csaw: ChIP-seq analysis with windows User's Guide

Aaron T. L. Lun[1]

[1]The Walter and Eliza Hall Institute of Medical Research, Melbourne, Australia

October 30, 2020

Abstract

This document contains instructions on how to use *csaw* for differential binding analyses of ChIP-seq data with windows. It covers read counting into windows, filtering for windows of interest, normalization of sample-specific biases, variance modelling and hypothesis testing, summarization of windows into regions, and visualization and annotation of detected regions.

First edition: 15 August 2012
Last revised: 2 June 2019
Last compiled: 30 October 2020

Package

csaw 1.24.0

Contents

Chapter 1

Introduction

1.1 Scope

This document describes the *Bioconductor* package *csaw* for detecting differential binding (DB) in ChIP-seq experiments. *csaw* uses sliding windows to identify significant changes in binding patterns for transcription factors (TFs) or histone marks across different biological conditions [1]. However, it can also be applied to any sequencing technique where reads represent coverage of enriched genomic regions.

The descriptions in this user's guide are detailed and will explore the theoretical and practical motivations behind each step of a *csaw* analysis. While all users are welcome to read it from start to finish, new users may prefer to examine the case studies presented in the *chipseqDB* package [2], which provides the important information in a more concise format. Experienced users (or those looking for some nighttime reading!) are more likely to benefit from the in-depth discussions in this document.

The statistical methods described here are based upon those in the *edgeR* package [3]. Knowledge of *edgeR* is useful but not a prerequesite for reading this guide.

1.2 How to get help

Most questions about *csaw* should be answered by the documentation. Every function mentioned in this guide has its own help page. For example, a detailed description of the arguments and output of the `windowCounts` function can be obtained by typing `?windowCounts` or `help(windowCounts)` at the *R* prompt. Further detail on the methods or the underlying theory can be found in the references at the bottom of each help page.

The authors of the package always appreciate receiving reports of bugs in the package functions or in the documentation. The same goes for well-considered suggestions for improvements. Other questions about how to use *csaw* are best sent to the Bioconductor support site at https://support.bioconductor.org. Please send requests for general assistance and advice to the support site, rather than to the individual authors. Users posting to the support site for the first time may find it helpful to read the posting guide at http://www.bioconductor.org/help/support/posting-guide.

1.3 How to cite this package

Most users of *csaw* should cite the following in any publications:

> A. T. Lun and G. K. Smyth. csaw: a Bioconductor package for differential binding analysis of ChIP-seq data using sliding windows. *Nucleic Acids Res.*, 44(5):e45, Mar 2016

Anyone who uses the Bioconductor workflow to construct their analyses should also cite:

> A. T. L. Lun and G. K. Smyth. From reads to regions: a Bioconductor workflow to detect differential binding in ChIP-seq data. *F1000Research*, 4, 2015

For people interested in combined p-values, their use in DB analyses was proposed in:

> A. T. Lun and G. K. Smyth. De novo detection of differentially bound regions for ChIP-seq data using peaks and windows: controlling error rates correctly. *Nucleic Acids Res.*, 42(11):e95, Jul 2014

The DB analyses shown here use methods from the *edgeR* package, which has its own citation recommendations. See the appropriate section of the *edgeR* user's guide for more details.

1.4 Quick start

A typical ChIP-seq analysis in *csaw* would look something like that described below. This assumes that a vector of file paths to sorted and indexed BAM files is provided in `bam.files` and a design matrix in supplied in `design`. The code is split across several steps:

1. Loading in data from BAM files.

```
library(csaw)
param <- readParam(minq=20)
data <- windowCounts(bam.files, ext=110, width=10, param=param)
```

2. Filtering out uninteresting regions.

```
library(edgeR)
keep <- aveLogCPM(asDGEList(data)) >= -1
data <- data[keep,]
```

3. Calculating normalization factors.

```
binned <- windowCounts(bam.files, bin=TRUE, width=10000, param=param)
data <- normFactors(binned, se.out=data)
```

4. Identifying DB windows.

```
y <- asDGEList(data)
y <- estimateDisp(y, design)
fit <- glmQLFit(y, design, robust=TRUE)
results <- glmQLFTest(fit)
```

5. Correcting for multiple testing.

```
        merged <- mergeResults(data, results$table, tol=1000L)
```

Chapter 2

Converting reads to counts

Hello, reader. A little box like this will be present at the start of each chapter. The idea is to list the objects from previous chapters that are needed to run the code in the current chapter. Hopefully, it'll provide a nice segue between chapters for tired eyes.

2.1 Types of input data

Sorted and indexed BAM (i.e., binary SAM) files [5] are required as input into the read counting functions in *csaw*. Sorting should be performed on the genomic coordinates of the mapped reads. Each read should only have one alignment in the file, i.e., secondary alignments should not be present.

For a given BAM file named 'xxx.bam', the corresponding index file should be named as 'xxx.bam.bai' in the same directory. The sensibility of the supplied index is not checked prior to counting. A common mistake is to replace or update the BAM file without updating the index. This will cause *csaw* to return incorrect results when it attempts to load alignments from the new BAM file.

