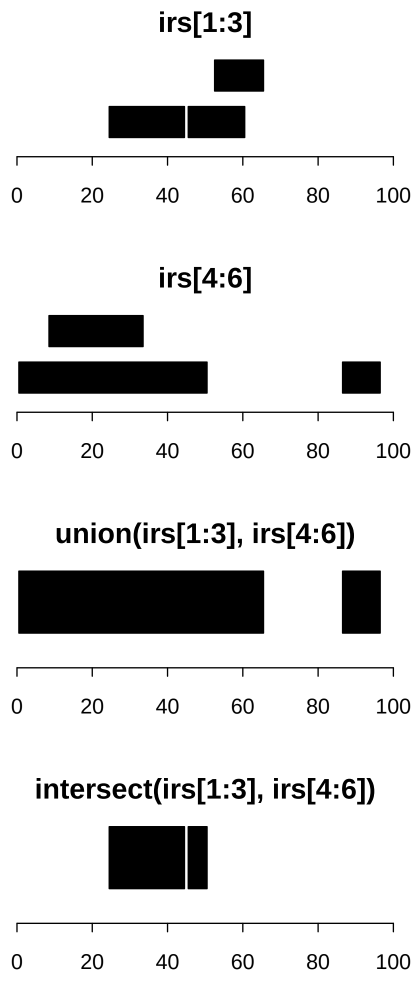


Figure 10: Union and intersect on IRanges

```
plotRanges(setdiff(irs[1:3],irs[4:6]),xlim=c(0,100),xaxis=T,
           cextitle=2, cexaxis=1.5)
plotRanges(setdiff(irs[4:6],irs[1:3]),xlim=c(0,100),xaxis=T,
           cextitle=2, cexaxis=1.5)
```

The same functions can be applied in a parallel fashion (i.e. on the first elements of the provided IRanges, then on the second elements, etc.)

```
punion(irs[1:2],irs[4:5]) #element-wise (aka "parallel") union

## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         9        44        36
##   [2]         1        60        60

pintersect(irs[1:2],irs[4:5])

## IRanges object with 2 ranges and 0 metadata columns:
```

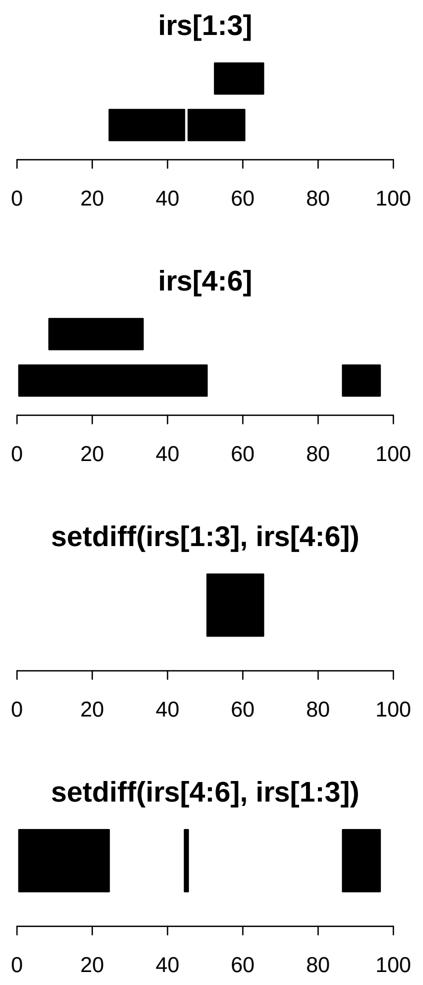


Figure 11: Asymetric differences with setdiff on IRanges

```
##           start       end     width
##       <integer> <integer> <integer>
##   [1]        25        33         9
##   [2]        46        50         5

psetdiff(irs[1:3],irs[4:6]) # asymmetric! difference

## IRanges object with 3 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]        34        44        11
##   [2]        51        60        10
##   [3]        53        65        13

pgap(irs[1:3],irs[4:6])

## IRanges object with 3 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]        34         33          0
##   [2]        51         50          0
##   [3]        66         86         21
```

3.2.4 Nearest methods

See `help('nearest-methods',package="IRanges")` for details and examples:

```
nearest(irs[4:6],irs[1:3])

## [1] 1 1 3

distance(irs[4:6],irs[1:3])

## [1]  0  0 21
```

Other examples are provided in paragraph 3.5.4

Between ranges operations.
These are mainly methods to find overlaps between ranges. These functions are exampli-fied below in paragraph 3.5.5. See `help('findOverlaps-methods',package="IRanges")` for details.

3.3 Rle

Run-length encoding is a data compression method highly adapted to long vectors containing repeated values (e.g. coverage on a whole chromosome). For example, the sequence $\{1, 1, 1, 2, 3, 3\}$ can be represented as $values = \{1, 2, 3\}$ and the paired $runlengths = \{3, 1, 2\}$. In *IRanges*, the *Rle* class is used to represent run-length encoded (compressed) atomic vectors.

Rle objects:

```
set.seed(123)
lambda = c(rep(0.001, 3500), #From IRanges vignette
           seq(0.001, 10, length = 500),
            seq(10, 0.001, length = 500))
xRle=Rle(rpois(1e4,lambda))
yRle=Rle(rpois(1e4, lambda[c(251:length(lambda), 1:250)]))
xRle

## integer-Rle of length 10000 with 1630 runs
##   Lengths:  326    1 1150    1  772    1 ...    1  170    1  466    1  262
##   Values :    0    1    0    1    0    1 ...    1    0    1    0    1    0

yRle
```

```
## integer-Rle of length 10000 with 1616 runs
##   Lengths: 1984    1 1268    1   15    1 ...    1    1    1    8    1 1264
##   Values :    0    1    0    1    0    1 ...    1    0    1    0    1    0

as.vector(object.size(xRle)/object.size(as.vector(xRle))) #Gain of memory

## [1] 0.3532468

head(runValue(xRle))

## [1] 0 1 0 1 0 1

head(runLength(xRle))

## [1]  326    1 1150    1  772    1

head(start(xRle)) #starts of the runs

## [1]    1  327  328 1478 1479 2251

head(end(xRle)) #ends of the runs

## [1]  326  327 1477 1478 2250 2251

nrun(xRle) #number of runs

## [1] 1630

findRun(as.integer(c(100,200,300,1200)),xRle)

## [1] 1 1 1 3

coverage(irs)

## integer-Rle of length 96 with 11 runs
##   Lengths:  8 16  9 11  1  5  2  8  5 21 10
##   Values :  1  2  3  2  1  2  1  2  1  0  1
```

These objects support a number of basic methods associated with R atomic vectors:

```
xRle+yRle

## integer-Rle of length 10000 with 2111 runs
##   Lengths:  326    1 1150    1  506    1 ...    1  170    1  466    1  262
##   Values :    0    1    0    1    0    1 ...    1    0    1    0    1    0

xRle>0

## logical-Rle of length 10000 with 211 runs
##   Lengths:   326     1  1150     1   772 ...   170     1   466     1   262
##   Values : FALSE  TRUE FALSE  TRUE FALSE ... FALSE  TRUE FALSE  TRUE FALSE

xRle>yRle

## logical-Rle of length 10000 with 367 runs
##   Lengths:   326     1  1150     1   772 ...   170     1   466     1   262
##   Values : FALSE  TRUE FALSE  TRUE FALSE ... FALSE  TRUE FALSE  TRUE FALSE

max(xRle)

## [1] 18

summary(xRle)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.0000  0.0000  0.9932  0.0000 18.0000

sqrt(xRle)

## numeric-Rle of length 10000 with 1630 runs
##   Lengths:              326                1 ...                262
##   Values :                0                1 ...                  0

rev(xRle)

## integer-Rle of length 10000 with 1630 runs
##   Lengths:  262    1  466    1  170    1 ...    1  772    1 1150    1  326
##   Values :    0    1    0    1    0    1 ...    1    0    1    0    1    0

table(xRle)

## xRle
##    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14
## 8200  201  198  202  212  184  167  146  140  101   89   49   51   25   12
##   15   16   17   18
##    6    7    8    2

union(xRle,yRle)

## integer-Rle of length 20 with 20 runs
##   Lengths:  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
##   Values :  0  1  3  2  4  5  6  7  8  9 11 12 10 13 17 14 16 15 18 22

cor(xRle,yRle)

## [1] 0.5690394
```

See `?'Rle-class'` and `?'Rle-utils'` for other methods.

There are useful functions to perform fixed-width running window summaries:

```
runmean(xRle,k=100) # See ?'Rle-runstat' for other examples

## numeric-Rle of length 9901 with 1852 runs
##   Lengths:  227  100 1051  100  673  100 ...  100   71  100  367  100  163
##   Values :    0 0.01    0 0.01    0 0.01 ... 0.01    0 0.01    0 0.01    0

#same result, more flexible but much slower:
Rle(aggregate(xRle, start = 1:(length(xRle)-99), width = 100, FUN = mean))

## numeric-Rle of length 9901 with 1852 runs
##   Lengths:  227  100 1051  100  673  100 ...  100   71  100  367  100  163
##   Values :    0 0.01    0 0.01    0 0.01 ... 0.01    0 0.01    0 0.01    0

runq(xRle,k=100,i=10) #10th smallest value in windows of 100

## integer-Rle of length 9901 with 41 runs
##   Lengths: 3558    7    2    6    4  102 ...    2    2   80   59   53 1090
##   Values :    0    1    0    1    0    1 ...    3    4    3    2    1    0
```

One typical application of *Rle* objects is to store the coverage of NGS reads along a chromosome. The coverage for all chromosomes can be stored in an *RleList* (see `?AtomicList` for details) on which most functions defined for *Rle* objects would also work.

Practically, any variable defined along a genome can be represented as

- an *RleList* with one *Rle* for each chromosome
- the mcols (metadata columns) of a *GRanges* object (see 3.5 below)

Sometimes, it is desirable to manipulate several of these variables in the same object (e.g. for plotting with *Gviz*). The `?genomicvars` help page provides usefull functions such as `bindAsGRanges` and `mcolAsRleList` to go from one representation to the other.

3.4 Views

As already mentioned, *Views* objects are used to store a sequence (a *Vector* object called the "subject") together with a set of ranges which define *views* onto the sequence. Specific subclasses exist for different classes of "subject" Vectors, such as *RleViews* from the *IRanges* package and *XStringViews* from the *Biostrings* package.

An *RleViews* object stores an *Rle* subject and its *views*:

```
coverage(irs)

## integer-Rle of length 96 with 11 runs
##   Lengths:  8 16  9 11  1  5  2  8  5 21 10
##   Values :  1  2  3  2  1  2  1  2  1  0  1

irs_views=Views(coverage(irs),start=c(-5,10,20,90),end=c(10,30,50,100))
irs_views #Views can be out of bound

## Views on a 96-length Rle subject
##
## views:
##     start end width
## [1]    -5  10    16 [1 1 1 1 1 1 1 1 2 2 ...]
## [2]    10  30    21 [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3]
## [3]    20  50    31 [2 2 2 2 2 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 ...]
## [4]    90 100    11 [1 1 1 1 1 1 1 ...]

try(irs_views[[1]]) #but can't be extracted
irs_views[[2]]

## integer-Rle of length 21 with 2 runs
##   Lengths: 15  6
##   Values :  2  3

start(irs_views)

## [1] -5 10 20 90
```

See`?'RleViews-class'` for details.

Other ways to create Views:

```
Views(coverage(irs),irs[c(1,2,6)]) #use an IRanges to extract Views

## Views on a 96-length Rle subject
##
## views:
##     start end width
## [1]    25  44    20 [3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2]
## [2]    46  60    15 [2 2 2 2 2 1 1 2 2 2 2 2 2 2 2]
## [3]    87  96    10 [1 1 1 1 1 1 1 1 1 1]

Views(coverage(irs),coverage(irs)>=1) #or a logical Rle

## Views on a 96-length Rle subject
##
## views:
##     start end width
## [1]     1  65    65 [1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 ...]
## [2]    87  96    10 [1 1 1 1 1 1 1 1 1 1]

slice(coverage(irs),3) #use slice

## Views on a 96-length Rle subject
##
## views:
##     start end width
## [1]    25  33     9 [3 3 3 3 3 3 3 3 3]

successiveViews(coverage(irs),width=rep(20,4)) #get successive Views

## Views on a 96-length Rle subject
##
## views:
##     start end width
## [1]     1  20    20 [1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2]
## [2]    21  40    20 [2 2 2 2 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2]
## [3]    41  60    20 [2 2 2 2 1 2 2 2 2 2 1 1 2 2 2 2 2 2 2 2]
## [4]    61  80    20 [1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

XStringViews is used to store *views* on an *XString* object:

```
dmup_views=Views(dm3_upstream[[1]],irs[1:2])
dmup_views

##   Views on a 2000-letter DNAString subject
## subject: GTTGGTGGCCCACCAGTGCCAAAATACACAAGA...CGTATAAAAGGCAAGTTTACCGGTTGCACGGT
## views:
##     start end width
## [1]    25  44    20 [TACACAAGAAGAAGAAACAG]
## [2]    46  60    15 [ATCTTGACACTAAAA]

nchar(dmup_views)

## [1] 20 15

toString(dmup_views) #see ?'XStringViews-class' for other methods
```

```
## [1] "TACACAAGAAGAAGAAACAG, ATCTTGACACTAAAA"
```

Furthermore there is a *ViewsList* virtual class. Its specialized subclass *RleViewsList* is useful to store coverage vectors along with their specific *views* over a set of chromosomes.

```
xyRleList=RleList(xRle,yRle)
xyRleList_views=Views(xyRleList,
                      IRangesList(IRanges(start=c(3700,4000),width=20),
                                  IRanges(yRle>17)+2))
xyRleList_views[[1]]

## Views on a 10000-length Rle subject
##
## views:
##     start  end width
## [1]  3700 3719    20 [1 0 2 7 2 9 5 3 3 6 4 4 3 7 5 5 3 5 8 1]
## [2]  4000 4019    20 [ 9 13 13  8  7 14  3 10 16  8 17  5  8  8 15 17 ...]

xyRleList_views[[2]]

## Views on a 10000-length Rle subject
##
## views:
##     start  end width
## [1]  3750 3754     5 [10 13 18 12 14]
## [2]  8250 8254     5 [12  8 22 10  9]

width(xyRleList_views)

## IntegerList of length 2
## [[1]] 20 20
## [[2]] 5 5
```

Specific functions are provided for fast looping over *Views* and *ViewsList* objects:

```
viewMins(irs_views) #same as min(irs_views)

## [1] 1 2 1 1

viewSums(irs_views) #same as sum(irs_views)

## [1] 12 48 70  7

viewWhichMaxs(irs_views) #get the (first) coordinate of viewMaxs

## [1]  9 25 25 90

# (which.max also works)
viewRangeMins(irs_views) #get the (first) range of viewMins

## IRanges object with 4 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         1         8         8
##   [2]         9        24        16
##   [3]        45        45         1
##   [4]        87        96        10
```

```
viewApply(irs_views,sd)

## [1] 0.4216370 0.4629100 0.5143113 0.0000000

viewMeans(xyRleList_views)

## NumericList of length 2
## [[1]] 4.15 10.35
## [[2]] 13.4 12.2
```

Note that the `min`,`max`,`sum`,`mean`,`which.min` and `which.max` functions now work on *Views* (but not on *RleViewsList* yet). The corresponding *view** functions might be deprecated in the future. See `?'view-summarization-methods'` for details.

3.5 GenomicRanges

3.5.1 GRanges objects

Here, we will present some of the classes and functions defined in the *GenomicRanges* package. This package is central to *Bioconductor* users who analyze NGS data, so you should consider reading thoroughly the excellent vignettes associated with this package.
The main class defined in *GenomicRanges* is the *GRanges* class which acts as a container for genomic locations and their associated annotations (see `?GRanges`).

GRanges (and *GRangesList*) build on *IRanges* (and *IRangesList* respectively) with the following specificities:

- The informations on `seqnames` (typically chromosomes) and `strand` is stored along with the information on ranges (SEW)
- An optional `seqinfo` slot contains information on the sequences: names, length (`seqlengths`), circularity and genome
- Optional *metadata* columns (`mcols`) containing additional informations on each range (e.g. score, GC content, etc.) which are stored as a *DataFrame*

By convention, in *Bioconductor* genomic coordinates:

- are 1-based
- are *left-most*, i.e. 'start' of ranges on the minus strand are the left-most coordinate, rather than the 5' coordinate
- represent closed intervals, i.e. the intervals contain start and end coordinates

The `GRanges` function can be used to create a *GRanges* object:

```
genes = GRanges(seqnames=c("chr2L", "chrX"),
                ranges=IRanges(start=c(7529, 18962306),
                               end =c(9484, 18962925),
                               names=c("FBgn0031208", "FBgn0085359")),
                strand=c("+", "-"),
                seqlengths=c(chr2L=23011544L, chrX=22422827L))

slotNames(genes)

## [1] "seqnames"        "ranges"          "strand"          "elementMetadata"
## [5] "seqinfo"         "metadata"

mcols(genes) = DataFrame(EntrezId=c("33155", "2768869"),
                         Symbol=c("CG11023", "CG34330"))
genome(genes)="dm3" #see ?seqinfo for details
genes

## GRanges object with 2 ranges and 2 metadata columns:
##               seqnames               ranges strand |    EntrezId      Symbol
##                  <Rle>            <IRanges>  <Rle> | <character> <character>
##   FBgn0031208    chr2L [    7529,     9484]      + |       33155     CG11023
##   FBgn0085359     chrX [18962306, 18962925]      - |     2768869     CG34330
##   -------
##   seqinfo: 2 sequences from dm3 genome
```

The *GRanges* accessors include *IRanges* accessors and others:

```
width(genes)

## [1] 1956  620

names(genes)

## [1] "FBgn0031208" "FBgn0085359"

seqnames(genes)

## factor-Rle of length 2 with 2 runs
##   Lengths:     1     1
##   Values : chr2L  chrX
## Levels(2): chr2L chrX

strand(genes)

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

ranges(genes)

## IRanges object with 2 ranges and 0 metadata columns:
##                   start       end     width
##               <integer> <integer> <integer>
##   FBgn0031208      7529      9484      1956
##   FBgn0085359  18962306  18962925       620

genes$Symbol

## [1] "CG11023" "CG34330"

mcols(genes)
```

```
## DataFrame with 2 rows and 2 columns
##      EntrezId      Symbol
##   <character> <character>
## 1       33155     CG11023
## 2     2768869     CG34330
```

```
seqinfo(genes)
```

```
## Seqinfo object with 2 sequences from dm3 genome:
##   seqnames seqlengths isCircular genome
##   chr2L      23011544         NA    dm3
##   chrX       22422827         NA    dm3
```

```
seqlevels(genes)
```

```
## [1] "chr2L" "chrX"
```

3.5.2 GRangesList

As for *IRangesList*, there is a *GRangesList* class which allows to store *GRanges* in a list-type object. This is typically used to store e.g. *GRanges* of exons arranged by transcripts or genes or *GRanges* of transcripts arranged by genes. The vignette of the *GenomicRanges* package provides a good example of 2 transcripts, one of which has 2 exons:

```
gr1 = GRanges(seqnames = "chr2", ranges = IRanges(3, 6),
              strand = "+", score = 5L, GC = 0.45)
gr2 = GRanges(seqnames = c("chr1", "chr1"),
              ranges = IRanges(c(7,13), width = 3),
              strand = c("+", "-"), score = 3:4, GC = c(0.3, 0.5))
grl = GRangesList("txA" = gr1, "txB" = gr2)
grl
```

```
## GRangesList object of length 2:
## $txA
## GRanges object with 1 range and 2 metadata columns:
##       seqnames    ranges strand |     score        GC
##          <Rle> <IRanges>  <Rle> | <integer> <numeric>
##   [1]     chr2    [3, 6]      + |         5      0.45
##
## $txB
## GRanges object with 2 ranges and 2 metadata columns:
##       seqnames   ranges strand | score  GC
##   [1]     chr1 [ 7,  9]      + |     3 0.3
##   [2]     chr1 [13, 15]      - |     4 0.5
##
## -------
## seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

```
length(grl)
```

```
## [1] 2
```

```
elementNROWS(grl)
```

```
## txA txB
##   1   2
```

```
grl["txB","GC"]
```

```
## GRangesList object of length 1:
## $txB
## GRanges object with 2 ranges and 1 metadata column:
##       seqnames    ranges strand |        GC
##          <Rle> <IRanges>  <Rle> | <numeric>
##   [1]     chr1  [ 7,  9]      + |       0.3
##   [2]     chr1  [13, 15]      - |       0.5
##
## -------
## seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

```
unlist(grl)
```

```
## GRanges object with 3 ranges and 2 metadata columns:
##       seqnames    ranges strand |     score        GC
##          <Rle> <IRanges>  <Rle> | <integer> <numeric>
##   txA     chr2  [ 3,  6]      + |         5      0.45
##   txB     chr1  [ 7,  9]      + |         3       0.3
##   txB     chr1  [13, 15]      - |         4       0.5
##   -------
##   seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

Please refer to the vignette for further examples.
Most accessors and functions defined for *GRanges* also work on *GRangesList*. However, note that `mcols(grl)` now refers to metadata at the list level rather than at level of the individual *GRanges* objects.

3.5.3 Annotations as GenomicRanges: TxDb* packages

Here, we will briefly present the *Bioconductor* annotation packages which provide genome-wide annotations directly as *GRanges* and *GRangesList* objects. These packages, called *TxDb** can be built using the *GenomicFeatures* package.
Here, we show how to extract information from the *TxDb.Dmelanogaster.UCSC.dm3.ensGene* package.
A *GRanges* object with the genomic coordinates for the genes can be obtained using:

```
Dmg=genes(TxDb.Dmelanogaster.UCSC.dm3.ensGene,single.strand.genes.only=T)
Dmg
```

```
## GRanges object with 15682 ranges and 1 metadata column:
##               seqnames               ranges strand |     gene_id
##                  <Rle>            <IRanges>  <Rle> | <character>
##   FBgn0000003    chr3R [ 2648220,  2648518]      + | FBgn0000003
##   FBgn0000008    chr2R [18024494, 18060346]      + | FBgn0000008
##   FBgn0000014    chr3R [12632936, 12655767]      - | FBgn0000014
##   FBgn0000015    chr3R [12752932, 12797958]      - | FBgn0000015
##   FBgn0000017    chr3L [16615470, 16640982]      - | FBgn0000017
##           ...      ...                  ...    ... .         ...
##   FBgn0264723    chr3L [12238610, 12239148]      - | FBgn0264723
##   FBgn0264724    chr3L [15327882, 15329271]      + | FBgn0264724
##   FBgn0264725    chr3L [12025657, 12026099]      + | FBgn0264725
##   FBgn0264726    chr3L [12020901, 12021253]      + | FBgn0264726
##   FBgn0264727    chr3L [22065469, 22065720]      + | FBgn0264727
##   -------
##   seqinfo: 15 sequences (1 circular) from dm3 genome
```

Other functions, such as `transcripts`, `exons`, `cds`, `promoters`, `microRNAs` and `tRNAs` allow to extract the corresponding genomic features.

A *GRangesList* object with the coordinates of the transcripts arranged by genes can be obtained using:

```
Dmt=transcriptsBy(TxDb.Dmelanogaster.UCSC.dm3.ensGene,by="gene")
Dmt

## GRangesList object of length 15682:
## $FBgn0000003
## GRanges object with 1 range and 2 metadata columns:
##       seqnames             ranges strand |     tx_id     tx_name
##          <Rle>          <IRanges>  <Rle> | <integer> <character>
##   [1]    chr3R [2648220, 2648518]      + |     17345 FBtr0081624
##
## $FBgn0000008
## GRanges object with 3 ranges and 2 metadata columns:
##       seqnames               ranges strand | tx_id     tx_name
##   [1]    chr2R [18024494, 18060339]      + |  7681 FBtr0100521
##   [2]    chr2R [18024496, 18060346]      + |  7682 FBtr0071763
##   [3]    chr2R [18024938, 18060346]      + |  7683 FBtr0071764
##
## $FBgn0000014
## GRanges object with 4 ranges and 2 metadata columns:
##       seqnames               ranges strand | tx_id     tx_name
##   [1]    chr3R [12632936, 12655767]      - | 21863 FBtr0306337
##   [2]    chr3R [12633349, 12653845]      - | 21864 FBtr0083388
##   [3]    chr3R [12633349, 12655300]      - | 21865 FBtr0083387
##   [4]    chr3R [12633349, 12655474]      - | 21866 FBtr0300485
##
## ...
## <15679 more elements>
## -------
## seqinfo: 15 sequences (1 circular) from dm3 genome
```

The functions `exonsBy`, `cdsBy`, `intronsByTranscript`, `fiveUTRsByTranscript`, `threeUTRsByTranscript` allow to extract the corresponding genomic features arranged in a *GRangesList* object.

Other functions including `transcriptsByOverlaps` and `exonsByOverlaps` allow to extract genomic features for genomic locations specified by a *GRanges* object:

```
exonsByOverlaps(TxDb.Dmelanogaster.UCSC.dm3.ensGene,genes)

## GRanges object with 6 ranges and 1 metadata column:
##       seqnames               ranges strand |   exon_id
##          <Rle>            <IRanges>  <Rle> | <integer>
##   [1]    chr2L [    7529,     8116]      + |         1
##   [2]    chr2L [    8193,     8589]      + |         2
##   [3]    chr2L [    8193,     9484]      + |         3
##   [4]    chr2L [    8229,     9484]      + |         4
##   [5]    chr2L [    8668,     9484]      + |         5
##   [6]     chrX [18962306, 18962925]      - |     75268
##   -------
##   seqinfo: 15 sequences (1 circular) from dm3 genome
```

When a *TxDb* package is paired with an appropriate *BSgenome* object, it is relatively straightforward to extract DNA sequences providing a *GRanges* or a *GRangesList*

```
getSeq(BSgenome.Dmelanogaster.UCSC.dm3,Dmg[1:2])

##   A DNAStringSet instance of length 2
##     width seq                                               names
## [1]   299 TCGACTGGAAGGTTGGCAGCTT...GATATGGTTGGACCACAATCT FBgn0000003
## [2] 35853 CGCGGCGGTCGCATCGGAGTCG...TTTACCTGAAAAGCAATATAC FBgn0000008

Dmc=cdsBy(TxDb.Dmelanogaster.UCSC.dm3.ensGene,by="tx")
cds_seq=extractTranscriptSeqs(BSgenome.Dmelanogaster.UCSC.dm3,Dmc[1:2])
cds_seq

##   A DNAStringSet instance of length 2
##     width seq                                               names
## [1]   855 ATGGGCGAGCGGGATCAGCCAC...GTATGGCAACGAATATATTGA 1
## [2]  1443 ATGGGCGAGCGGGATCAGCCAC...ATCGTCGACGGAGAGTTGTGA 2

translate(cds_seq)

##   A AAStringSet instance of length 2
##     width seq                                               names
## [1]   285 MGERDQPQSSERISIFNPPVYT...HDRFNEITQDDKSTVWQRIY* 1
## [2]   481 MGERDQPQSSERISIFNPPVYT...QSEMLYFRKKMALEIVDGEL* 2
```

3.5.4 GRanges methods

Most if not all functions defined for *IRanges* objects are also defined for *GRanges* and *GRangesLIst* objects and they are generally *seqnames*- and *strand*-aware. These include:

- intra-range operations (`shift`, etc. ; see paragraph 3.2.1 and `help('intra-range-methods',"GenomicRanges")`)
- inter-range operations (`reduce`, etc. ; see paragraph 3.2.2 and `help('inter-range-methods',"GenomicRanges")`)
- set operations (see paragraph 3.2.3 and `help('setops-methods',"GenomicRanges")`)
- nearest methods (`nearest`, etc. ; see paragraph 3.2.4 and `help('nearest-methods',"GenomicRanges")`)

- between ranges ("overlaps") operations presented below in paragraph 3.5.5

Here, we briefly illustrate some of these functions:

```
genes

## GRanges object with 2 ranges and 2 metadata columns:
##               seqnames               ranges strand |    EntrezId      Symbol
##                  <Rle>            <IRanges>  <Rle> | <character> <character>
##   FBgn0031208    chr2L [    7529,     9484]      + |       33155     CG11023
##   FBgn0085359     chrX [18962306, 18962925]      - |     2768869     CG34330
##   -------
##   seqinfo: 2 sequences from dm3 genome

genes2=Dmg[c(1:2,21:22,36:37)]
genes2

## GRanges object with 6 ranges and 1 metadata column:
##               seqnames               ranges strand |     gene_id
##                  <Rle>            <IRanges>  <Rle> | <character>
##   FBgn0000003    chr3R [ 2648220,  2648518]      + | FBgn0000003
##   FBgn0000008    chr2R [18024494, 18060346]      + | FBgn0000008
##   FBgn0000052    chr2L [ 6041178,  6045970]      - | FBgn0000052
##   FBgn0000053    chr2L [ 7014861,  7023940]      - | FBgn0000053
##   FBgn0000084     chrX [20065478, 20067552]      - | FBgn0000084
##   FBgn0000092     chrX [ 2586765,  2587919]      - | FBgn0000092
##   -------
##   seqinfo: 15 sequences (1 circular) from dm3 genome

sort(genes2)

## GRanges object with 6 ranges and 1 metadata column:
##               seqnames               ranges strand |     gene_id
##                  <Rle>            <IRanges>  <Rle> | <character>
##   FBgn0000052    chr2L [ 6041178,  6045970]      - | FBgn0000052
##   FBgn0000053    chr2L [ 7014861,  7023940]      - | FBgn0000053
##   FBgn0000008    chr2R [18024494, 18060346]      + | FBgn0000008
##   FBgn0000003    chr3R [ 2648220,  2648518]      + | FBgn0000003
##   FBgn0000092     chrX [ 2586765,  2587919]      - | FBgn0000092
##   FBgn0000084     chrX [20065478, 20067552]      - | FBgn0000084
##   -------
##   seqinfo: 15 sequences (1 circular) from dm3 genome

c(genes,genes2,ignore.mcols=T) #combine

## GRanges object with 8 ranges and 0 metadata columns:
##               seqnames               ranges strand
##                  <Rle>            <IRanges>  <Rle>
##   FBgn0031208    chr2L [    7529,     9484]      +
##   FBgn0085359     chrX [18962306, 18962925]      -
##   FBgn0000003    chr3R [ 2648220,  2648518]      +
##   FBgn0000008    chr2R [18024494, 18060346]      +
##   FBgn0000052    chr2L [ 6041178,  6045970]      -
##   FBgn0000053    chr2L [ 7014861,  7023940]      -
##   FBgn0000084     chrX [20065478, 20067552]      -
##   FBgn0000092     chrX [ 2586765,  2587919]      -
##   -------
##   seqinfo: 15 sequences (1 circular) from dm3 genome

intersect(genes,c(genes2,Dmg['FBgn0031208'])) #set operations

## GRanges object with 1 range and 0 metadata columns:
##       seqnames       ranges strand
##          <Rle>    <IRanges>  <Rle>
##   [1]    chr2L [7529, 9484]      +
##   -------
##   seqinfo: 15 sequences (1 circular) from dm3 genome

nearest(genes,genes2) #nearest-methods

## [1] NA  5

nearest(genes,genes2,ignore.strand=T) #strand-aware by default

## [1] 3 5

precede(genes,genes2,ignore.strand=T)

## [1] 3 5

promoters(genes,upstream=200,downstream=1) #intra-range operations

## GRanges object with 2 ranges and 2 metadata columns:
##               seqnames               ranges strand |    EntrezId      Symbol
##                  <Rle>            <IRanges>  <Rle> | <character> <character>
##   FBgn0031208    chr2L [    7329,     7529]      + |       33155     CG11023
##   FBgn0085359     chrX [18962925, 18963125]      - |     2768869     CG34330
##   -------
##   seqinfo: 2 sequences from dm3 genome

reduce(Dmt[[2]]) #inter-range operations

## GRanges object with 1 range and 0 metadata columns:
##       seqnames               ranges strand
##          <Rle>            <IRanges>  <Rle>
##   [1]    chr2R [18024494, 18060346]      +
##   -------
##   seqinfo: 15 sequences (1 circular) from dm3 genome
```

See also `help('GenomicRanges-comparison',"GenomicRanges")` for other functions for comparing *GenomicRanges*.

3.5.5 Overlaps between ranges

A very common task on genomic ranges is to search for overlaps between sets of genomic ranges which corresponds to an operation between ranges. These functions are defined for several classes of objects, including *IRanges*, *GRanges* and their list-type counterparts *IRangesList* and *GRangesList* but also *Views* and *ViewsList* among others. Details on function definitions can be found using `?'findOverlaps-methods'`.

As first example, let's count how many of the transcription start sites (TSS) in the Drosophila melanogaster genome are located at more than 500bp from another gene:

```
Dm_tss=unlist(reduce(promoters(Dmt,up=0,down=1))) #get all TSS
cov_tss_g500=countOverlaps(Dm_tss,Dmg+500) #strand-aware!
table(cov_tss_g500)

## cov_tss_g500
##     1     2     3     4     5     6     7     8     9
## 15586  4488   490   116    65    47    18     6     3

sum(cov_tss_g500>1)

## [1] 5233

cov_tss_g500_bs=countOverlaps(Dm_tss,Dmg+500,ignore.strand=T) #both strands
sum(cov_tss_g500_bs>1)

## [1] 10768
```

Getting the corresponding overlaps with `findOverlaps`:

```
fov_tss_g500_bs=findOverlaps(Dm_tss,Dmg+500,ignore.strand=T)
Dmg[c(1,1383)]

## GRanges object with 2 ranges and 1 metadata column:
##               seqnames             ranges strand |     gene_id
##                  <Rle>          <IRanges>  <Rle> | <character>
##   FBgn0000003    chr3R [2648220, 2648518]      + | FBgn0000003
##   FBgn0011904    chr3R [2648685, 2648757]      - | FBgn0011904
##   -------
##   seqinfo: 15 sequences (1 circular) from dm3 genome
```

Now, imagine we have a set of 10K NGS reads:

```
set.seed(0)
randomreads2L=GRanges(seqnames="chr2L",
                    ranges=IRanges(start=floor(runif(10000,5000,50000)),
                                   width=100),
                    strand="*")
```

And we want to select only the reads overlapping with those two genes:

```
sort(Dmg)[1:2]

## GRanges object with 2 ranges and 1 metadata column:
##               seqnames         ranges strand |     gene_id
##                  <Rle>      <IRanges>  <Rle> | <character>
##   FBgn0031208    chr2L [ 7529,  9484]      + | FBgn0031208
##   FBgn0263584    chr2L [21952, 24237]      + | FBgn0263584
##   -------
##   seqinfo: 15 sequences (1 circular) from dm3 genome
```

We could use:

```
subsetByOverlaps(randomreads2L,sort(Dmg)[1:2])

## GRanges object with 1017 ranges and 0 metadata columns:
##          seqnames         ranges strand
##              <Rle>      <IRanges>  <Rle>
##      [1]    chr2L [ 7780,  7879]      *
##      [2]    chr2L [22284, 22383]      *
##      [3]    chr2L [22101, 22200]      *
##      [4]    chr2L [22375, 22474]      *
##      [5]    chr2L [22207, 22306]      *
##      ...      ...            ...    ...
##   [1013]    chr2L [22674, 22773]      *
##   [1014]    chr2L [23550, 23649]      *
##   [1015]    chr2L [22940, 23039]      *
##   [1016]    chr2L [22316, 22415]      *
##   [1017]    chr2L [ 8623,  8722]      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

or:

```
randomreads2L[overlapsAny(randomreads2L,sort(Dmg)[1:2])]
```

```
## GRanges object with 1017 ranges and 0 metadata columns:
##           seqnames         ranges strand
##              <Rle>      <IRanges>  <Rle>
##      [1]    chr2L [ 7780,  7879]      *
##      [2]    chr2L [22284, 22383]      *
##      [3]    chr2L [22101, 22200]      *
##      [4]    chr2L [22375, 22474]      *
##      [5]    chr2L [22207, 22306]      *
##      ...      ...            ...    ...
##   [1013]    chr2L [22674, 22773]      *
##   [1014]    chr2L [23550, 23649]      *
##   [1015]    chr2L [22940, 23039]      *
##   [1016]    chr2L [22316, 22415]      *
##   [1017]    chr2L [ 8623,  8722]      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

To count the number of reads overlapping with those 2 genes we could use:

```
assays(summarizeOverlaps(sort(Dmg)[1:2],randomreads2L,mode="Union"))$counts
```

```
##             reads
## FBgn0031208   465
## FBgn0263584   552
```

Further examples of counting reads are provided in paragraph 5.6.