In this guide, the behavior of each step will be demonstrated with some publicly available data from the *chipseqDBData* package. The dataset below focuses on changes in the binding profile of the NF-YA transcription factor between embryonic stem cells and terminal neurons [6]. This will be used as a case study for most of the code examples throughout the guide.

```
library(chipseqDBData)
tf.data <- NFYAData()
tf.data
```

```
## DataFrame with 5 rows and 3 columns
##          Name   Description      Path
##   <character>   <character>    <List>
## 1   SRR074398 NF-YA ESC (1) <BamFile>
## 2   SRR074399 NF-YA ESC (2) <BamFile>
## 3   SRR074417  NF-YA TN (1) <BamFile>
## 4   SRR074418  NF-YA TN (2) <BamFile>
## 5   SRR074401         Input <BamFile>
```

```
bam.files <- head(tf.data$Path, -1) # skip the input.
bam.files
```

```
## List of length 4
```

2.2 Counting reads into windows

2.2.1 Overview

The `windowCounts` function uses a sliding window approach to count fragments for a set of libraries. For single-end data, the fragment corresponding to a read is imputed by directionally extending each read to the average fragment length. We define a window as a fixed-width genomic interval, and we count the number of fragments overlapping that window in each library. This is repeated after sliding the window along the genome to a new position. A count is then obtained for each window in each library, thus quantifying protein binding intensity across the genome.

```
frag.len <- 110
win.width <- 10
param <- readParam(minq=20)
data <- windowCounts(bam.files, ext=frag.len, width=win.width, param=param)
```

The function returns a *RangedSummarizedExperiment* object where the matrix of counts is stored as the first entry in the `assays` slot. Each row corresponds to a genomic window while each column corresponds to a library. The coordinates of each window are stored in the `rowRanges` slot. The total number of reads in each library (also referred to as the library size) is stored as `totals` in the `colData` slot.

```
head(assay(data))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    3    3    4
## [2,]    3    6    3    4
## [3,]    5    5    0    0
## [4,]    4    7    0    0
## [5,]    5    9    0    0
## [6,]    3    9    0    0
```

```
head(rowRanges(data))
```

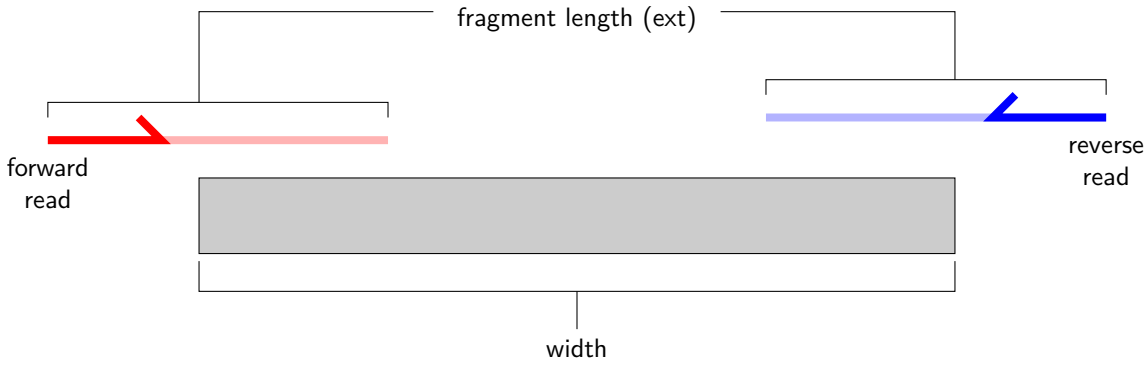
```
## GRanges object with 6 ranges and 0 metadata columns:
##       seqnames          ranges strand
##          <Rle>       <IRanges>  <Rle>
##   [1]     chr1 3003701-3003710      *
##   [2]     chr1 3003751-3003760      *
##   [3]     chr1 3003801-3003810      *
##   [4]     chr1 3003951-3003960      *
##   [5]     chr1 3004001-3004010      *
##   [6]     chr1 3004051-3004060      *
##   -------
```

```
##   seqinfo: 66 sequences from an unspecified genome

data$totals

## [1] 18363361 21752369 25095004 24104691
```

For single-end data, we estimate the average fragment length from a cross-correlation plot (see Section 2.4) for use as `ext`. Alternatively, the length can be estimated from diagnostics during ChIP or library preparation, e.g., post-fragmentation gel electrophoresis images. Typical values range from 100 to 300 bp, depending on the efficiency of sonication and the use of size selection steps in library preparation.



The `width` argument defines the window size, which we interpret as the width of the binding site for the target protein. This is user-specified and has important implications for the power and resolution of a DB analysis, which are discussed in Section 2.5. For TF analyses with small windows, the choice of spacing interval will also be affected by the choice of window size – see Section 2.2.5 for more details.

2.2.2 Filtering out low-quality reads

Read extraction from the BAM files is controlled with the `param` argument in `windowCounts`. This takes a *readParam* object that specifies a number of extraction parameters. The idea is to define the *readParam* object once in the entire analysis pipeline, which is then reused for all relevant functions. This ensures that read loading is consistent throughout the analysis.

```
param

##     Extracting reads in single-end mode
##     Duplicate removal is turned off
##     Minimum allowed mapping score is 20
##     Reads are extracted from both strands
##     No restrictions are placed on read extraction
##     No regions are specified to discard reads
```

In the example above, reads are filtered out based on the minimum mapping score with the `minq` argument. Low mapping scores are indicative of incorrectly and/or non-uniquely aligned sequences. Removal of these reads is highly recommended as it will ensure that only

the reliable alignments are supplied to *csaw*. The exact value of the threshold depends on the range of scores provided by the aligner. The *subread* program [7] was used to align the reads in this dataset, so a value of 20 might be appropriate.

Reads mapping to the same genomic position can be marked as putative PCR duplicates using software like the *MarkDuplicates* program from the *Picard* suite. Marked reads in the BAM file can be ignored during counting by setting `dedup=TRUE` in the *readParam* object. This reduces the variability caused by inconsistent amplification between replicates, and avoid spurious duplicate-driven DB between groups. An example of counting with duplicate removal is shown below, where fewer reads are used from each library relative to `data$totals`.

```
dedup.param <- readParam(minq=20, dedup=TRUE)
demo <- windowCounts(bam.files, ext=frag.len, width=win.width,
                     param=dedup.param)
demo$totals
```

```
## [1] 14799759 14773466 17424949 20386739
```

Duplicate removal is generally not recommended for routine DB analyses. This is because it caps the number of reads at each position, reducing DB detection power in high-abundance regions. Spurious differences may also be introduced when the same upper bound is applied to libraries of varying size. However, it may be unavoidable in some cases, e.g., involving libraries generated from low quantities of DNA. Duplicate removal is also acceptable for paired-end data, as exact overlaps for both paired reads are required to define duplicates. This greatly reduces the probability of incorrectly discarding read pairs from non-duplicate DNA fragments (assuming that a pair-aware method was used during duplicate marking).

2.2.3 Avoiding problematic genomic regions

Read extraction and counting can be restricted to particular chromosomes by specifying the names of the chromosomes of interest in `restrict`. This avoids the need to count reads on unassigned contigs or uninteresting chromosomes, e.g., the mitochondrial genome for ChIP-seq studies targeting nuclear factors. Alternatively, it allows `windowCounts` to work on huge datasets or in limited memory by analyzing only one chromosome at a time.

```
restrict.param <- readParam(restrict=c("chr1", "chr10", "chrX"))
```

Reads lying in certain regions can also be removed by specifying the coordinates of those regions in `discard`. This is intended to remove reads that are wholly aligned within known repeat regions but were not removed by the `minq` filter. Repeats are problematic as changes in repeat copy number or accessibility between conditions can lead to spurious DB. Removal of reads within repeat regions can avoid detection of these irrelevant differences.

```
repeats <- GRanges("chr1", IRanges(3000001, 3041000)) # telomere
discard.param <- readParam(discard=repeats)
```

Coordinates of annotated repeats can be obtained from several different sources. A curated blacklist of problematic regions is available from the ENCODE project [8], and can be obtained at https://sites.google.com/site/anshulkundaje/projects/blacklists. This list is constructed empirically from the ENCODE datasets and includes obvious offenders like telomeres, microsatellites and some rDNA genes. Alternatively, repeats can be predicted from the genome

sequence using software like *RepeatMasker*. These calls are available from the UCSC website (e.g., hgdownload.soe.ucsc.edu/goldenPath/mm10/bigZips/chromOut.tar.gz for mouse) or they can be extracted from an appropriate masked *BSgenome* object.

Using `discard` is more appropriate than simply ignoring windows that overlap the repeat regions. For example, a large window might contain both repeat and non-repeat regions. Discarding the window because of the former will compromise detection of DB features in the latter. Of course, any DB sites within the discarded regions will be lost from downstream analyses. Some caution is therefore required when specifying the regions of disinterest. For example, many more repeats are called by *RepeatMasker* than are present in the ENCODE blacklist, so the use of the former may result in loss of potentially interesting features.

2.2.4 Additional notes about parameter specification

Users can modify an existing *readParam* object using the `reform` method. The example below copies `param` and replaces `minq` and `discard` with new values. This is safer than directly modifying the slots, as appropriate type/value checking of each class member is performed.

```
another.param <- reform(param, minq=20, discard=repeats)
another.param
```

```
##     Extracting reads in single-end mode
##     Duplicate removal is turned off
##     Minimum allowed mapping score is 20
##     Reads are extracted from both strands
##     No restrictions are placed on read extraction
##     Reads in 1 region will be discarded
```

Users are encouraged to construct their own *readParam* objects and apply them consistently throughout their analyses. A good measure of synchronisation between `windowCounts` calls is to check that the values of `...$totals` are identical between calls. This suggests that the same reads are being extracted from the BAM files in each call.

2.2.5 Increasing speed and memory efficiency

The `spacing` parameter controls the distance between adjacent windows in the genome. By default, this is set to 50 bp, i.e., sliding windows are shifted 50 bp forward at each step. Using a higher value will reduce computational work as fewer features need to be counted. This may be useful when machine memory is limited. Of course, spatial resolution is lost with larger spacings. Adjacent positions are not counted and thus cannot be distinguished.

```
demo <- windowCounts(bam.files, spacing=100, ext=frag.len,
                     width=win.width, param=param)
head(rowRanges(demo))
```

```
## GRanges object with 6 ranges and 0 metadata columns:
##       seqnames          ranges strand
##          <Rle>       <IRanges>  <Rle>
##   [1]     chr1 3003701-3003710      *
##   [2]     chr1 3003801-3003810      *
```

```
##   [3]     chr1 3004001-3004010      *
##   [4]     chr1 3008301-3008310      *
##   [5]     chr1 3008401-3008410      *
##   [6]     chr1 3010801-3010810      *
##   -------
##   seqinfo: 66 sequences from an unspecified genome
```

For analyses with large windows, we suggest increasing the `spacing` to a fraction of the specified `width`. This reduces the computational work by decreasing the number of windows and extracted counts. Any loss in spatial resolution due to a larger spacing interval is negligible compared to that already lost by using a large window size. Conversely, `spacing` should not be larger than `ext/2` for analyses with small windows. This ensures that a narrow binding site will not be overlooked if it falls between two windows. If `ext` is also very small, `spacing` should be set to `width` to avoid loading too many small windows.