3.6 RangedData

The *RangedData* class is defined in the *IRanges* package. It contains genomic coordinates along with some values (typically a numerical variable) defined on these ranges. In practice, metadata (*mcols*) of a *GRanges* object can contain such data and the user would rather work with *GRanges*. However, we mention here the *RangedData* class because it is a more general data structure used in some packages (see for example the *rtracklayer* vignette).

4 Working with FASTQ files

In this paragraph, we present some functions from the *ShortRead* package which is primarily dedicated to manipulating and analyzing raw (not mapped) NGS reads contained in FASTQ files [15].

4.1 FASTQ format

During the sequencing process, raw data are generated (some sort of physical measurements) and processed (image analysis, signal processing, etc.) resulting in sequences and quality informations which are typically presented in FASTQ files. The FASTQ format contains both sequence and quality informations ("FASTQ = FASta + Quality"). Both sequences and Phred quality scores are encoded using ASCII printable characters. In a FASTQ file, the information for each read occupies 4 lines:

@read01087	`read name`
GCCTTCTTGTACGTTTTTGCTGTGAGT	`read sequence`
+	`separation line`
GGGGBB@DB?+>B?B>EE?:E######	`read quality`

4.2 Reading FASTQ files

The simplest function to import a FASTQ files in *R* is:

```r
fq1_path=system.file(package="ShortRead","extdata","E-MTAB-1147",
                     "ERR127302_1_subset.fastq.gz")
# A FASTQ file containing 20K reads
myFastq=readFastq(fq1_path)
```

The resulting object can be explored using:

```r
myFastq

## class: ShortReadQ
## length: 20000 reads; width: 72 cycles

myFastq[1:5]
```

```
## class: ShortReadQ
## length: 5 reads; width: 72 cycles
```

```
head(sread(myFastq),3)
```

```
##   A DNAStringSet instance of length 3
##     width seq
## [1]    72 GTCTGCTGTATCTGTGTCGGCTGTCTCGCGGG...TCAATGAAGGCCTGGAATGTCACTACCCCCAG
## [2]    72 CTAGGGCAATCTTTGCAGCAATGAATGCCAAT...AGTGGCTTTTGAGGCCAGAGCAGACCTTCGGG
## [3]    72 TGGGCTGTTCCTTCTCACTGTGGCCTGACTAA...GGCATTAAGAAAGAGTCACGTTTCCCAAGTCT
```

```
head(quality(myFastq),3)
```

```
## class: FastqQuality
## quality:
##   A BStringSet instance of length 3
##     width seq
## [1]    72 HHHHHHHHHHHHHHHHHHHHHHEBDBB?B:BBGG...BFEFBDBD@DDECEE3>:?;@@@>?=BAB?##
## [2]    72 IIIIHIIIGIIIIIIIHIIIIEGBGHIIIIHG...IIHIIIHIIIIIGIIIEGIIGBGE@DDGGGIG
## [3]    72 GGHBHGBGGGHHHHDHHHHHHHHHHFGHHHHHH...HHHHHHHGHFHHHHHHHHHHHHHH8AGDGGG>
```

```
head(id(myFastq),3)
```

```
##   A BStringSet instance of length 3
##     width seq
## [1]    53 ERR127302.8493430 HWI-EAS350_0441:1:34:16191:2123#0/1
## [2]    53 ERR127302.21406531 HWI-EAS350_0441:1:88:9330:2587#0/1
## [3]    55 ERR127302.22173106 HWI-EAS350_0441:1:91:10434:14757#0/1
```

```
encoding(quality(myFastq))[2:50]
```

```
##  "  #  $  %  &  '  (  )  *  +  ,  -  .  /  0  1  2  3  4  5  6  7  8  9  :
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
##  ;  <  =  >  ?  @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R
## 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
```

```
alphabet(sread(myFastq))
```

```
##  [1] "A" "C" "G" "T" "M" "R" "W" "S" "Y" "K" "V" "H" "D" "B" "N" "-" "+" "."
```

However, FASTQ files are often too large to be imported at once in R. The first solution, often used to perform QA/QC, is to obtain a random sample of the reads:

```
set.seed(123)
fqs1K = FastqSampler(fq1_path,1000)
reads_sample=yield(fqs1K)
close(fqs1K) #close connection
reads_sample
```

```
## class: ShortReadQ
## length: 1000 reads; width: 72 cycles
```

The default sample size is 1 million reads which fits easily into memory.

If we need to work on all the reads, we would process the FASTQ file in chunks. In this trivial example we count the number of reads in the FASTQ file:

```
nr_myFastq=0
strm <- FastqStreamer(fq1_path,1000)
repeat {
        fq <- yield(strm)
        if (length(fq) == 0)
          break
  ## Get FASTQ chunk
        nr_myFastq=nr_myFastq+length(fq)
  ## Do something on the chunk
}
close(strm) #close the connection
nr_myFastq

## [1] 20000
```

4.3 Quality assessment on FASTQ files

One major reason to analyze FASTQ files is to evaluate the quality and potential biases associated with the sequencing process. Outside of R, the Fastqc tool is widely used to perform such an analysis. It uses a random sample drawn from the FASTQ file to obtain some statistics on the reads. One of its limitation is that it generates one report for each sample but MultiQC allows to easily integrate them (as well as results from many other tools).
The *ShortRead* package also provides efficient functions to generate an HTML QA report for multiple samples:

```
fqPath = system.file(package="ShortRead", "extdata", "E-MTAB-1147")
fqFiles = dir(fqPath, pattern="fastq.gz", full=TRUE)
coll = QACollate(QAFastqSource(fqFiles), QAReadQuality(),
                 QAAdapterContamination(), QANucleotideUse(),
                 QAQualityUse(), QASequenceUse(),
                 QAFrequentSequence(n=10), QANucleotideByCycle(),
                 QAQualityByCycle())
qa2OnFastq = qa2(coll,BPPARAM=SerialParam(), verbose=FALSE)
## qa_report=report(qa2OnFastq) #generate the report
## browseURL(qa_report) #display in your browser
slotNames(qa2OnFastq)

## [1] "src"             "filtered"        "flagged"         "listData"
## [5] "elementType"     "elementMetadata" "metadata"

names(qa2OnFastq)

## [1] "QAReadQuality"          "QAAdapterContamination"
## [3] "QANucleotideUse"        "QAQualityUse"
## [5] "QASequenceUse"          "QAFrequentSequence"
## [7] "QANucleotideByCycle"    "QAQualityByCycle"

slotNames(qa2OnFastq[["QANucleotideUse"]])

## [1] "addFilter" "useFilter" "values"    "flag"      "html"

qa2OnFastq[["QANucleotideUse"]]
```

```
## class: QANucleotideUse
## html template: /usr/lib/R/library/ShortRead/template.../QANucleotideUse.html
## useFilter: TRUE; addFilter: TRUE
```

As illustrated, individual results from the quality analysis can be extracted from the *.QA* object but this is not well documented yet, nor particularly easy.

An alternative is to use the qa function from which it is relatively easy to extract individual components and to generate separate plots such as Figure 12:

```
qaOnFastq = qa(fqPath,"fastq.gz",BPPARAM=SerialParam()) #collect statistics
qaOnFastq[["readCounts"]]

##                                 read filter aligned
## ERR127302_1_subset.fastq.gz 20000     NA      NA
## ERR127302_2_subset.fastq.gz 20000     NA      NA

readQuals <- qaOnFastq[["readQualityScore"]]
ggplot(readQuals,aes(x=quality, y=density, colour=lane))+
    geom_line(size=1)+
    scale_colour_discrete(name="Lane",
                          breaks=c("ERR127302_1_subset.fastq.gz",
                                   "ERR127302_2_subset.fastq.gz"),
                          labels=c("ERR127302_1","ERR127302_2")) +
    theme_bw()
```

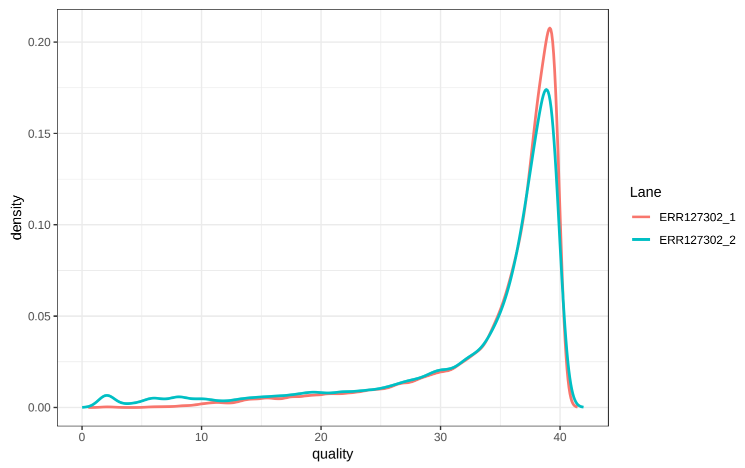


Figure 12: Distribution of average base quality

There are also other *Bioconductor* packages more or less dedicated to generate and plot QA/QC statistics from FASTQ files. These include the *seqTools*, *qrqc* and *Rqc* packages. The latter two packages make use of *ggplot2* graphics. A couple examples using *Rqc* are illustrated in Figure 13

```
rqcResultSet = rqcQA(fqFiles,sample=T)
rqcCycleQualityPlot(rqcResultSet[1])
rqcCycleBaseCallsLinePlot(rqcResultSet[2])
```

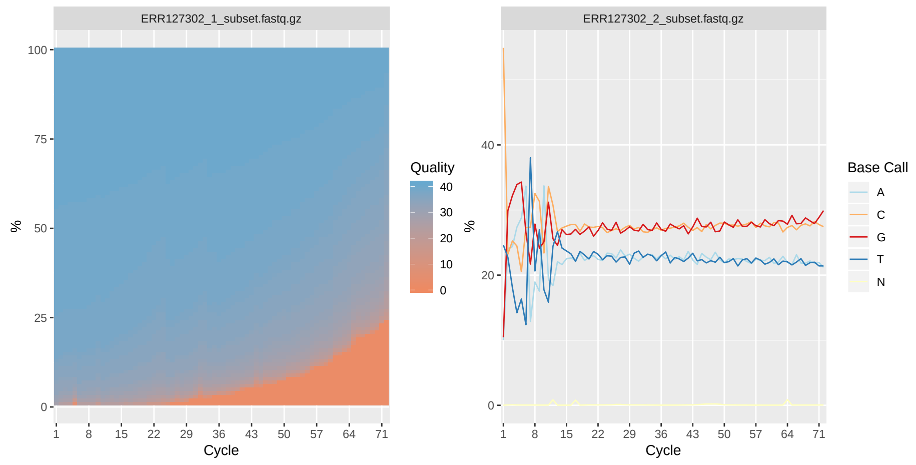


Figure 13: Examples of QC plots using the Rqc package

4.4 Reads filtering and trimming

Based on QA/QC it is sometimes advisable to perform read filtering and to trim the extremities of the reads. The *ShortRead* allows to perform such filtering and trimming steps with great flexibility. Note that the choices made below are only to illustrate the different functions, not to recommend some specific preprocessing in the general case.
First, we create some filters:

```
max1N=nFilter(threshold=1L)
#Remove reads with more than 1N
goodq = srFilter(function(x){
                            apply(as(quality(x),"matrix"),
                            1,median,na.rm=T)>=30
                            },
                 name="MedianQualityAbove30")
#Custom filter: Remove reads with median quality<30
myFilter=compose(max1N,goodq) #combine filters
```

Then, we create a function to apply filters and trimming on chunks of a FASTQ file:

```
FilterAndTrim = function(fl,destination=sprintf("%s_filtered",fl))
{
  stream = FastqStreamer(fl)
  on.exit(close(stream))
  ## open input stream

  repeat {
    fq=yield(stream)
    if (length(fq)==0)
      break
    ## get fastq chunk

    ###TRIM
    fq = narrow(fq,start=5,end=70)
    ## trim the first 4 and the last 2 bases

    ####FILTER
    fq = fq[myFilter(fq)]
    ## remove reads that:
      ## contain more than 1 N
      ## have median quality < 30

    writeFastq(fq, destination, mode="a")
    ## Append to fastq file
  }
}
```

Finally we apply this function on a FASTQ file:

```
FilterAndTrim(fqFiles[1],
              destination=file.path(getwd(),"FilteredFastq.fastq"))
FilteredFastq=readFastq("FilteredFastq.fastq")
FilteredFastq
```

5 Working with SAM/BAM files

5.1 Tools for SAM/BAM files

When a reference genome sequence is available, the reads are generally mapped to this reference using one of the NGS read aligner available (see paragraph 2.4). Note that R provides an interface to bowtie and SpliceMap [16] aligners via the *Rbowtie* package which is used in the pipeline of the *QuasR* package. The *Rsubread* package also provides a relatively complete pipeline for NGS data analysis. It includes a specific aligner (*Subread*) based on an original "seed-and-vote" mapping algorithm [12] as well as tools to count (*featureCounts*) the reads within genomic features [17] and to map exon junctions from RNA-seq data.

Once aligned, the reads and their genomic coordinates (defined by chromosome, position and strand) are typically returned in a SAM file [18]. A BAM file contains the same information as the corresponding SAM file but BAM is a binary format and thus BAM file size is reduced and its content is not human-readable. These files are often manipulated with Samtools, Picard Tools and Python scripts. A nice explanation of what SAM/BAM files contain can be found here.

Here, we will mainly use the following libraries:

- *Rsamtools* which provides an interface to samtools, bcftools and tabix (see htslib)
- *GenomicAlignments* which provides efficient tools to manipulate short genomic alignments

Examples of BAM files from single- and paired-read sequencing are provided in the *pasillaBamSubset*.

5.2 Importing BAM files

5.2.1 Single-end reads

The path for the BAM file is obtained with:

```
sr_bamFile=untreated1_chr4() # from passilaBamSubset package
```

If the BAM file was not indexed yet (i.e. a .bai file present in the same directory and nammed as the .bam file), we could build such an index using:

```
indexBam(sr_bamFile)
```

Note that a number of other functions are available to manipulate SAM/BAM files, such as `asSam`/`asBam`, `sortBam` or `mergeBam`. See `?scanBam` for details.

Now, we could define which regions of the genome we would like to import (*'which'* ; note that here only chr4 is available in the BAM file but for the example we try to extract data from chr2L also), which columns of the BAM file we would like to import (*'what'*) and possibly filters for unwanted reads (*'flag'*). These informations are stored in a *ScanBamParam* object:

```
which = RangesList("chr2L"=IRanges(7000,10000),
                   "chr4"=IRanges(c(75000,1190000),c(85000,1203000)))

scanBamWhat() #available fields

##  [1] "qname"       "flag"        "rname"       "strand"      "pos"
##  [6] "qwidth"      "mapq"        "cigar"       "mrnm"        "mpos"
## [11] "isize"       "seq"         "qual"        "groupid"     "mate_status"

what = c("rname","strand","pos","qwidth","seq")
flag=scanBamFlag(isDuplicate=FALSE)
param=ScanBamParam(which=which,what=what,flag=flag)
```

See `?ScanBamParam` for other options and examples. As briefly illustrated it is easy to define filters (based on SAM flags using `scanBamFlag` or on tags present in the SAM/BAM files) when importing your BAM files.

Now, we use `scanBam` from *Rsamtools* to import the data in R:

```
mysrbam=scanBam(sr_bamFile,param=param)
class(mysrbam)

## [1] "list"

names(mysrbam)
```

```
## [1] "chr2L:7000-10000"     "chr4:75000-85000"     "chr4:1190000-1203000"
sapply(mysrbam,sapply,length)["rname",] #number of imported reads
##     chr2L:7000-10000     chr4:75000-85000 chr4:1190000-1203000
##                    0                  304                  736
```

The resulting object is a list which is not always easy to manipulate for downstream applications.