Windows that are overlapped by few fragments are filtered out based on the `filter` argument. A window is removed if the sum of counts across all libraries is below `filter`. This improves memory efficiency by discarding the majority of low-abundance windows corresponding to uninteresting background regions. The default value of the filter threshold is 10, though it can be raised to reduce memory usage for large libraries. More sophisticated filtering is recommended and should be applied later (see Chapter 3).

```
demo <- windowCounts(bam.files, ext=frag.len, width=win.width,
                     filter=30, param=param)
head(assay(demo))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    4   12   10    5
## [2,]    5    6    8   11
## [3,]    7    1    9   19
## [4,]    3    1    7   22
## [5,]    4    2   10   15
## [6,]    6    5   10   11
```

Users can parallelize read counting and several other functions by setting the `BPPARAM` argument. This will load and process reads from multiple BAM files simultaneously. The number of workers and type of parallelization can be specified using *BiocParallelParam* objects. By default, parallelization is turned off (i.e., set to a *SerialParam* object) because it provides little benefit for small files or on systems with I/O bottlenecks.

2.2.6 Assigning reads into bins

Setting `bin=TRUE` will direct `windowCounts` to count reads into contiguous bins across the genome. Here, `spacing` is set to `width` such that each window forms a bin. Only the 5′ end of each read is used for counting into bins, without any directional extension. (For paired-end data, the midpoint of the originating fragment is used – see below.)

```
demo <- windowCounts(bam.files, width=1000, bin=TRUE, param=param)
head(rowRanges(demo))
```

```
## GRanges object with 6 ranges and 0 metadata columns:
##       seqnames          ranges strand
```

```
##          <Rle>       <IRanges>  <Rle>
##   [1]     chr1 3000001-3001000      *
##   [2]     chr1 3001001-3002000      *
##   [3]     chr1 3002001-3003000      *
##   [4]     chr1 3003001-3004000      *
##   [5]     chr1 3004001-3005000      *
##   [6]     chr1 3005001-3006000      *
##   -------
##   seqinfo: 66 sequences from an unspecified genome
```

The `filter` argument is automatically set to 1, which means that counts will be returned for each non-empty genomic bin. Users should set `width` to a reasonably large value, to avoid running out of memory with a large number of small bins. We can also force `windowCounts` to return bins for *all* bins by setting `filter=0` manually.

2.3 Experiments involving paired-end data

ChIP experiments with paired-end sequencing are accomodated by setting `pe="both"` in the `param` object supplied to `windowCounts`. Read extension is not required as the genomic interval spanned by the originating fragment is explicitly defined as that between the 5′ positions of the paired reads. The number of fragments overlapping each window is then counted as previously described. By default, only proper pairs are used in which the two paired reads are on the same chromosome, face inward and are no more than `max.frag` apart.

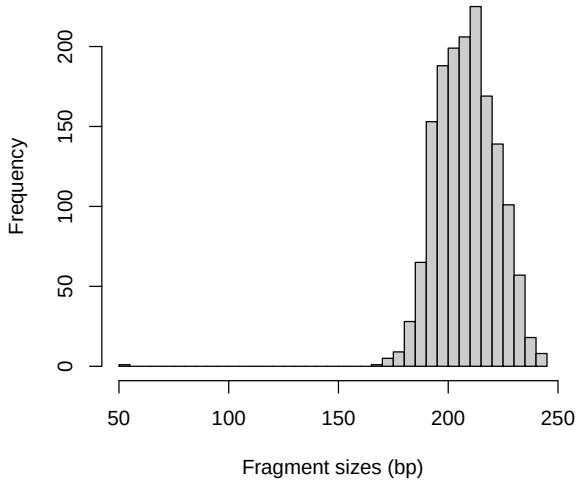
```
# Using the BAM file in Rsamtools as an example.
pe.bam <- system.file("extdata", "ex1.bam", package="Rsamtools",
    mustWork=TRUE)

pe.param <- readParam(max.frag=400, pe="both")
demo <- windowCounts(pe.bam, ext=250, param=pe.param)
demo$totals
```

```
## [1] 1572
```

A suitable value for `max.frag` is chosen by examining the distribution of fragment sizes from the `getPESizes` function. In this example, we might use a value of around 400 bp as it is larger than the vast majority of fragment sizes. The plot can also be used to examine the quality of the PE sequencing procedure. The location of the mode should be consistent with the fragmentation and size selection steps in library preparation.

```
out <- getPESizes(pe.bam)
frag.sizes <- out$sizes[out$sizes<=800]
hist(frag.sizes, breaks=50, xlab="Fragment sizes (bp)",
     ylab="Frequency", main="", col="grey80")
abline(v=400, col="red")
```



The number of fragments exceeding the maximum size is recorded for quality control. The `getPESizes` function also returns the number of single reads, pairs with one unmapped read, improperly orientated pairs and inter-chromosomal pairs. A non-negligble proportion of these reads may be indicative of problems with paired-end alignment or sequencing.

```
c(out$diagnostics, too.large=sum(out$sizes > 400))
```

```
##   total.reads  mapped.reads        single mate.unmapped    unoriented
##          3307          3307             0           163             0
##     inter.chr     too.large
##             0             0
```

Note that all of the paired-end methods in *csaw* depend on correct mate information for each alignment. This is usually enforced by the aligner in the output BAM file. Any file manipulations that might break the synchronisation should be corrected (e.g., with the *FixMateInformation* program from the *Picard* suite) prior to read counting.

Paired-end data can also be treated as single-end by specifiying `pe="first"` or `"second"` in the `readParam` constructor. This will only use the first or second read of each read pair, regardless of the validity of the pair or the relative quality of the alignments. This setting may be useful for contrasting paired- and single-end analyses, or in disastrous situations where paired-end sequencing has failed, e.g., due to ligation between DNA fragments.

```
first.param <- readParam(pe="first")
demo <- windowCounts(pe.bam, param=first.param)
demo$totals
```

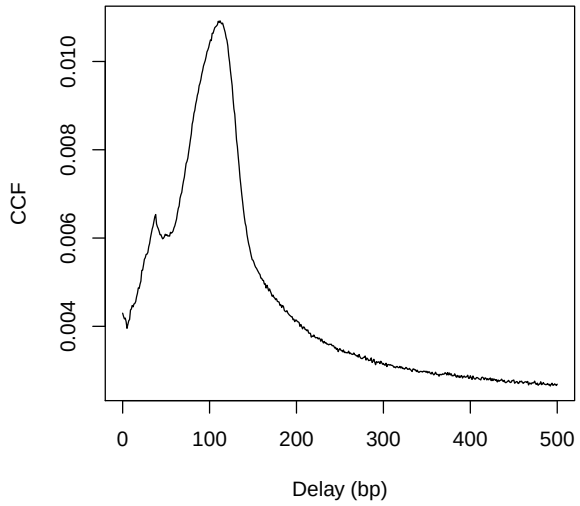
```
## [1] 1654
```

2.4 Estimating the average fragment length

2.4.1 Using cross-correlation plots

Cross-correlation plots are generated directly from BAM files using the `correlateReads` function. This provides a measure of the immunoprecipitation (IP) efficiency of a ChIP-seq experiment [9]. Efficient IP should yield a smooth peak at a delay distance corresponding to the average fragment length. This reflects the strand-dependent bimodality of reads around narrow regions of enrichment, e.g., TF binding sites.

```
max.delay <- 500
dedup.on <- reform(param, dedup=TRUE)
x <- correlateReads(bam.files, max.delay, param=dedup.on)
plot(0:max.delay, x, type="l", ylab="CCF", xlab="Delay (bp)")
```



The location of the peak is used as an estimate of the fragment length for read extension in `windowCounts`. An estimate of ~110 bp is obtained from the plot above. We can do this more precisely with the `maximizeCcf` function, which returns a similar value.

```
maximizeCcf(x)
```

```
## [1] 112
```

A sharp spike may also be observed in the plot at a distance corresponding to the read length. This is thought to be an artifact, caused by the preference of aligners towards uniquely mapped reads. Duplicate removal is typically required here (i.e., set `dedup=TRUE` in `readParam`) to reduce the size of this spike. Otherwise, the fragment length peak will not be visible as a separate entity. The size of the smooth peak can also be compared to the height of the spike to assess the signal-to-noise ratio of the data [10]. Poor IP efficiency will result in a smaller or absent peak as bimodality is less pronounced.

Cross-correlation plots can also be used for fragment length estimation of narrow histone marks such as histone acetylation and H3K4 methylation. However, they are less effective for regions of diffuse enrichment where bimodality is not obvious (e.g., H3K27 trimethylation).