So we would rather use the `readGAlignments` from the *GenomicAlignments* package:

```
mysrbam2=readGAlignments(sr_bamFile,
                         param=ScanBamParam(which=which,
                                            what="seq",
                                            flag=flag))
mysrbam2[1:2]
## GAlignments object with 2 alignments and 1 metadata column:
##       seqnames strand             cigar    qwidth     start       end
##          <Rle>  <Rle>       <character> <integer> <integer> <integer>
##   [1]     chr4      + 21M13615N50M55N4M        75     72990     86734
##   [2]     chr4      - 4M13615N50M55N21M        75     73007     86751
##           width     njunc |                   seq
##       <integer> <integer> |        <DNAStringSet>
##   [1]     13745         2 | AAAAACTGCA...CGTAGCCACA
##   [2]     13745         2 | ATACCTGTGA...TGGACGGCTG
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

Note that we have redefined the ScanBamParam. This is because `readGAlignments` comes with predefined fields to import and we just need to add those extra fields we want to import in the *ScanBamParam* parameter object. Here we imported the sequences to show that they are imported as a *DNAStringSet* but it is generally not necessary to keep these sequences once the reads have been mapped.

So we don't keep them for the next steps:

```
mysrbam2=readGAlignments(sr_bamFile,
                         param=ScanBamParam(which=which))
mysrbam2[1:2]
## GAlignments object with 2 alignments and 0 metadata columns:
##       seqnames strand             cigar    qwidth     start       end
##          <Rle>  <Rle>       <character> <integer> <integer> <integer>
##   [1]     chr4      + 21M13615N50M55N4M        75     72990     86734
##   [2]     chr4      - 4M13615N50M55N21M        75     73007     86751
##           width     njunc
##       <integer> <integer>
##   [1]     13745         2
##   [2]     13745         2
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

The object returned is a *GAlignments* (see `?'GAlignments-class'` for details and accessors).

These objects are highly similar to *GRanges*. They can be accessed with similar functions:

```
head(start(mysrbam2))

## [1] 72990 73007 73007 73007 73007 73007

head(width(mysrbam2))

## [1] 13745 13745 13745 13745 13745 13745

seqnames(mysrbam2)

## factor-Rle of length 1040 with 1 run
##   Lengths: 1040
##   Values : chr4
## Levels(8): chr2L chr2R chr3L chr3R chr4 chrM chrX chrYHet

cigar(mysrbam2)[1:3]

## [1] "21M13615N50M55N4M" "4M13615N50M55N21M" "4M13615N50M55N21M"

head(njunc(mysrbam2))

## [1] 2 2 2 2 2 2
```

They can be converted to *GRanges*:

```
granges(mysrbam2)[1:2]

## GRanges object with 2 ranges and 0 metadata columns:
##       seqnames          ranges strand
##          <Rle>       <IRanges>  <Rle>
##   [1]     chr4 [72990, 86734]      +
##   [2]     chr4 [73007, 86751]      -
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

And one can easily access the details of each read alignment as *GRanges* organized in a *GRangesList* using:

```
grglist(mysrbam2)[[1]] #only first read shown element here

## GRanges object with 3 ranges and 0 metadata columns:
##       seqnames          ranges strand
##          <Rle>       <IRanges>  <Rle>
##   [1]     chr4 [72990, 73010]      +
##   [2]     chr4 [86626, 86675]      +
##   [3]     chr4 [86731, 86734]      +
##   -------
##   seqinfo: 8 sequences from an unspecified genome

junctions(mysrbam2)[[1]] #and the corresponding junctions
```

```
## GRanges object with 2 ranges and 0 metadata columns:
##       seqnames         ranges strand
##          <Rle>      <IRanges>  <Rle>
##   [1]     chr4 [73011, 86625]      +
##   [2]     chr4 [86676, 86730]      +
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

See below and in `help('junctions-methods',"GenomicAlignments")`, `help('findOverlaps-methods',"GenomicAlignments")` and `help('coverage-methods',"GenomicAlignments")` for other methods defined on *GAlignments* objects.

5.2.2 Paired-end reads

An example of paired-end data is available in the *pasillaBamSubset* package:

```
pr_bamFile=untreated3_chr4() # from passilaBamSubset package
```

We can extract these data using:

```
myprbam=readGAlignmentPairs(pr_bamFile,
                              param=ScanBamParam(which=which))
myprbam[1:2]

## GAlignmentPairs object with 2 pairs, strandMode=1, and 0 metadata columns:
##       seqnames strand :         ranges --           ranges
##          <Rle>  <Rle> :      <IRanges> --        <IRanges>
##   [1]     chr4      - : [13711, 13747] -- [ 74403,  77053]
##   [2]     chr4      * : [74403, 77053] -- [955236, 964043]
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

The *GAlignmentPairs* class holds only read pairs (reads with no mate or with ambiguous pairing are discarded). Note that the `readGalignmentPairs` function has a strandMode argument to specify how to report the strand of a pair. For stranded protocols, depending how the libraries were generated, strandMode should be set to 1 (the default for e.g. directional Illumina protocol by ligation) or 2 (e.g. for dUTP or Illumina stranded TruSeq PE protocol). The individual reads can be accessed as *GAlignments* objects using:

```
myprbam[1] #first record

## GAlignmentPairs object with 1 pair, strandMode=1, and 0 metadata columns:
##       seqnames strand :         ranges --         ranges
##          <Rle>  <Rle> :      <IRanges> --      <IRanges>
##   [1]     chr4      - : [13711, 13747] -- [74403, 77053]
##   -------
##   seqinfo: 8 sequences from an unspecified genome

first(myprbam[1]) #first sequenced fragment

## GAlignments object with 1 alignment and 0 metadata columns:
##       seqnames strand       cigar    qwidth     start       end     width
##          <Rle>  <Rle> <character> <integer> <integer> <integer> <integer>
##   [1]     chr4      -         37M        37     13711     13747        37
##          njunc
##      <integer>
##   [1]        0
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

```
last(myprbam[1]) #last sequenced fragment
```

```
## GAlignments object with 1 alignment and 0 metadata columns:
##       seqnames strand       cigar    qwidth     start       end     width
##          <Rle>  <Rle> <character> <integer> <integer> <integer> <integer>
##   [1]     chr4      +  33M2614N4M        37     74403     77053      2651
##          njunc
##      <integer>
##   [1]        1
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

We could also use the `readGAlignmentsList` function which returns both mate-pairs and non-mates in a more classical "list-like" structure:

```
myprbam_list=readGAlignmentsList(pr_bamFile,
                                 param=ScanBamParam(which=which))
myprbam_list[1:2]
```

```
## GAlignmentsList object of length 2:
## [[1]]
## GAlignments object with 2 alignments and 0 metadata columns:
##       seqnames strand cigar qwidth start   end width njunc
##   [1]     chr4      +   37M     37 80628 80664    37     0
##   [2]     chr4      -   37M     37 80775 80811    37     0
##
## [[2]]
## GAlignments object with 2 alignments and 0 metadata columns:
##       seqnames strand cigar qwidth start   end width njunc
##   [1]     chr4      +   37M     37 81002 81038    37     0
##   [2]     chr4      -   37M     37 83706 83742    37     0
##
## -------
## seqinfo: 8 sequences from an unspecified genome
```

```
table(elementNROWS(myprbam_list))
```

```
##
##   1   2
##  74 287
```

```
summary(mcols(myprbam_list)$mate_status) #mate status as metadata
```

```
##     mated ambiguous   unmated
##       287         0        74
```

Sometimes, it is desirable to process the BAM file in chunk. The `BamFile` allows to create a reference to a BAM file which can be opened, used in a loop and then closed (example taken from the excellent HOWTO vignette from the *GenomicRanges* package):

```
bf = BamFile(sr_bamFile,yieldSize=100000) #create reference
open(bf) #open connection
cvg = NULL #initialize

repeat {
  chunk <- readGAlignments(bf) #loop on the BAM file
  if (length(chunk) == 0L)
    break
  chunk_cvg <- coverage(chunk)
  if (is.null(cvg)) {
    cvg <- chunk_cvg
  } else {
    cvg <- cvg + chunk_cvg
  }
}
close(bf)

cvg$chr4

## integer-Rle of length 1351857 with 122061 runs
##   Lengths:  891   27    5   12   13   45 ...    3  106   75 1600   75 1659
##   Values :    0    1    2    3    4    5 ...    6    0    1    0    1    0
```

Note that such a loop is now directly performed by the `coverage` method for *BamFile* class. For details, see `?"coverage,GAlignments-method"`.

5.3 Some QA/QC on aligned reads

The `qa` function from the *ShortRead* also performs quality assessements on BAM files. Alternatively, a quick summary can be generated using:

```
quickBamFlagSummary(pr_bamFile)

##                                  group |    nb of |    nb of | mean / max
##                                     of |  records |   unique | records per
##                                records | in group |   QNAMEs | unique QNAME
## All records........................ A |   175346 |    93620 | 1.87 / 10
##   o template has single segment.... S |        0 |        0 |   NA / NA
##   o template has multiple segments. M |   175346 |    93620 | 1.87 / 10
##       - first segment.............. F |    88069 |    83413 | 1.06 / 8
##       - last segment............... L |    87277 |    82600 | 1.06 / 9
##       - other segment.............. O |        0 |        0 |   NA / NA
##
## Note that (S, M) is a partitioning of A, and (F, L, O) is a partitioning of M.
## Indentation reflects this.
##
```

```
## Details for group M:
##   o record is mapped.............. M1 |   175346 |    93620 | 1.87 / 10
##       - primary alignment......... M2 |   175346 |    93620 | 1.87 / 10
##       - secondary alignment....... M3 |        0 |        0 |   NA / NA
##   o record is unmapped............ M4 |        0 |        0 |   NA / NA
##
## Details for group F:
##   o record is mapped.............. F1 |    88069 |    83413 | 1.06 / 8
##       - primary alignment......... F2 |    88069 |    83413 | 1.06 / 8
##       - secondary alignment....... F3 |        0 |        0 |   NA / NA
##   o record is unmapped............ F4 |        0 |        0 |   NA / NA
##
## Details for group L:
##   o record is mapped.............. L1 |    87277 |    82600 | 1.06 / 9
##       - primary alignment......... L2 |    87277 |    82600 | 1.06 / 9
##       - secondary alignment....... L3 |        0 |        0 |   NA / NA
##   o record is unmapped............ L4 |        0 |        0 |   NA / NA
```

This could allow to select relevant filters when importing the data in *R*.

5.4 Computing a coverage

Computing a coverage (number of reads aligning at a given position on the genome) on a *GAlignments* is straightforward:

```
cvg_sr=coverage(mysrbam2)
cvg_sr$chr4

## integer-Rle of length 1351857 with 1266 runs
##   Lengths:  72989     17      4   2092 ...      5   6175     49 143448
##   Values :      0      1      9      0 ...      1      0      1      0
```

The result is an *RleList* organized by chromosomes.

We can extract specific *Views* from this object:

```
#convert imported intervals to GRanges:
which_chr4_gr=GRanges(seqnames="chr4",
                      ranges=which$chr4,strand="*")
#Get the exons
ex_chr4=exonsByOverlaps(TxDb.Dmelanogaster.UCSC.dm3.ensGene,which_chr4_gr)
head(Views(cvg_sr$chr4,ranges(ex_chr4))) #extract Views

## Views on a 1351857-length Rle subject
##
## views:
##     start   end width
## [1] 77123 77175    53 [3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 ...]
## [2] 77866 78037   172 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 ...]
## [3] 80602 81041   440 [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 ...]
## [4] 81196 81600   405 [3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 ...]
## [5] 83599 83750   152 [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 ...]
## [6] 76457 76957   501 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...]
```

Note that the coverage function is not strand specific. If we want the coverage for the minus strand only, we could use:

```
coverage(mysrbam2[strand(mysrbam2)=="-"])$chr4

## integer-Rle of length 1351857 with 724 runs
##   Lengths:  73006      4   3895     75 ...      5   6175     49 143448
##   Values :      0      8      0      1 ...      1      0      1      0
```

It is sometimes necessary to shift or resize the reads before computing coverage. This is a typical situation in ChIP-seq data analysis when the binding of a transcription factor or a chromatin mark with punctuate enrichment is studied. Here, we load some ChIP-seq experimental data from the *MMDiffBamSubset* and import the reads as a *GAlignments* objet:

```
# library(MMDiffBamSubset)
ChIPex_path = WT.AB2() #from MMDiffBamSubset package
ChIP_ga=readGAlignments(ChIPex_path,
                        param=ScanBamParam(
                          which=GRanges(seqnames="chr1",
                                        ranges=IRanges(3e6,5e6),
                                        strand="*")))
```

Then we can compute the coverage after applying some tranformation on the reads (shift or resize):

```
coverage(ChIP_ga)$chr1

## integer-Rle of length 197195432 with 63367 runs
##   Lengths:   3000431        54        70 ...         9        17 192195393
##   Values :         0         1         0 ...         2         1         0

#directly in coverage (!shift is not strand-aware):
coverage(ChIP_ga,shift=150)$chr1

## integer-Rle of length 197195432 with 63367 runs
##   Lengths:   3000581        54        70 ...         9        17 192195243
##   Values :         0         1         0 ...         2         1         0

#now "strand-aware" shift:
coverage(ChIP_ga,shift=150*as.numeric(paste(strand(ChIP_ga),"1",sep="")))$chr1

## integer-Rle of length 197195432 with 63427 runs
##   Lengths:   3000494        54        33 ...        45         9 192195260
##   Values :         0         1         0 ...         2         1         0

#resize via a GRanges (strand-aware):
coverage(resize(granges(ChIP_ga),300))$chr1

## integer-Rle of length 197195432 with 64106 runs
##   Lengths:   3000398        33       124 ...        62         9 192195164
##   Values :         0         1         2 ...         2         1         0
```

5.5 Finding peaks in read coverage

In ChIP-seq experiments, one often look for "peaks" (or broad regions in e.g. ChIP of chromatin marks or PolII) in the read coverage. This task, often called "peak calling" or "peak finding" can be performed outside of R with one of the numerous "peak callers" available such as MACS / MACS2, SPP or SICER (for broad regions). A number of specific and advanced tools also exist in R to perform peak calling such as: *BayesPeak*, *chipseq*, *bumphunter*, *exomePeak*, *CSAR*, *jmosaics* or *PICS*. Here, we only illustrate a naive approach to get a list of peaks by applying a single threshold on the coverage object using the `slice` function:

```
cvg_H3K4me3=coverage(resize(granges(ChIP_ga),300))
slice(cvg_H3K4me3,lower=20)$chr1

## Views on a 197195432-length Rle subject
##
## views:
##         start     end width
##   [1] 3027290 3027346    57 [20 21 21 21 21 21 21 21 21 21 21 21 21 21 ...]
##   [2] 3042723 3042728     6 [20 20 20 20 20 20]
##   [3] 3042793 3042903   111 [20 20 20 20 20 20 20 20 20 20 20 20 20 20 ...]
##   [4] 3042911 3042916     6 [20 20 20 20 20 20]
##   [5] 3048694 3048705    12 [20 20 20 20 20 20 20 20 20 20 20 20]
##   [6] 3048777 3048800    24 [20 20 20 20 20 20 20 20 20 20 20 20 20 20 ...]
##   [7] 3052480 3052488     9 [20 20 20 20 20 20 20 20 20]
##   [8] 3052492 3052880   389 [20 20 20 20 20 20 20 20 20 20 20 20 20 21 ...]
##   [9] 3129266 3129557   292 [20 20 20 21 21 22 22 22 21 21 21 21 22 24 ...]
##   ...     ...     ...   ... ...
## [272] 4964448 4964489    42 [20 20 21 21 21 21 21 21 21 21 21 21 21 21 ...]
## [273] 4976296 4976296     1 [20]
## [274] 4976308 4976317    10 [20 20 20 20 20 21 21 21 21 21]
## [275] 4976482 4976502    21 [20 20 20 20 20 20 20 20 20 20 20 20 20 20 ...]
## [276] 4977358 4977359     2 [20 20]
## [277] 4977366 4977443    78 [20 20 20 20 20 20 20 20 20 20 20 20 20 20 ...]
## [278] 4977485 4977599   115 [20 21 21 21 21 21 21 21 21 21 21 21 22 22 ...]
## [279] 4977615 4977630    16 [20 20 20 20 20 20 20 20 20 20 20 20 20 20 ...]
## [280] 4978067 4978377   311 [20 20 20 21 21 21 21 21 21 21 21 22 22 22 ...]
```

FixMe: more examples soon...maybe...

5.6 Counting reads / read summarization

Another common task is to count how many reads align to a set of genomic features. This counting operation is sometimes called read summarization or data reduction. It is typically done in RNA-seq experiments to produce a count matrix for subsequent analysis. The counts can be obtained on e.g. genes, transcripts or exons depending on the aim of the study. Some packages such as *Rsubread*, *QuasR* and *easyRNAseq* provide specific functions to perform the counting step. Here we only present the general function `summarizeOverlaps` from *GenomicAlignments*.

We want to count the reads aligning on the exons. So we first get the exons, organized by genes, which are located in our region of interest:

```
exbygn_chr4=subsetByOverlaps(exonsBy(TxDb.Dmelanogaster.UCSC.dm3.ensGene,
                                     by="gene"),ex_chr4)
```

Then we use the `summarizeOverlaps` function to obtain the counts (*Note that the counting is strand-aware by default!*):

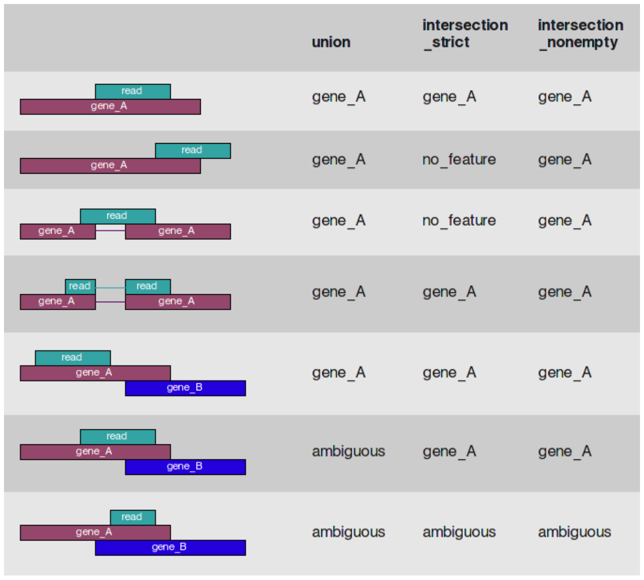
```
count_res=summarizeOverlaps(exbygn_chr4, mysrbam2, mode="Union")
count_res

## class: RangedSummarizedExperiment
## dim: 3 1
## metadata(0):
## assays(1): counts
## rownames(3): FBgn0002521 FBgn0004859 FBgn0264617
## rowData names(0):
## colnames(1): reads
## colData names(2): object records

assays(count_res)$counts

##             reads
## FBgn0002521   346
## FBgn0004859    12
## FBgn0264617   117
```

Different count modes are available, as in HTSeq . These are illustrated in Figure 14.

The result of `summarizeOverlaps` (here `count_res`) is a *SummarizedExperiment* object (See *help(SummarizedExperiment, package="GenomicRanges")* for details and accessors). This class is very similar to the *eSet* class (from the *Biobase* package) which is often used to analyze microarray data. In *eSet* objects, the features are typically microarray probes, probesets or genes corresponding to these probes. In *summarizedExperiment* objects, the features (i.e. rows) are ranges of interest. Figure 15 illustrates the structure of a *SummarizedExperiment* object.

To count reads from multiple BAM files we just need to enter the path to the BAM files as a character vector:

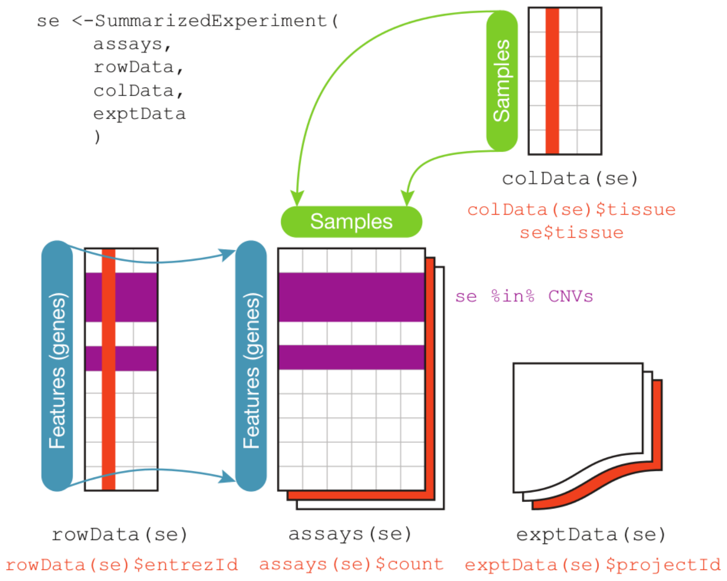
```
count_res2=summarizeOverlaps(exbygn_chr4,
                             c(sr_bamFile,pr_bamFile),
                             mode="Union")


assays(count_res2)$counts
```



	union	intersection _strict	intersection _nonempty
	gene_A	gene_A	gene_A
	gene_A	no_feature	gene_A
	gene_A	no_feature	gene_A
	gene_A	gene_A	gene_A
	gene_A	gene_A	gene_A
	ambiguous	gene_A	gene_A
	ambiguous	ambiguous	ambiguous

read
gene_A
gene_B

Figure 14: Count modes
Taken from the HTSeq website



```
se <-SummarizedExperiment(
     assays,
     rowData,
     colData,
     exptData
     )
```

Figure 15: Structure of *summarizedExperiment* objects
Taken from [3]. The `assays` component contains the data as rectangular matrices. The `rowData` and `colData` components contain metadata on the features and on the samples respectively. The `exptData` component contains experiment-level data. Code examples illustrate how to create a `summarizedExperiment`, subset it and access the components.

```
##             untreated1_chr4.bam untreated3_chr4.bam
## FBgn0002521                 346                 210
## FBgn0004859                 200                  96
## FBgn0264617                 117                  53
```

There are several options available to perform read counting with `summarizeOverlaps`. Some examples are provided below:

```
assays(summarizeOverlaps(exbygn_chr4,myprbam,mode="Union"))$counts

##             reads
## FBgn0002521    97
## FBgn0004859     6
## FBgn0264617    32

assays(summarizeOverlaps(exbygn_chr4,myprbam,mode="Union", #ignore strand
                         ignore.strand=T))$counts

##             reads
## FBgn0002521   192
## FBgn0004859     7
## FBgn0264617    50

assays(summarizeOverlaps(exbygn_chr4,mysrbam2,mode="Union"))$counts

##             reads
## FBgn0002521   346
## FBgn0004859    12
## FBgn0264617   117

#resize the reads to 100:
assays(
    summarizeOverlaps(exbygn_chr4,mysrbam2,mode="Union",
                      preprocess.reads=function(x){
                          resize(granges(x),100)
                          })
    )$counts

##             reads
## FBgn0002521   346
## FBgn0004859     4
## FBgn0264617   117
```

See `?summarizeOverlaps` for details and examples.

The count matrix could then be used for normalization and differential analysis with the *DESeq2*, *edgeR*, *limma*, *DEXseq* packages for example.

6 More annotation packages in *Bioconductor*

6.1 Types of annotation packages

There are a number of annotation packages in *Bioconductor* which can be browsed here. Annotations are provided as *AnnotationDb* objects and the *AnnotationDbi* provides a common interface to these objects. The vignette of the *AnnotationDbi* provides a good introduction to the different types of package available. Figure 16, adapted from this vignette, illustrates the different types of *gene*- and *genome*-centered annotation packages available in *Bioconductor*.

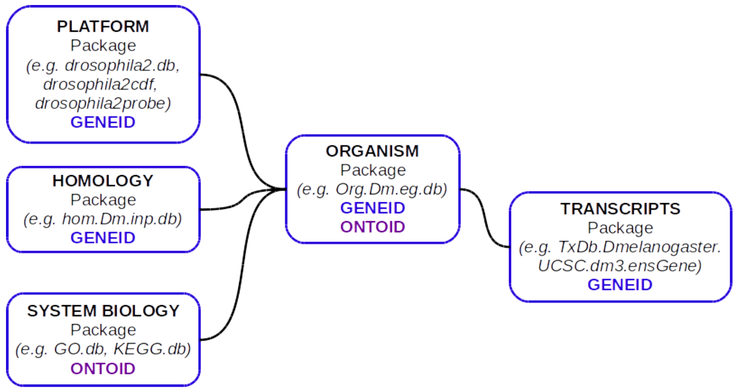


Figure 16: AnnotationDb packages
Adapted from the vignette of *AnnotationDbi*

There are gene-centric packages:

- Organism packages (*OrgDb* class ; e.g. *org.Dm.eg.db*)
- Platform-level (essentially microarrays) packages (*ChipDb* class: e.g. *drosophila2.db* and the corresponding *probe* and *cdf* packages: *drosophila2probe* and *drosophila2cdf*)
- Homology packages (*InparanoidDb* class ; e.g. *hom.Dm.inp.db*)
- System-biology packages (e.g. *GO.db*, *KEGG.db*, *reactome.db*)

And genome-centric packages:

- Transcriptome-oriented packages (*TxDb* class ; e.g. *TxDb.Dmelanogaster.UCSC.dm3.ensGene*)
- Generic Genome feature packages (*FeatureDb* class, e.g. *FDb.UCSC.tRNAs*) that can be created with the *GenomicFeatures* package

In addition, *OrganismDb* packages were recently added based on the *OrganismDbi* package. These packages basically combine all the above packages in a single package. These are available for *Homo.sapiens*, *Mus.musculus* and *Rattus.norvegicus*. Finally packages are available in *Bioconductor* to enable the user to build his own annotation packages. These include *AnnotationForge* and *GenomicFeatures*.

6.2 Accessing annotations

There are 4 basic functions to access nearly all annotations:

- The `column` function shows the fields from which annotatios can be extracted
- The `keytypes` function shows the field from which annotations can be extracted AND which can be used as keys to access these annotations

- The `keys` function retrieves the keys themselves (i.e. values from a 'keytype' field used to access the annotation of the corresponding feature)
- The `select` function extracts the data from the *AnnotationDb* object from the *columns* specified and providing a set of *keys* of a given *keytype*.

In our examples, we will explore the main packages for Drosophila melanogaster:

- *org.Dm.eg.db*
- *drosophila2.db* and the corresponding *drosophila2probe* and *drosophila2cdf* packages
- *hom.Dm.inp.db*
- *GO.db* (not specific to D. melanogaster)
- *TxDb.Dmelanogaster.UCSC.dm3.ensGene*

First, we explore the *OrgDb* package which contains gene-level annotations:

```
## library(org.Dm.eg.db)
columns(org.Dm.eg.db)

##  [1] "ACCNUM"       "ALIAS"        "ENSEMBL"      "ENSEMBLPROT"
##  [5] "ENSEMBLTRANS" "ENTREZID"     "ENZYME"       "EVIDENCE"
##  [9] "EVIDENCEALL"  "FLYBASE"      "FLYBASECG"    "FLYBASEPROT"
## [13] "GENENAME"     "GO"           "GOALL"        "MAP"
## [17] "ONTOLOGY"     "ONTOLOGYALL"  "PATH"         "PMID"
## [21] "REFSEQ"       "SYMBOL"       "UNIGENE"      "UNIPROT"

## help("PATH")
## keytypes(org.Dm.eg.db) #same as columns(org.Dm.eg.db) in this case
uniKeys = keys(org.Dm.eg.db,keytype="UNIPROT")[c(5,6,24)]

cols = c("SYMBOL","GO")
select(org.Dm.eg.db,
       keys=uniKeys[1:2],
       columns=cols,
       keytype="UNIPROT")

## 'select()' returned 1:many mapping between keys and columns

##    UNIPROT  SYMBOL         GO EVIDENCE ONTOLOGY
## 1   Q95RU8     G9a GO:0000791      IDA       CC
## 2   Q95RU8     G9a GO:0002165      IMP       BP
## 3   Q95RU8     G9a GO:0005634      IDA       CC
## 4   Q95RU8     G9a GO:0005705      IDA       CC
## 5   Q95RU8     G9a GO:0007614      IMP       BP
## 6   Q95RU8     G9a GO:0007616      IMP       BP
## 7   Q95RU8     G9a GO:0008270      IEA       MF
## 8   Q95RU8     G9a GO:0008345      IMP       BP
## 9   Q95RU8     G9a GO:0010468      IMP       BP
## 10  Q95RU8     G9a GO:0018024      IDA       MF
## 11  Q95RU8     G9a GO:0035076      IGI       BP
## 12  Q95RU8     G9a GO:0035220      IMP       BP
## 13  Q95RU8     G9a GO:0046959      IMP       BP
## 14  Q95RU8     G9a GO:0050688      IMP       BP
## 15  Q95RU8     G9a GO:0050775      IMP       BP
## 16  Q95RU8     G9a GO:0051567      IMP       BP
## 17  Q95RU8     G9a GO:1900111      IMP       BP
## 18  Q9W5H1 CG13377 GO:0002121      IMP       BP
## 19  Q9W5H1 CG13377 GO:0016319      IMP       BP
## 20  Q9W5H1 CG13377 GO:0042048      IMP       BP
```

The *GO.db* package can be used to retrieve information on the identified Gene Ontology categories (chosen also based on Evidence codes):

```
## library(GO.db)
mygos=c("GO:0000791","GO:0002165", "GO:0008270")
select(GO.db,
       columns=columns(GO.db)[2:4],
       keys=mygos,
       keytype="GOID")

## 'select()' returned 1:1 mapping between keys and columns

##         GOID ONTOLOGY                                TERM
## 1 GO:0000791       CC                          euchromatin
## 2 GO:0002165       BP instar larval or pupal development
## 3 GO:0008270       MF                     zinc ion binding
```

We can also extract a whole table as a *data.frame*:

```
## ls("package:GO.db")
toTable(GOTERM)[1:3,2:4]

##        go_id                             Term Ontology
## 1 GO:0000001        mitochondrion inheritance       BP
## 2 GO:0000002 mitochondrial genome maintenance       BP
## 3 GO:0000003                     reproduction       BP
```

We can also search for a specific pattern in the keys:

```
keys(org.Dm.eg.db,
     keytype="SYMBOL",
     pattern="EcR")

## [1] "EcR"    "DopEcR"

select(org.Dm.eg.db,
       keys=c("EcR","DopEcR"),
       columns=c("ENTREZID","FLYBASE","GENENAME"),
       keytype="SYMBOL")

## 'select()' returned 1:1 mapping between keys and columns

##   SYMBOL ENTREZID     FLYBASE                     GENENAME
## 1    EcR    35540 FBgn0000546            Ecdysone receptor
## 2 DopEcR    38539 FBgn0035538 Dopamine/Ecdysteroid receptor
```

ChipDb packages contain annotations based on microarray probes:

```
## library(drosophila2.db)
ls("package:drosophila2.db")[1:8]

## [1] "drosophila2"            "drosophila2ACCNUM"
## [3] "drosophila2ALIAS2PROBE" "drosophila2CHR"
## [5] "drosophila2CHRLENGTHS"  "drosophila2CHRLOC"
## [7] "drosophila2CHRLOCEND"   "drosophila2.db"

## drosophila2.db #provides information on the underlying database
columns(drosophila2.db)[1:7]

## [1] "ACCNUM"       "ALIAS"        "ENSEMBL"      "ENSEMBLPROT"
## [5] "ENSEMBLTRANS" "ENTREZID"     "ENZYME"

select(drosophila2.db,columns=c("ENTREZID","SYMBOL","ENSEMBL"),
       keys=c("1639797_at","1627097_at","1628020_at"),keytype="PROBEID")

## 'select()' returned 1:1 mapping between keys and columns

##      PROBEID ENTREZID  SYMBOL     ENSEMBL
## 1 1639797_at  4379890 CG34109 FBgn0083945
## 2 1627097_at   318986   t-cup FBgn0051858
## 3 1628020_at    33366 CG15358 FBgn0031373
```

The associated *probe* and *cdf* packages provide information on microarray probe sequences and on position on the array respectively. They are mainly used by different methods during the analysis of Affymetrix microaray data:

```
## library(drosophila2probe)
## library(drosophila2cdf)
drosophila2probe[1:2,] #a data frame with probe sequences

##                    sequence   x   y Probe.Set.Name
## 1 CCTGAATCCTGGCAATGTCATCATC 599 305     1622893_at
## 2 ATCCTGGCAATGTCATCATCAATGG 267  45     1622893_at
##   Probe.Interrogation.Position Target.Strandedness
## 1                          137           Antisense
## 2                          142           Antisense

ls("package:drosophila2cdf")

## [1] "drosophila2cdf" "drosophila2dim" "i2xy"           "xy2i"
```

See the vignettes of the *AnnotationDbi* packages for details.

Homology *InparanoidDb* packages contain the gene orthologs for a number of species. They are extracted from the Inparanoid database [19] and can be interrogated both ways:

```
## library(hom.Dm.inp.db)
select(hom.Dm.inp.db,
       columns=c("HOMO_SAPIENS","CULEX_PIPIENS"),
       keys=c("FBpp0084497", "FBpp0077213"),
       keytype="DROSOPHILA_MELANOGASTER")

## 'select()' returned 1:1 mapping between keys and columns

##   DROSOPHILA_MELANOGASTER CULEX_PIPIENS    HOMO_SAPIENS
## 1             FBpp0084497    CPIJ009610            <NA>
## 2             FBpp0077213    CPIJ004808 ENSP00000362314

select(hom.Dm.inp.db,
       columns=c("HOMO_SAPIENS","DROSOPHILA_MELANOGASTER"),
       keys=c("CPIJ014347","CPIJ005780"),
       keytype="CULEX_PIPIENS")

## 'select()' returned 1:1 mapping between keys and columns

##   CULEX_PIPIENS DROSOPHILA_MELANOGASTER    HOMO_SAPIENS
## 1    CPIJ014347             FBpp0088446 ENSP00000303147
## 2    CPIJ005780             FBpp0072468 ENSP00000360532
```

Finally, *TxDb* packages can also be interrogated using the same methods:

```
select(TxDb.Dmelanogaster.UCSC.dm3.ensGene,
       columns=c("TXID","TXCHROM","TXSTRAND","TXSTART","TXEND"),
       keys=c("FBgn0039183","FBgn0264342","FBgn0030583"),
       keytype='GENEID')

## 'select()' returned 1:1 mapping between keys and columns

##        GENEID  TXID TXCHROM TXSTRAND  TXSTART    TXEND
## 1 FBgn0039183 19327   chr3R        + 20159145 20162971
## 2 FBgn0264342   559   chr2L        +  4647051  4647920
## 3 FBgn0030583 25509    chrX        + 14730758 14731363
```

7 Import/export of genomic data

7.1 *Bioconductor* packages to import/export genomic data

In addition to annotation packages presented in paragraph 6, a number of *Bioconductor* software packages provide interfaces with online databases containing annotations and experimental data.
Here, we will give some examples with the following packages:

- *rtracklayer* [20]
- *AnnotationHub*
- *biomaRt*
- *GEOquery*
- *SRAdb*

7.2 The *rtracklayer* package

The main functionnality of the *rtracklayer* [20] is to import/export files in a number of different standard formats. These include GFF, BED, WIG, bigWig and bedGraph. Additionally, *rtracklayer* also provides functions to interact with genome browsers and in particular UCSC Genome Browser. However, we do not illustrate these functionnalities here.

First we get the path for some example files (from the *Gviz* package):