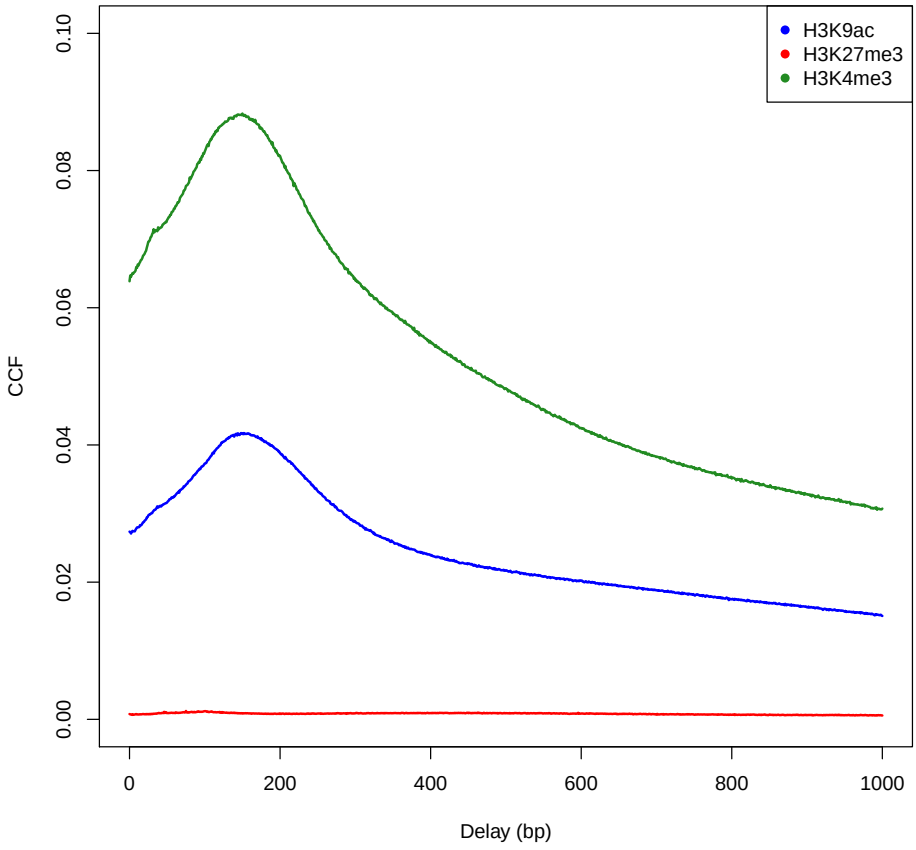
```
n <- 1000

# Using more data sets from 'chipseqDBData'.
acdata <- H3K9acData()
h3k9ac <- correlateReads(acdata$Path[1], n, param=dedup.on)

k27data <- H3K27me3Data()
h3k27me3 <- correlateReads(k27data$Path[1], n, param=dedup.on)

k4data <- H3K4me3Data()
h3k4me3 <- correlateReads(k4data$Path[1], n, param=dedup.on)

plot(0:n, h3k9ac, col="blue", ylim=c(0, 0.1), xlim=c(0, 1000),
     xlab="Delay (bp)", ylab="CCF", pch=16, type="l", lwd=2)
lines(0:n, h3k27me3, col="red", pch=16, lwd=2)
lines(0:n, h3k4me3, col="forestgreen", pch=16, lwd=2)
legend("topright", col=c("blue", "red", "forestgreen"),
       c("H3K9ac", "H3K27me3", "H3K4me3"), pch=16)
```



2.4.2 Variable fragment lengths between libraries

The `windowCounts` function also supports the use of library-specific fragment lengths. For example, libraries with less efficient fragmentation will have larger fragment lengths and wider peaks. Single-end reads in the peaks of such libraries will require more directional extension to impute a fragment interval that covers the binding site. However, some work is required to avoid detecting irrelevant DB from differences in peak widths. This is done by resizing the inferred fragments to the same length in all libraries. (Consider a bimodal peak, present in several libraries that have different fragment lengths. Resizing ensures that the subpeak on the forward strand is centered at the same location in each library. The same applies for the subpeak on the reverse strand.) Thus, the effect of differences in peak width between libraries can be largely mitigated.

Variable read extension is performed in `windowCounts` by setting `ext` to a list with two elements. The first element is a vector where each entry specifies the average fragment length to be used for the corresponding library. The second specifies the final length to which the inferred fragments are to be resized. If the second element is set to `NA`, no rescaling is performed and the library-specific fragment sizes are used directly. This also works for analyses with paired-end data, though the first element of `ext` will be ignored as directional extension is not performed. The example below rescales all fragments to 200 bp in all libraries. Extension information is stored in the *RangedSummarizedExperiment* object for later use.

```
multi.frag.lens <- list(c(100, 150, 200, 250), 200)
demo <- windowCounts(bam.files, ext=multi.frag.lens, filter=30, param=param)
demo$ext
```

```
## [1] 100 150 200 250
```

```
metadata(demo)$final
```

```
## [1] 200
```

In general, use of different extension lengths is unnecessary in well-controlled datasets. Difference in lengths between libraries are usually smaller than 50 bp. This is less than the inherent variability in fragment lengths within each library (see the histogram for the paired-end data in Section 2.3). The effect on the coverage profile of within-library variability in lengths will likely mask the effect of small between-library differences in the average lengths. Thus, an `ext` list should only be specified for datasets that exhibit large differences in the average fragment sizes between libraries.

2.5 Choosing an appropriate window size

We interpret the window size as the width of the binding "footprint" for the target protein, where the protein residues directly contact the DNA. TF analyses typically use a small window size, e.g., 10 - 20 bp, which maximizes spatial resolution for optimal detection of narrow regions of enrichment. For histone marks, widths of at least 150 bp are recommended [11]. This corresponds to the length of DNA wrapped up in each nucleosome, which is the smallest relevant unit for histone mark enrichment. We consider diffuse marks as chains of adjacent histones, for which the combined footprint may be very large (e.g., 1-10 kbp).

The choice of window size controls the compromise between spatial resolution and count size. Larger windows will yield larger read counts that can provide more power for DB detection. However, spatial resolution is also lost for large windows whereby adjacent features can no longer be distinguished. Reads from a DB site may be counted alongside reads from a non-DB site (e.g., non-specific background) or even those from an adjacent site that is DB in the opposite direction. This will result in the loss of DB detection power.

We might expect to be able to infer the optimal window size from the data, e.g., based on the width of the enriched regions. However, in practice, a clear-cut choice of distance/window size is rarely found in real datasets. For many non-TF targets, the widths of the enriched regions can be highly variable, suggesting that no single window size is optimal. Indeed, even if all enriched regions were of constant width, the width of the DB events occurring *within* those regions may be variable. This is especially true of diffuse marks where the compromise between resolution and power is more arbitrary.

We suggest performing an initial DB analysis with small windows to maintain spatial resolution. The widths of the final merged regions (see Section 6.2.1) can provide an indication of the appropriate window size. Alternatively, the analysis can be repeated with a series of larger windows, and the results combined (see Section 6.4.1). This examines a spread of resolutions for more comprehensive detection of DB regions.