```
bamExFile_path=system.file(package="Gviz","extdata","test.bam")
gff3ExFile_path=system.file(package="Gviz","extdata","test.gff3")
gtfExFile_path=system.file(package="Gviz","extdata","test.gff2")
bedExFile_path=system.file(package="Gviz","extdata","test.bed")
wigExFile_path=system.file(package="Gviz","extdata","test.wig")
bedGraphExFile_path=system.file(package="Gviz","extdata","test.bedGraph")
```

The `import` and `export` functions of *rtracklayer* will adapt to the different file formats via the *format* argument. If this argument is missing the format is derived from the file extension. Here, we simply import these files using the `import` function without any other argument than the file path:

```
head(import(bedExFile_path))

## GRanges object with 6 ranges and 4 metadata columns:
##       seqnames                 ranges strand |        name     score
##          <Rle>              <IRanges>  <Rle> | <character> <numeric>
##   [1]     chr7 [127471197, 127472363]      + |        Pos1         0
##   [2]     chr7 [127472364, 127473530]      + |        Pos2         0
##   [3]     chr7 [127473531, 127474697]      + |        Pos3         0
##   [4]     chr7 [127474698, 127475864]      + |        Pos4         0
##   [5]     chr7 [127475865, 127477031]      - |        Neg1         0
##   [6]     chr7 [127477032, 127478198]      - |        Neg2         0
##           itemRgb                  thick
##       <character>              <IRanges>
##   [1]     #FF0000 [127471197, 127472363]
##   [2]     #FF0000 [127472364, 127473530]
##   [3]     #FF0000 [127473531, 127474697]
##   [4]     #FF0000 [127474698, 127475864]
##   [5]     #0000FF [127475865, 127477031]
##   [6]     #0000FF [127477032, 127478198]
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths

head(import(wigExFile_path)) #binned at 300bp

## GRanges object with 6 ranges and 1 metadata column:
##       seqnames               ranges strand |     score
##          <Rle>            <IRanges>  <Rle> | <numeric>
##   [1]    chr19 [49302001, 49302300]      * |        -1
##   [2]    chr19 [49302301, 49302600]      * |     -0.75
##   [3]    chr19 [49302601, 49302900]      * |      -0.2
##   [4]    chr19 [49302901, 49303200]      * |      -0.1
##   [5]    chr19 [49303201, 49303500]      * |         0
##   [6]    chr19 [49303501, 49303800]      * |      0.25
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths

Rle(rep(import(wigExFile_path)$score,each=300)) #convert to Rle

## numeric-Rle of length 2700 with 9 runs
##   Lengths:   300   300   300   300   300   300   300   300   300
##   Values :    -1 -0.75  -0.2  -0.1     0  0.25   0.4  0.55     1
```

Comment: Try to import the other files and see what you get. Also try to export files to different formats using the `export` function.
Note that a genome-wide coverage as an *RleList* object can be directly exported as WIG or bedGraph files using the `export` function from *rtracklayer*.

7.3 The *AnnotationHub* package

The *AnnotationHub* allows to easily retrieve a number of public datasets and annotation tracks as standard *Bioconductor* objects such as *GRanges* and *GRangesList*. Note however that many datasets have been pre-processed so the user relies on others for these steps. The datasets available come from e.g. ENSEMBL, NCBI, UCSC, ENCODE and Inparanoid). Here, we briefly illustrate the main commands used to retrieve results of interest.
After loading the library, the user must create an *AnnotationHub* object:

```
library(AnnotationHub)
```

```
ah=AnnotationHub()

## updating metadata:  retrieving 1 resource
## snapshotDate():  2017-10-27
```

To explore this object we take a look at its metadata:

```
annot_ah=mcols(ah) #Informations on the different records
table(annot_ah$rdataclass) #type of files that can be retrieved

##
##      AAStringSet       BigWigFile           biopax        ChainFile
##                1            10247                9             1113
##       data.frame            EnsDb           FaFile          GRanges
##               24              282             5122            19268
##    Inparanoid8Db             list           MSnSet         mzRident
##              268                2                1                1
##          mzRpwiz            OrgDb              Rle SQLiteConnection
##                1             1018             1586                1
##       TwoBitFile             TxDb          VcfFile
##             4027               45                8

table(annot_ah$dataprovider)[1:6] #providers

##
##                         BroadInstitute                                 ChEA
##                                  18248                                    1
##                                  CRIBI                                dbSNP
##                                      1                                    8
##                                Ensembl ftp://ftp.ncbi.nlm.nih.gov/gene/DATA/
##                                  12003                                 1018
```

orgDb objects are available in the hub:

```
query(ah,"orgDb") #search for available orgDb packages

## AnnotationHub with 1018 records
## # snapshotDate(): 2017-10-27
## # $dataprovider: ftp://ftp.ncbi.nlm.nih.gov/gene/DATA/
## # $species: Escherichia coli, Acanthamoeba castellanii_str._Neff, Acantha...
## # $rdataclass: OrgDb
## # additional mcols(): taxonomyid, genome, description,
## #   coordinate_1_based, maintainer, rdatadateadded, preparerclass,
## #   tags, rdatapath, sourceurl, sourcetype
## # retrieve records with, e.g., 'object[["AH57964"]]'
##
##             title
##   AH57964 | org.Ag.eg.db.sqlite
##   AH57965 | org.At.tair.db.sqlite
##   AH57966 | org.Bt.eg.db.sqlite
##   AH57967 | org.Cf.eg.db.sqlite
##   AH57968 | org.Gg.eg.db.sqlite
##   ...       ...
##   AH59987 | org.Pseudoalteromonas_piscicida.eg.sqlite
##   AH59988 | org.Bacteroides_fragilis_YCH46.eg.sqlite
##   AH59989 | org.Pseudomonas_mendocina_ymp.eg.sqlite
##   AH59990 | org.Salmonella_enterica_subsp._enterica_serovar_Typhimurium_s...
##   AH59991 | org.Acinetobacter_baumannii.eg.sqlite
```

```
query(ah,c("orgDb","Arabidopsis")) #those for Arabidopsis only

## AnnotationHub with 2 records
## # snapshotDate(): 2017-10-27
## # $dataprovider: ftp://ftp.ncbi.nlm.nih.gov/gene/DATA/
## # $species: Arabidopsis lyrata_subsp._lyrata, Arabidopsis thaliana
## # $rdataclass: OrgDb
## # additional mcols(): taxonomyid, genome, description,
## #   coordinate_1_based, maintainer, rdatadateadded, preparerclass,
## #   tags, rdatapath, sourceurl, sourcetype
## # retrieve records with, e.g., 'object[["AH57965"]]'
##
##             title
##   AH57965 | org.At.tair.db.sqlite
##   AH59047 | org.Arabidopsis_lyrata_subsp._lyrata.eg.sqlite

newAt=ah[["AH57965"]] #retrieve the orgDb object

## loading from cache '/home/pmartin//.AnnotationHub/64711'

keytypes(newAt) #explore the object

##  [1] "ARACYC"       "ARACYCENZYME" "ENTREZID"     "ENZYME"
##  [5] "EVIDENCE"     "EVIDENCEALL"  "GENENAME"     "GO"
##  [9] "GOALL"        "ONTOLOGY"     "ONTOLOGYALL"  "PATH"
## [13] "PMID"         "REFSEQ"       "SYMBOL"       "TAIR"

select(newAt,
       keys="AT1G01010",
       keytype="TAIR",
       columns=c("REFSEQ","ENTREZID","GO"))

## 'select()' returned 1:many mapping between keys and columns

##         TAIR    REFSEQ ENTREZID         GO EVIDENCE ONTOLOGY
## 1  AT1G01010 NM_099983   839580 GO:0003700      ISS       MF
## 2  AT1G01010 NM_099983   839580 GO:0005634      ISM       CC
## 3  AT1G01010 NM_099983   839580 GO:0006888      RCA       BP
## 4  AT1G01010 NM_099983   839580 GO:0007275      ISS       BP
## 5  AT1G01010 NM_099983   839580 GO:0043090      RCA       BP
## 6  AT1G01010 NP_171609   839580 GO:0003700      ISS       MF
## 7  AT1G01010 NP_171609   839580 GO:0005634      ISM       CC
## 8  AT1G01010 NP_171609   839580 GO:0006888      RCA       BP
## 9  AT1G01010 NP_171609   839580 GO:0007275      ISS       BP
## 10 AT1G01010 NP_171609   839580 GO:0043090      RCA       BP
```

Note that it is also possible to select resources from the Hub using `$`,`subset` and `display`

Another example with data from Roadmap Epigenomics project:

```
epiFiles=query(ah,"EpigenomeRoadMap")
epiFiles

## AnnotationHub with 18248 records
## # snapshotDate(): 2017-10-27
## # $dataprovider: BroadInstitute
## # $species: Homo sapiens
## # $rdataclass: BigWigFile, GRanges, data.frame
## # additional mcols(): taxonomyid, genome, description,
## #   coordinate_1_based, maintainer, rdatadateadded, preparerclass,
## #   tags, rdatapath, sourceurl, sourcetype
## # retrieve records with, e.g., 'object[["AH28856"]]'
##
##             title
##   AH28856 | E001-H3K4me1.broadPeak.gz
##   AH28857 | E001-H3K4me3.broadPeak.gz
##   AH28858 | E001-H3K9ac.broadPeak.gz
##   AH28859 | E001-H3K9me3.broadPeak.gz
##   AH28860 | E001-H3K27me3.broadPeak.gz
##   ...       ...
##   AH49540 | E058_mCRF_FractionalMethylation.bigwig
##   AH49541 | E059_mCRF_FractionalMethylation.bigwig
##   AH49542 | E061_mCRF_FractionalMethylation.bigwig
##   AH49543 | E081_mCRF_FractionalMethylation.bigwig
##   AH49544 | E082_mCRF_FractionalMethylation.bigwig

unique(epiFiles$species) # sanity check

## [1] "Homo sapiens"

unique(epiFiles$genome) # sanity check

## [1] "hg19"

table(epiFiles$sourcetype) #types of files available

##
##    BED BigWig    GTF    tab    Zip
##   8298   9932      3      1     14

#more precise description of the files available:
head(sort(table(epiFiles$description), decreasing=TRUE))

##
## Bigwig File containing -log10(p-value) signal tracks from EpigenomeRoadMap Project
##                                                                                6881
## Bigwig File containing fold enrichment signal tracks from EpigenomeRoadMap Project
##                                                                                2947
##    Narrow ChIP-seq peaks for consolidated epigenomes from EpigenomeRoadMap Project
##                                                                                2894
##     Broad ChIP-seq peaks for consolidated epigenomes from EpigenomeRoadMap Project
##                                                                                2534
##    Gapped ChIP-seq peaks for consolidated epigenomes from EpigenomeRoadMap Project
##                                                                                2534
##        Narrow DNasePeaks for consolidated epigenomes from EpigenomeRoadMap Project
##                                                                                 131

#a more precise query:
query(ah , c("EpigenomeRoadMap","H3K36ME3","broadPeak","liver"))

## AnnotationHub with 5 records
## # snapshotDate(): 2017-10-27
## # $dataprovider: BroadInstitute
## # $species: Homo sapiens
## # $rdataclass: GRanges
## # additional mcols(): taxonomyid, genome, description,
## #   coordinate_1_based, maintainer, rdatadateadded, preparerclass,
## #   tags, rdatapath, sourceurl, sourcetype
## # retrieve records with, e.g., 'object[["AH29351"]]'
##
##             title
##   AH29351 | E066-H3K36me3.broadPeak.gz
##   AH41908 | BI.Adult_Liver.H3K36me3.3.broadPeak.gz
##   AH41909 | BI.Adult_Liver.H3K36me3.4.broadPeak.gz
##   AH41910 | BI.Adult_Liver.H3K36me3.5.broadPeak.gz
##   AH42615 | UCSD.Adult_Liver.H3K36me3.STL011.broadPeak.gz
```

```
k36Peaks=ah[["AH29351"]] #retrieve the data
```

```
## loading from cache '/home/pmartin//.AnnotationHub/34791'
```

```
k36Peaks # a GRanges object
```

```
## GRanges object with 199604 ranges and 5 metadata columns:
##            seqnames                 ranges strand |        name     score
##               <Rle>              <IRanges>  <Rle> | <character> <numeric>
##        [1]    chr12 [133758469, 133774384]      * |      Rank_1       158
##        [2]    chr19 [ 58695221,  58726784]      * |      Rank_2       134
##        [3]    chr19 [ 44670183,  44683357]      * |      Rank_3       133
##        [4]     chr9 [ 35694083,  35727997]      * |      Rank_4       131
##        [5]    chr19 [ 44509434,  44519252]      * |      Rank_5       131
##        ...      ...                    ...    ... .         ...       ...
##   [199600]     chr9 [  3634043,   3634277]      * | Rank_199600         0
##   [199601]     chr1 [182763503, 182763836]      * | Rank_199601         0
##   [199602]     chr2 [173674414, 173674905]      * | Rank_199602         0
##   [199603]     chr1 [117660710, 117661344]      * | Rank_199603         0
##   [199604]     chr3 [ 75749824,  75750091]      * | Rank_199604         0
##            signalValue    pValue    qValue
##              <numeric> <numeric> <numeric>
##        [1]     7.70772  18.84335  15.87884
##        [2]     6.41733  16.12752  13.42749
##        [3]     6.30442  15.98875  13.30686
##        [4]     6.47104  15.84522  13.15117
##        [5]     6.51529  15.79857  13.16733
##        ...         ...       ...       ...
##   [199600]     1.45074   1.00778   0.00937
##   [199601]     1.44989   1.00766   0.00929
##   [199602]     1.45868   1.00763   0.00937
##   [199603]     1.44478   1.00691   0.00885
##   [199604]     1.43371   1.00530   0.00789
##   -------
##   seqinfo: 93 sequences (1 circular) from hg19 genome
```

```
metadata(k36Peaks)
```

```
## $AnnotationHubName
## [1] "AH29351"
##
## $`File Name`
## [1] "E066-H3K36me3.broadPeak.gz"
##
## $`Data Source`
## [1] "http://egg2.wustl.edu/roadmap/data/byFileType/peaks/consolidated/broadPeak/E066-H3K36me3.broadPeak.g
##
## $Provider
## [1] "BroadInstitute"
##
## $Organism
## [1] "Homo sapiens"
##
## $`Taxonomy ID`
## [1] 9606
```

Another example with a chainfile: Lifting genomic coordinates from one genome build to another, requires a chain file. Here, we use the `liftOver` function from *rtracklayer package* to perform the lift over and use the chain file obtained from AnnotationHub.

```
query(ah,c("dm3","dm6","chainfile")) #search for a chain file

## AnnotationHub with 1 record
## # snapshotDate(): 2017-10-27
## # names(): AH15105
## # $dataprovider: UCSC
## # $species: Drosophila melanogaster
## # $rdataclass: ChainFile
## # $rdatadateadded: 2014-12-15
## # $title: dm3ToDm6.over.chain.gz
## # $description: UCSC liftOver chain file from dm3 to dm6
## # $taxonomyid: 7227
## # $genome: dm3
## # $sourcetype: Chain
## # $sourceurl: http://hgdownload.cse.ucsc.edu/goldenpath/dm3/liftOver/dm3T...
## # $sourcesize: NA
## # $tags: c("liftOver", "chain", "UCSC", "genome", "homology")
## # retrieve record with 'object[["AH15105"]]'

chain=ah[["AH15105"]] #retrieve the chain file

## loading from cache '/home/pmartin//.AnnotationHub/19200'

genes #Drosophila genes

## GRanges object with 2 ranges and 2 metadata columns:
##               seqnames               ranges strand |    EntrezId      Symbol
##                  <Rle>            <IRanges>  <Rle> | <character> <character>
##   FBgn0031208    chr2L [    7529,     9484]      + |       33155     CG11023
##   FBgn0085359     chrX [18962306, 18962925]      - |     2768869     CG34330
##   -------
##   seqinfo: 2 sequences from dm3 genome
```

```
liftOver(genes,chain) #new coordinates

## GRangesList object of length 2:
## $FBgn0031208
## GRanges object with 1 range and 2 metadata columns:
##       seqnames       ranges strand |    EntrezId      Symbol
##          <Rle>    <IRanges>  <Rle> | <character> <character>
##   [1]    chr2L [7529, 9484]      + |       33155     CG11023
##
## $FBgn0085359
## GRanges object with 1 range and 2 metadata columns:
##       seqnames               ranges strand | EntrezId  Symbol
##   [1]     chrX [19068273, 19068892]      - |  2768869 CG34330
##
## -------
## seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

Versions of the hub can be accessed and selected using:

```
possibleDates(ah)[1:10] # available dates for the Hub

##  [1] "2013-03-19" "2013-03-21" "2013-03-26" "2013-04-04" "2013-04-29"
##  [6] "2013-06-24" "2013-06-25" "2013-06-26" "2013-06-27" "2013-10-29"

snapshotDate(ah) #date currently in use (can be changed using <-)

## [1] "2017-10-27"
```

The location of the downloaded files and the Hub URL are accessed using:

```
hubCache(ah)

## [1] "/home/pmartin//.AnnotationHub"

hubUrl(ah)

## [1] "https://annotationhub.bioconductor.org"
```

The AnnotationHub can also be explored more interactively in a browser using:

```
d=display(ah)
```

7.4 The *biomaRt* package

The *biomaRt* package [21, 22] provides an interface to the Biomart web services. The so-called "Marts" represent a large family of annotation resources:

```
## library(biomaRt)
listMarts(host="www.ensembl.org")

##                biomart               version
## 1 ENSEMBL_MART_ENSEMBL      Ensembl Genes 93
## 2   ENSEMBL_MART_MOUSE      Mouse strains 93
## 3      ENSEMBL_MART_SNP  Ensembl Variation 93
## 4 ENSEMBL_MART_FUNCGEN Ensembl Regulation 93
```

Now can select the source (or "Mart") we want to use:

```
ens=useMart('ENSEMBL_MART_ENSEMBL',host='www.ensembl.org')
ens

## Object of class 'Mart':
##   Using the ENSEMBL_MART_ENSEMBL BioMart database
##   No dataset selected.

listDatasets(ens)[1:5,]

##                        dataset                         description
## 1        mpahari_gene_ensembl Shrew mouse genes (PAHARI_EIJ_v1.1)
## 2 aplatyrhynchos_gene_ensembl           Duck genes (BGI_duck_1.0)
## 3       mauratus_gene_ensembl    Golden Hamster genes (MesAur1.0)
## 4       cjacchus_gene_ensembl        Marmoset genes (ASM275486v1)
## 5   ptroglodytes_gene_ensembl      Chimpanzee genes (Pan_tro_3.0)
##           version
## 1 PAHARI_EIJ_v1.1
## 2    BGI_duck_1.0
## 3       MesAur1.0
## 4     ASM275486v1
## 5     Pan_tro_3.0

rattus=useMart('ENSEMBL_MART_ENSEMBL',
               host='www.ensembl.org',
               dataset='rnorvegicus_gene_ensembl')
```

Now we need to set up filters in order to get results focused on our objects of interest.

```
head(keytypes(rattus)) #see also listFilters(rattus)

## [1] "affy_rae230a"          "affy_rae230b"          "affy_raex_1_0_st_v1"
## [4] "affy_ragene_1_0_st_v1" "affy_ragene_2_1_st_v1" "affy_rat230_2"

head(columns(rattus)) #see also listAttributes(rattus)

## [1] "3utr"        "3_utr_end"   "3_utr_end"   "3_utr_start" "3_utr_start"
## [6] "5utr"
```

To get all the values ('keys') taken by a specific filter ('keytype'), we can use:

```
keys(rattus,keytype="chromosome_name")[1:5]

## [1] "1" "2" "3" "4" "5"
```

Unfortunately, the `keys` method does not work with all keytypes in *biomaRt*.

Once we know where to find the keys corresponding to our suject of interest, we use the `getBM` function to extract the information we selected:

```
goi=c('ENSRNOG00000012586','ENSRNOG00000018113') #genes of interest
getBM(attributes=c('ensembl_gene_id', 'strand',
                   'chromosome_name','start_position','end_position'),
                    filters = 'ensembl_gene_id',
                    values = goi, mart = rattus)

##      ensembl_gene_id strand chromosome_name start_position end_position
## 1 ENSRNOG00000012586     -1               5      172077282    172078760
## 2 ENSRNOG00000018113      1              17       66548818     66551947
```

Note that when ENSEMBL is used as *Mart*, other functions are available, such as `getSequence` and `getGene` which are wrappers to GetBM.

The `select` method presented previously also works on *Mart* objects:

```
select(rattus,keys=goi,keytype='ensembl_gene_id',
       columns=c('ensembl_gene_id', 'strand',
                 'chromosome_name','start_position','end_position'))

##      ensembl_gene_id strand chromosome_name start_position end_position
## 1 ENSRNOG00000012586     -1               5      172077282    172078760
## 2 ENSRNOG00000018113      1              17       66548818     66551947
```

The vignette of the *biomaRt* provides several practical examples of queries to Biomart resources.

7.5 The *GEOquery* and *SRAdb* packages

The *GEOquery* package [23] retrieves data from the Gene Expression Omnibus (GEO) repository and the *SRAdb* package retrieves data from the Short Read Archive (SRA) repository. These resources are of primary importance for functional genomics and contain thousands of datasets. *SRAdb* also provides some functions to open and interact with the Integrative Genome Browser (IGV) which are not illustrated here.

Extracting data from GEO.
In GEO, the data are organized as

- Platforms (GPLxxx identifiers): info on microarray designs and sequencing platforms
- Samples (GSMxxx identifiers): sample-level data and protocols
- Series (GSExxx identifiers): data and information from a same experiment/project
- Dataset (GDSxxx identifiers): statistically comparable data obtained from a unique platform

The *GEOmetadb* package (not used here) aims at faciliating the search for relevant entries in the GEO database.

To automatically download all the raw data from a GEO entry (files can be huge!) in a new folder created in your working directory, use:

```
library(GEOquery)
RawGSE13149=getGEOSuppFiles('GSE13149') #!! large files !!
```

The `getGEO` allows to retrieve different types of data from GEO (GPL, GSM, GSE, GDS). Here, for a Series, the data are downloaded to a temporary directory and stored in R as an *ExpressionSet*.

```
gse13149=getGEO('GSE13149')
show(gse13149) #Here only one Expression Set
GEOeset=gse13149[[1]] #get the Expression Set
```

GEOquery is essentially dedicated to microarray data. While GEO microarray datasets are imported as relevant *ExpressionSet* and *limma* *MAList* objects, NGS data cannot be imported with *GEOquery*. Note that the *ChIPseeker* provides interesting functions to download bed files from ChIP-seq experiment directly from GEO.

Extracting data from SRA.
On the other hand, the *SRAdb* package is dedicated to retrieve NGS datasets from the NCBI Short Read Archive, many of which are also present in GEO.
The data available in SRA are:

- Studies (SRP/ERP/DRP identifiers): metadata describing a sequencing project or study
- Experiment (SRX/ERX/DRX identifiers): metadata describing the libraries, platform selection and protocols used in a particular sequencing experiment. An experiment may contain several Runs.
- Run (SRR/ERR/DRR identifiers): sequencing data for a particular sequencing experiment
- Sample (SRS/ERS/DRS identifiers): metadata describing the physical sample that has been the suject of the sequencing process
- Analysis (SRZ identifiers): A BAM file resulting from an analysis and the metadata describing the analysis performed

All these 5 types of objects are also regrouped in a virtual container called *Accession* (SRA/ERA/DRA identifiers) which are used to track the submission.

The *SRAdb* provides an SRAdb SQLite file updated regularly which allows to query the database and find relevant content. The corresponding file can be downloaded and uncompressed using:

```
library(SRAdb)
sqlfile = 'SRAmetadb.sqlite'
if(!file.exists('SRAmetadb.sqlite')) sqlfile <<- getSRAdbFile() #large file!!
```

To create a connection to the file use:

```
sra_con = dbConnect(SQLite(),sqlfile)
```

Explore the content of the database:

```
dbListTables(sra_con) #Tables available in the database
dbListFields(sra_con,"study") #Fields for the study table
colDesc=colDescriptions(sra_con=sra_con) #Description of the fields
```

colDesc contains a description of the different fields and their default values.

Query the database using SQL:

```
rs = dbGetQuery(sra_con,"select * from study limit 3")
rs[, 1:3]
##   study_ID              study_alias study_accession
## 1        1                DRP000001       DRP000001
## 2        2                DRP000002       DRP000002
## 3        3 DLD1_normoxia_nucleosome       DRP000003
```

More examples in the *SRAdb* vignette.

Convert SRA identifiers and get their links. For examples, which runs and samples are associated with this study?:

```
sraConvert( 'SRP001007', sra_con = sra_con )
##       study submission    sample experiment       run
## 1 SRP001007  SRA009276 SRS004650  SRX007396 SRR020740
## 2 SRP001007  SRA009276 SRS004650  SRX007396 SRR020739
```

Queries with (getSRA):

```
rs = getSRA( search_terms = "RNASeq",
              out_types = c('study'), sra_con )
dim(rs)
## [1] 1172   12
head(rs[,1:2])
##   study_alias     study
## 1   DRP000366 DRP000366
## 2   PRJDB2653 DRP000375
## 3   PRJDB2395 DRP000376
## 4   PRJDB2135 DRP000535
## 5   PRJDB2122 DRP000536
## 6   PRJDB2739 DRP000537
```

More examples in the *SRAdb* vignette.

The ftp adresses of the files can be obtained with:

```
listSRAfile( c("SRX000122"), sra_con, fileType = 'sra' )
#or: getSRAinfo("SRX000122",sra_con,sraType="sra")
```

The function `getSRAfile` allows to download SRA accessions from NCBI SRA and Fastq files from EBI ENA.

Disconnect from the database with:

```
dbDisconnect(sra_con)
```

8 Visualization of genomic data

8.1 Introduction to *R* graphics for genomic data

Visualizing NGS data in specific genomic context is an excellent, although not sufficient, mean to assess data quality or the robustness and specificity of findings obtained from data analysis. A number of Genome Browsers are available for this task, such as UCSC Genome Browser, Gbrowse or IGV or jbrowse. They allow to interactively 'navigate' the genome and to display several annotation and data types. However, batch extraction of genome views is not always available. Here, we briefly illustrate some tools available in *Bioconductor* to assemble data in genome browser-like plots.

R is quite famous for the excellent quality of its graphics and its graphical capabilities in general (see CRAN Task view on graphics). R base graphics already provide a number of functionalities. The *lattice* package has provided a complementary approach. Later, the *ggplot2* package [24] has implemented the concepts presented in the book 'The Grammar of Graphics' [25] and is now widely used (see an example at paragraph 4.3). The *ggvis* further adds interactivity and web graphics to *ggplot2*.

In *Bioconductor* specific packages have been developed to plot genomic data and annotations (see *Bioconductor* Visualization View), in particular:

- *Gviz* which is illustrated in paragraph 8.2
- *ggbio* [26] which extends *ggplot2* to genomic data visualization
- *genomation* which has some interesting functionalities to plot heatmaps and average profiles
- *Sushi*, which is particularly good to generate multipanel figures

Interactive graphs are increasingly being developped notably based on *shiny*.

8.2 The *Gviz* package

The *Gviz* package vignette is extremely well documented. Here we only briefly illustrate some basic functionnalities but the package allows much more and protein-related tracks can also be added using the *Pviz* package. *Gviz* defines a set of classes which support a wide range of standard formats (see Table 1) and can be created from the typical *Bioconductor* objects used for genomic data (see Table 2).

The first step is to define the region we are interested in. We are going to focus on the region of Drosophila melanogaster chromosome 4 previously defined in paragraph 5.2.1. First we define a *GRanges* object corresponding to this region:

```
ROI=GRanges(seqnames="chr4",ranges=which$chr4[1],strand="*")
```

Let's search for the motif 'TATAAA' in this region:

Gviz class	File type	Extension	Details
Annotation Track	BED	.bed	Fields `chrom`, `chromStart`, `chromEnd`, `strand`, `name` and `itemRbg` are used only (first 3 fields are mandatory)
	GFF	.gff, .gff1	Fields `seqname`, `start`, `end`, `strand`, `feature` and `group` are recognized
	GFF2	.gff2	Same as above. Feature grouping can be provided as `Group` or `Parent` attribute
	GFF3	.gff3	Same as above but feature grouping information has to be provided as the `Parent` attribute
	BAM	.bam	Streaming available. `Start`, `end` and `strand` information for the reads are used. Reads ids are used for track item grouping
GeneRegion Track	GTF	.gtf	Gene, transcript and exon ids and names can be parsed from the `gene_id`, `gene_name`, `transcript_id`, `transcript_name`, `exon_id` or `exon_name attributes`
	GFF	.gff, .gff1	Supports very limited grouping. Not adapted to encode complete gene models.
	GFF2	.gff2	Same as GTF. Files could be renamed .gtf
	GFF3	.gff3	The gene-to-transcript and transcript-to-exon relationships are encoded in the `parent` and `type` attribute. Most gff3 variants are supported.
DataTrack Track	BedGraph	.bedGraph	
	WIG	.wig	
	BigWig	.bigWig, .bw	Streaming available.
	BAM	.bam	Streaming available. Read coverage only is extracted from the BAM file.
Sequence Track	FASTA	.fa, .fasta	Streaming available only if an index file is found in the same directory as the fasta file.
	2Bit	.2bit	Streaming available.
Alignments Track	BAM	.bam	Streaming available. Always needs an index file in the same directory as the BAM file

Table 1: **Gviz Track classes and standard NGS formats**

```
TATAAA_on_ROI=shift(union(ranges(matchPattern('TATAAA',
                                              subseq(Dmelanogaster$chr4,
                                                     start=start(ROI),
                                                     end=end(ROI)))),
                          ranges(matchPattern('TTTATA',
                                              subseq(Dmelanogaster$chr4,
                                                     start=start(ROI),
                                                     end=end(ROI))))),
                    start(ROI))
TATAAAgr=GRanges(seqnames="chr4",
                 ranges=TATAAA_on_ROI,
                 strand="*")
```

From this *GRanges* object, we can define an *AnnotationTrack*:

Gviz class	*Bioconductor* class
Annotation Track	*data.frame*
	IRanges
	GRanges
	GRangesList
GeneRegion Track	*data.frame*
	IRanges
	GRanges
	GRangesList
	TxDb
DataTrack Track	*data.frame*
	IRanges
	GRanges
Sequence Track	*DNAStringSet*
	BSgenome

Table 2: **Gviz Track classes and standard *Bioconductor* objects**

```
atrack=AnnotationTrack(TATAAAgr,name="TATAAA motif")
```

Next, we create a genome axis (i.e. genomic coordinates) track:

```
gtrack=GenomeAxisTrack()
```

An ideogram of the chromosome can be downloaded from UCSC using:

```
itrack=IdeogramTrack(genome="dm3", chromosome="chr4")
#I'm having some unresolved issues with these ideograms
#so I don't use them below
```

Let's take a look at our plot at this point (Figure 17):

```
plotTracks(list(gtrack,atrack)) #add itrack if possible
```

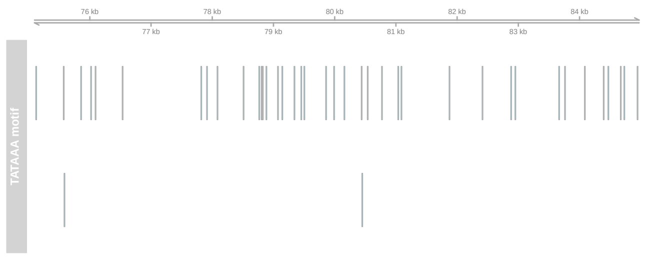


Figure 17: **Genome axis and TATAAA Annotationtracks**

Then, we import Gene models from the *TxDb.Dmelanogaster.UCSC.dm3.ensGene* using:

```
grtrack=GeneRegionTrack(TxDb.Dmelanogaster.UCSC.dm3.ensGene,
                        start=start(ROI),
                        end=end(ROI),
                        genome="dm3",chromosome="chr4",
                        name="Gene Model")
```

and sequence information using *BS.genome.Dmelanogaster.UCSC.dm3*:

```
strack=SequenceTrack(Dmelanogaster, chromosome="chr4")
```

This gives Figure 18 where the sequence track is not visible (too small):

```
plotTracks(list(gtrack,atrack,grtrack,strack))
```

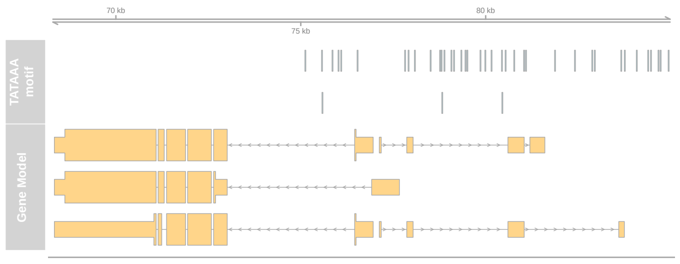


Figure 18: Visualizing Gene and Sequence Tracks

We can zoom in and out using for example (Figure 19):

```
plotTracks(list(gtrack,atrack,grtrack),
           extend.left = 0.5, extend.right = 10000)
```

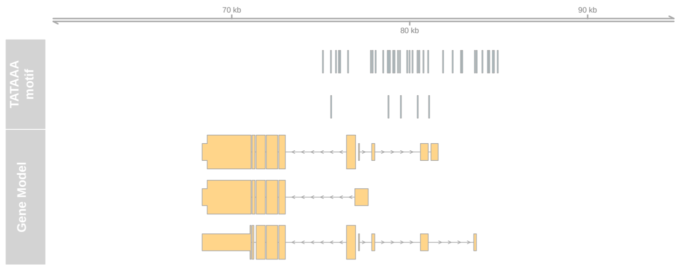


Figure 19: Zomming in and out

or (Figure 20):

```
plotTracks(list(gtrack,atrack,strack),
           from = 79900, to = 80100)
```

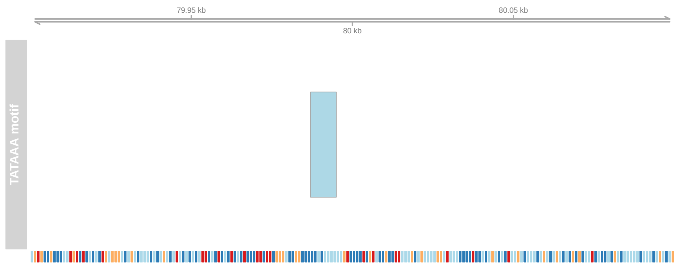


Figure 20: Zomming in

If we zoom enough we get the actual sequence (Figure 21):

```
plotTracks(list(gtrack,atrack,strack),
           from = 79975, to = 80015)
```

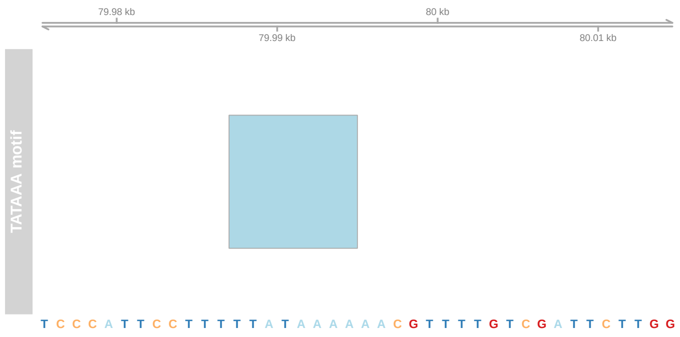


Figure 21: Zomming in some more to read the sequence

Now we import some paired-end reads from a BAM file:

```
altrack=AlignmentsTrack(pr_bamFile, isPaired=TRUE)
```