2.6 Miscellaneous functions for non-standard counting

2.6.1 Counting over manually specified regions

The *csaw* package focuses on counting reads into windows. However, it may be occasionally desirable to use the same conventions (e.g., duplicate removal, quality score filtering) when counting reads into pre-specified regions. This can be performed with the `regionCounts` function, which is largely a wrapper for `countOverlaps` from the *GenomicRanges* package.

```
my.regions <- GRanges(c("chr11", "chr12", "chr15"),
                      IRanges(c(75461351, 95943801, 21656501),
                      c(75461610, 95944810, 21657610)))
reg.counts <- regionCounts(bam.files, my.regions, ext=frag.len, param=param)
head(assay(reg.counts))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   43   68  116  111
## [2,]    1    0    0    0
## [3,]   17   10   14   12
```

2.6.2 Strand-specific counting

Techniques like CLIP-seq, MeDIP-seq or CAGE provide strand-specific sequence information. The *csaw* package can analyze these datasets through strand-specific counting. This can be done manually setting the `forward` slot in the *readParam* object to `TRUE` or `FALSE`, to count only forward- or reverse-strand reads respectively in `windowCounts` or `regionCounts`. Alternatively, the `strandedCounts` wrapper function can be used to obtain strand-specific

counts for each window or region. The strand of each output range indicates the strand on which reads were counted for that row. Up to two rows can be generated for each window or region, depending on filtering.

```
ss.param <- reform(param, forward=NULL)
ss.counts <- strandedCounts(bam.files, ext=frag.len, width=win.width,
                            param=ss.param)
strand(rowRanges(ss.counts))
```

```
## factor-Rle of length 924432 with 72 runs
##   Lengths: 67133 64898 22552 22019 38909 ...  1819     6     3    70    82
##   Values :     +     -     +     -     + ...     -     +     -     +     -
## Levels(3): + - *
```

Note that `strandedCounts` operates internally by calling `windowCounts` (or `regionCounts`) twice with different settings for `param$forward`. Any value for `forward` in the input `param` object will be ignored. In fact, the function will *only* accept a `NULL` value for this slot. This is intended to protect the user, as any attempt to re-use the `ss.param` object in functions that are not designed for strand specificity will (appropriately) raise an error.

Chapter 3

Filtering out uninteresting windows

This chapter will require `bam.files`, `frag.len`, `param` and `data` from the last chapter. We'll also need the `aveLogCPM` function from *csaw*, so load the package if you haven't done so already.

3.1 Independent filtering for count data

Many of the low abundance windows in the genome correspond to background regions in which DB is not expected. Indeed, windows with low counts will not provide enough evidence against the null hypothesis to obtain sufficiently low p-values for DB detection. Similarly, some approximations used in the statistical analysis will fail at low counts. Removing such uninteresting or ineffective tests reduces the severity of the multiple testing correction, increases detection power amongst the remaining tests and reduces computational work.

Filtering is valid so long as it is independent of the test statistic under the null hypothesis [12]. In the negative binomial (NB) framework, this (probably) corresponds to filtering on the overall NB mean. The DB p-values retained after filtering on the overall mean should be uniform under the null hypothesis, by analogy to the normal case. Row sums can also be used for datasets where the effective library sizes are not very different, or where the counts are assumed to be Poisson-distributed between biological replicates.

In *edgeR*, the log-transformed overall NB mean is referred to as the average abundance. This is computed with the `aveLogCPM` function, as shown below for each window.

```
abundances <- aveLogCPM(asDGEList(data))
summary(abundances)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -2.327  -2.303  -2.214  -2.143  -2.079   7.931
```

For demonstration purposes, an arbitrary threshold of -1 is used here to filter the window abundances. This restricts the analysis to windows with abundances above this threshold.

```
keep.simple <- abundances > -1
filtered.data <- data[keep.simple,]
summary(keep.simple)
```

```
##    Mode   FALSE    TRUE
## logical 4688607   15708
```

The exact choice of filter threshold may not be obvious. In particular, there is often no clear distinction in abundances between genuine binding and background events, e.g., due to the presence of many weak but genuine binding sites. A threshold that is too small will be ineffective, whereas a threshold that is too large may decrease power by removing true DB sites. Arbitrariness is unavoidable when balancing these opposing considerations.

Nonetheless, several strategies for defining the threshold are described below. Users should start by choosing **one** of these filtering approaches to implement in their analyses. Each approach yields a logical vector that can be used in the same way as `keep.simple`.

3.2 By count size

The simplest approach is to simply filter according to the count size. This removes windows for which the counts are simply too low for modelling and hypothesis testing. The code below retains windows with (library size-adjusted) average counts greater than 5.

```
keep <- abundances > aveLogCPM(5, lib.size=mean(data$totals))
summary(keep)
```

```
##    Mode   FALSE    TRUE
## logical 4599548  104767
```

However, a count-based filter becomes less effective as the library size increases. More windows will be retained with greater sequencing depth, even in uninteresting background regions. This increases both computational work and the severity of the multiplicity correction. The threshold may also be inappropriate when library sizes are very different.