And visualize these reads (Figure 22):

```
plotTracks(list(gtrack,atrack,
                grtrack,altrack)) #use type="coverage" to see only the coverage
```

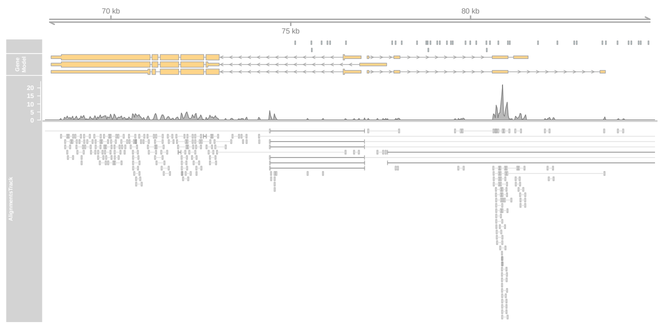


Figure 22: Visualizing Alignment Tracks

With a zoom and some more options (Figure 23):

```
plotTracks(list(gtrack,atrack,grtrack,altrack,strack),
           from=72000,to=73500,
           col.mates="purple",
           col.gaps="orange")
```

Gviz has lots of functionalities to plot quantitative data associated to genomic positions (*DataTrack*). Here we provide a simple illustration based on simulated data:

```
set.seed(255)
lim <- c(start(ROI), end(ROI))
coords <- sort(c(lim[1], sample(seq(from=lim[1], to=lim[2]), 99), lim[2]))
```

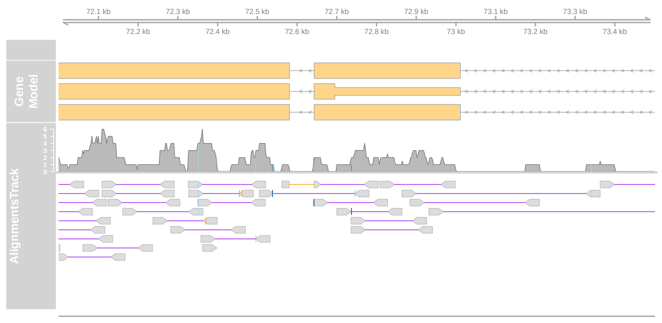


Figure 23: Visualizing Alignment Tracks

```
c1=runif(100, min=-10, max=8)
c2=runif(100, min=-10, max=8)
dat=GRanges(seqnames="chr4",strand="*",
            ranges=IRanges(start=coords[-length(coords)], end=coords[-1]),
            ctrol1=c1,ctrol2=c2,
            treated1=c1+rnorm(100,3),
            treated2=c2+rnorm(100,3))
genome(dat)="dm3"
dtrack <- DataTrack(dat,name="Uniform")
```

Which gives Figure 24

```
plotTracks(list(gtrack, atrack, grtrack, dtrack),
           from=lim[1], to=lim[2], type=c("a","p"),
           groups=rep(c("ctrol","treated"),each=2))
```

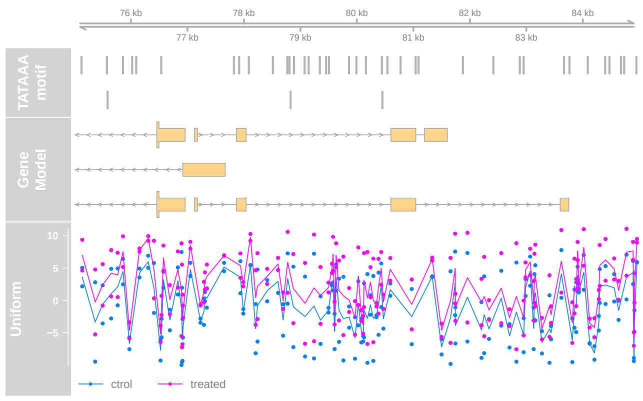


Figure 24: Visualizing Data Tracks

The *Gviz* package offers many more functionalities that are well illustrated in the package vignette.

9 Session info

- R version 3.4.4 (2018-03-15), `x86_64-pc-linux-gnu`
- Locale: `LC_CTYPE=fr_FR.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=fr_FR.UTF-8`, `LC_COLLATE=fr_FR.UTF-8`, `LC_MONETARY=fr_FR.UTF-8`, `LC_MESSAGES=fr_FR.UTF-8`, `LC_PAPER=fr_FR.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=fr_FR.UTF-8`, `LC_IDENTIFICATION=C`
- Running under: `Ubuntu 14.04.5 LTS`
- Matrix products: default
- BLAS: `/usr/lib/libblas/libblas.so.3.0`
- LAPACK: `/usr/lib/lapack/liblapack.so.3.0`
- Base packages: base, datasets, graphics, grDevices, grid, methods, parallel, stats, stats4, utils
- Other packages: ade4 1.7-11, AnnotationDbi 1.40.0, AnnotationHub 2.10.1, Biobase 2.38.0, BiocGenerics 0.24.0, BiocParallel 1.12.0, BiocStyle 2.6.1, biomaRt 2.34.2, Biostrings 2.46.0, bitops 1.0-6, BSgenome 1.46.0, BSgenome.Dmelanogaster.UCSC.dm3 1.4.0, BSgenome.Dmelanogaster.UCSC.dm3.masked 1.3.99, BSgenome.Hsapiens.UCSC.hg19 1.4.0, DelayedArray 0.4.1, drosophila2cdf 2.18.0, drosophila2.db 3.2.3, drosophila2probe 2.18.0, GenomeInfoDb 1.14.0, GenomicAlignments 1.14.2, GenomicFeatures 1.30.3, GenomicRanges 1.30.3, GEOquery 2.46.15, ggplot2 3.0.0, GO.db 3.5.0, graph 1.56.0, grImport 0.9-0, Gviz 1.22.3, hom.Dm.inp.db 3.1.2, IRanges 2.12.0, JASPAR2018 1.0.0, knitr 1.20, matrixStats 0.53.1, MMDiffBamSubset 1.14.0, MotifDb 1.20.0, motifStack 1.22.4, MotIV 1.34.0, org.Dm.eg.db 3.5.0, pasillaBamSubset 0.16.0, RCurl 1.95-4.11, Rqc 1.12.0, Rsamtools 1.30.0, RSQLite 2.1.0, rtracklayer 1.38.3, S4Vectors 0.16.0, seqLogo 1.44.0, ShortRead 1.36.1, SRAdb 1.40.0, SummarizedExperiment 1.8.1, TFBSTools 1.16.0, TxDb.Dmelanogaster.UCSC.dm3.ensGene 3.2.2, XML 3.98-1.11, XVector 0.18.0
- Loaded via a namespace (and not attached): acepack 1.4.1, annotate 1.56.2, AnnotationFilter 1.2.0, assertthat 0.2.0, backports 1.1.2, base64enc 0.1-3, bindr 0.1.1, bindrcpp 0.2.2, BiocInstaller 1.28.0, biovizBase 1.26.0, bit 1.1-12, bit64 0.9-7, blob 1.1.1, caTools 1.17.1, checkmate 1.8.5, cluster 2.0.7-1, CNEr 1.14.0, codetools 0.2-15, colorspace 1.3-2, compiler 3.4.4, crayon 1.3.4, curl 3.2, data.table 1.11.0, DBI 1.0.0, dichromat 2.0-0, digest 0.6.15, DirichletMultinomial 1.20.0, dplyr 0.7.4, ensembldb 2.2.2, evaluate 0.10.1, foreign 0.8-70, Formula 1.2-2, GenomeInfoDbData 1.0.0, GenomicFiles 1.14.0, glue 1.3.0, gridExtra 2.3, gtable 0.2.0, gtools 3.5.0, highr 0.6, Hmisc 4.1-1, hms 0.4.2, htmlTable 1.11.2, htmltools 0.3.6, htmlwidgets 1.2, httpuv 1.4.1, httr 1.3.1, hwriter 1.3.2, interactiveDisplayBase 1.16.0, KEGGREST 1.18.1, later 0.7.2, lattice 0.20-35, latticeExtra 0.6-28, lazyeval 0.2.1, limma 3.34.9, magrittr 1.5, markdown 0.8, MASS 7.3-50, Matrix 1.2-14, memoise 1.1.0, mime 0.5, munsell 0.5.0, nnet 7.3-12, pillar 1.2.2, pkgconfig 2.0.1, plyr 1.8.4, png 0.1-7, poweRlaw 0.70.1, prettyunits 1.0.2, progress 1.2.0, promises 1.0.1, ProtGenerics 1.10.0, purrr 0.2.5, R6 2.2.2, RColorBrewer 1.1-2, Rcpp 0.12.18, readr 1.1.1, reshape2 1.4.3, rGADEM 2.26.0, rlang 0.2.2, rmarkdown 1.9, R.methodsS3 1.7.1, RMySQL 0.10.15, R.oo 1.22.0, rpart 4.1-13, rprojroot 1.3-2, rstudioapi 0.7, R.utils 2.6.0, scales 1.0.0,

shiny 1.0.5, splines 3.4.4, splitstackshape 1.4.4, stringi 1.2.4, stringr 1.3.1, survival 2.42-3, TFMPvalue 0.0.6, tibble 1.4.2, tidyr 0.8.0, tools 3.4.4, VariantAnnotation 1.24.5, VGAM 1.0-5, withr 2.1.2, xml2 1.2.0, xtable 1.8-2, yaml 2.1.19, zlibbioc 1.24.0

References

[1] S. Tippmann. Programming tools: Adventures with R. *Nature*, 517(7532):109–110, Jan 2015. [DOI:10.1038/517109a] [PubMed:25557714].

[2] Robert C Gentleman, Vincent J. Carey, Douglas M. Bates, and others. Bioconductor: Open software development for computational biology and bioinformatics. *Genome Biology*, 5:R80, 2004. URL: http://genomebiology.com/2004/5/10/R80.

[3] W. Huber, V. J. Carey, R. Gentleman, S. Anders, M. Carlson, B. S. Carvalho, H. C. Bravo, S. Davis, L. Gatto, T. Girke, R. Gottardo, F. Hahne, K. D. Hansen, R. A. Irizarry, M. Lawrence, M. I. Love, J. MacDonald, V. Obenchain, A. K. Ole, H. Pages, A. Reyes, P. Shannon, G. K. Smyth, D. Tenenbaum, L. Waldron, and M. Morgan. Orchestrating high-throughput genomic analysis with Bioconductor. *Nat. Methods*, 12(2):115–121, Jan 2015. [DOI:10.1038/nmeth.3252] [PubMed:25633503].

[4] G. Tan and B. Lenhard. TFBSTools: an R/bioconductor package for transcription factor binding site analysis. *Bioinformatics*, 32(10):1555–1556, 05 2016.

[5] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, Mar 2010. [PubMed Central:PMC2828108] [DOI:10.1093/bioinformatics/btp698] [PubMed:20080505].

[6] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, Jul 2009. [PubMed Central:PMC2705234] [DOI:10.1093/bioinformatics/btp324] [PubMed:19451168].

[7] 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(3):R25, 2009. [PubMed Central:PMC2690996] [DOI:10.1186/gb-2009-10-3-r25] [PubMed:19261174].

[8] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nat. Methods*, 9(4):357–359, Apr 2012. [PubMed Central:PMC3322381] [DOI:10.1038/nmeth.1923] [PubMed:22388286].

[9] M. Pertea, D. Kim, G. M. Pertea, J. T. Leek, and S. L. Salzberg. Transcript-level expression analysis of RNA-seq experiments with HISAT, StringTie and Ballgown. *Nat Protoc*, 11(9):1650–1667, 09 2016.

[10] D. Kim, B. Langmead, and S. L. Salzberg. HISAT: a fast spliced aligner with low memory requirements. *Nat. Methods*, 12(4):357–360, Apr 2015.

[11] A. Dobin, C. A. Davis, F. Schlesinger, J. Drenkow, C. Zaleski, S. Jha, P. Batut, M. Chaisson, and T. R. Gingeras. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics*, 29(1):15–21, Jan 2013. [PubMed Central:PMC3530905] [DOI:10.1093/bioinformatics/bts635] [PubMed:23104886].

[12] Y. Liao, G. K. Smyth, and W. Shi. The Subread aligner: fast, accurate and scalable read mapping by seed-and-vote. *Nucleic Acids Res.*, 41(10):e108, May 2013. [PubMed Central:PMC3664803] [DOI:10.1093/nar/gkt214] [PubMed:23558742].

[13] U. Bodenhofer, E. Bonatesta, C. Horej?-Kainrath, and S. Hochreiter. msa: an R package for multiple sequence alignment. *Bioinformatics*, 31(24):3997–3999, Dec 2015.

[14] M. Lawrence, W. Huber, H. Pages, P. Aboyoun, M. Carlson, R. Gentleman, M. T. Morgan, and V. J. Carey. Software for computing and annotating genomic ranges. *PLoS Comput. Biol.*, 9(8):e1003118, 2013. [PubMed Central:PMC3738458] [DOI:10.1371/journal.pcbi.1003118] [PubMed:23950696].

[15] M. Morgan, S. Anders, M. Lawrence, P. Aboyoun, H. Pages, and R. Gentleman. ShortRead: a bioconductor package for input, quality assessment and exploration of high-throughput sequence data. *Bioinformatics*, 25(19):2607–2608, Oct 2009. [PubMed Central:PMC2752612] [DOI:10.1093/bioinformatics/btp450] [PubMed:19654119].

[16] K. F. Au, H. Jiang, L. Lin, Y. Xing, and W. H. Wong. Detection of splice junctions from paired-end RNA-seq data by SpliceMap. *Nucleic Acids Res.*, 38(14):4570–4578, Aug 2010. [PubMed Central:PMC2919714] [DOI:10.1093/nar/gkq211] [PubMed:20371516].

[17] Y. Liao, G. K. Smyth, and W. Shi. featureCounts: an efficient general purpose program for assigning sequence reads to genomic features. *Bioinformatics*, 30(7):923–930, Apr 2014. [DOI:10.1093/bioinformatics/btt656] [PubMed:24227677].

[18] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, Aug 2009. [PubMed Central:PMC2723002] [DOI:10.1093/bioinformatics/btp352] [PubMed:19505943].

[19] A. C. Berglund, E. Sjolund, G. Ostlund, and E. L. Sonnhammer. InParanoid 6: eukaryotic ortholog clusters with inparalogs. *Nucleic Acids Res.*, 36(Database issue):D263–266, Jan 2008. [PubMed Central:PMC2238924] [DOI:10.1093/nar/gkm1020] [PubMed:18055500].

[20] M. Lawrence, R. Gentleman, and V. Carey. rtracklayer: an R package for interfacing with genome browsers. *Bioinformatics*, 25(14):1841–1842, Jul 2009. [PubMed Central:PMC2705236] [DOI:10.1093/bioinformatics/btp328] [PubMed:19468054].

[21] S. Durinck, Y. Moreau, A. Kasprzyk, S. Davis, B. De Moor, A. Brazma, and W. Huber. BioMart and Bioconductor: a powerful link between biological databases and microarray data analysis. *Bioinformatics*, 21(16):3439–3440, Aug 2005. [DOI:10.1093/bioinformatics/bti525] [PubMed:16082012].

[22] S. Durinck, P. T. Spellman, E. Birney, and W. Huber. Mapping identifiers for the integration of genomic datasets with the R/Bioconductor package biomaRt. *Nat Protoc*, 4(8):1184–1191, 2009. [PubMed Central:PMC3159387] [DOI:10.1038/nprot.2009.97] [PubMed:19617889].

[23] S. Davis and P. S. Meltzer. GEOquery: a bridge between the Gene Expression Omnibus (GEO) and BioConductor. *Bioinformatics*, 23(14):1846–1847, Jul 2007. [DOI:10.1093/bioinformatics/btm254] [PubMed:17496320].

[24] Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009. URL: http://had.co.nz/ggplot2/book.

[25] Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[26] T. Yin, D. Cook, and M. Lawrence. ggbio: an R package for extending the grammar of graphics for genomic data. *Genome Biol.*, 13(8):R77, 2012. [PubMed Central:PMC4053745] [DOI:10.1186/gb-2012-13-8-r77] [PubMed:22937822].

[27] C. Trapnell, L. Pachter, and S. L. Salzberg. TopHat: discovering splice junctions with RNA-Seq. *Bioinformatics*, 25(9):1105–1111, May 2009. [PubMed Central:PMC2672628] [DOI:10.1093/bioinformatics/btp120] [PubMed:19289445].

[28] D. Kim, G. Pertea, C. Trapnell, H. Pimentel, R. Kelley, and S. L. Salzberg. TopHat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions. *Genome Biol.*, 14(4):R36, 2013. [PubMed Central:PMC4053844] [DOI:10.1186/gb-2013-14-4-r36] [PubMed:23618408].

[29] Y. Zhu, R. M. Stephens, P. S. Meltzer, and S. R. Davis. SRAdb: query and use public next-generation sequencing data from within R. *BMC Bioinformatics*, 14:19, Jan 2013.

[30] P. T. Pyl, J. Gehring, B. Fischer, and W. Huber. h5vc: scalable nucleotide tallies with HDF5. *Bioinformatics*, 30(10):1464–1466, May 2014.

[31] M. Lawrence and M. Morgan. Scalable Genomics with R and Bioconductor. *Statistical Science*, 29(2):214–226, 2014.