3.3 By proportion

One approach is to to assume that only a certain proportion - say, 0.1% - of the genome is genuinely bound. This corresponds to the top proportion of high-abundance windows. The total number of windows is calculated from the genome length and the `spacing` interval used in `windowCounts`. The `filterWindowsProportion` function returns the ratio of the rank of each window to this total, where higher-abundance windows have larger ranks. Users can then retain those windows with rank ratios above the unbound proportion of the genome.

```
keep <- filterWindowsProportion(data)$filter > 0.999
sum(keep)
```

```
## [1] 54616
```

This approach is simple and has the practical advantage of maintaining a constant number of windows for the downstream analysis. However, it may not adapt well to different datasets where the proportion of bound sites can vary. Using an inappropriate percentage of binding sites will result in the loss of potential DB regions or inclusion of background regions.

3.4 By global enrichment

An alternative approach involves choosing a filter threshold based on the fold change over the level of non-specific enrichment. The degree of background enrichment is estimated by counting reads into large bins across the genome. Binning is necessary here to increase the size of the counts when examining low-density background regions. This ensures that precision is maintained when estimating the background abundance.

```
bin.size <- 2000L
binned <- windowCounts(bam.files, bin=TRUE, width=bin.size, param=param)
```

The median of the average abundances across all bins is computed and used as a global estimate of the background coverage. This global background is then compared to the window-based abundances. This determines whether a window is driven by background enrichment, and thus, unlikely to be interesting. However, some care is required as the sizes of the regions used for read counting are different between bins and windows. The average abundance of each bin must be scaled down to be comparable to those of the windows.

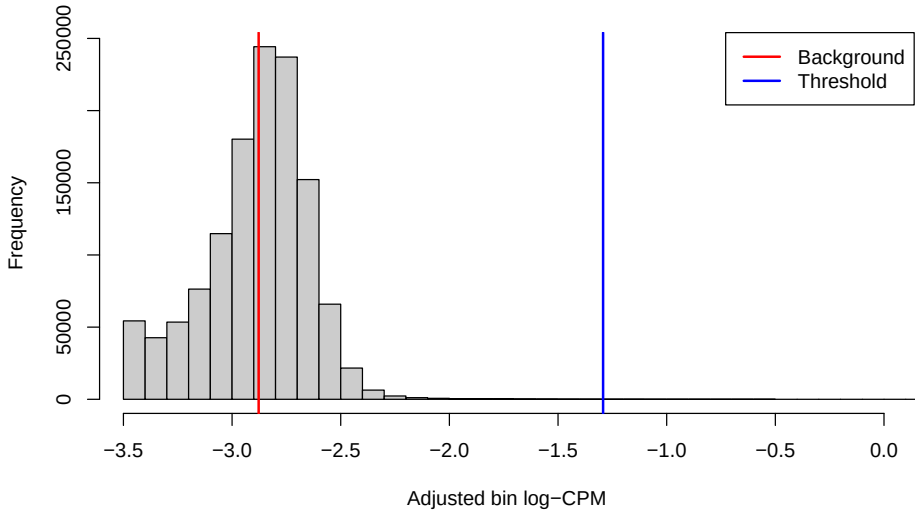
The `filterWindowsGlobal` function returns the increase in the abundance of each window over the global background. Windows are filtered by setting some minimum threshold on this increase. The aim is to eliminate the majority of uninteresting windows prior to further analysis. Here, a fold change of 3 is necessary for a window to be considered as containing a binding site. This approach has an intuitive and experimentally relevant interpretation that adapts to the level of non-specific enrichment in the dataset.

```
filter.stat <- filterWindowsGlobal(data, background=binned)
keep <- filter.stat$filter > log2(3)
sum(keep)
```

```
## [1] 23949
```

The effect of filtering can also be visualized with a histogram. This allows users to confirm that the bulk of (assumed) background bins are discarded upon filtering. Note that bins containing genuine binding sites will usually not be visible on such plots. This is due to the dominance of the background-containing bins throughout the genome.

```
hist(filter.stat$back.abundances, xlab="Adjusted bin log-CPM", breaks=100,
     main="", col="grey80", xlim=c(min(filter.stat$back.abundances), 0))
global.bg <- filter.stat$abundances - filter.stat$filter
abline(v=global.bg[1], col="red", lwd=2)
abline(v=global.bg[1]+log2(3), col="blue", lwd=2)
legend("topright", lwd=2, col=c('red', 'blue'),
       legend=c("Background", "Threshold"))
```



Of course, the pre-specified minimum fold change may be too aggressive when binding is weak. For TF data, a large cut-off works well as narrow binding sites will have high read densities and are unlikely to be lost during filtering. Smaller minimum fold changes are recommended for diffuse marks where the difference from background is less obvious.

3.5 By local enrichment

3.5.1 Mimicking single-sample peak callers

Local background estimators can also be constructed, which avoids inappropriate filtering when there are differences in background coverage across the genome. Here, the 2 kbp region surrounding each window will be used as the "neighborhood" over which a local estimate of non-specific enrichment for that window can be obtained. The counts for these regions are first obtained with the `regionCounts` function. This should be synchronized with `windowCounts` by using the same `param`, if any non-default settings were used.

```
surrounds <- 2000
neighbor <- suppressWarnings(resize(rowRanges(data), surrounds, fix="center"))
wider <- regionCounts(bam.files, regions=neighbor, ext=frag.len, param=param)
```

We apply `filterWindowsLocal` to compute enrichment values, i.e., the increase in the abundance of each window over its neighborhood. In this function, counts for each window are subtracted from the counts for its neighborhood. This ensures that any enriched regions or binding sites inside the window will not interfere with estimation of its local background. The width of the window is also subtracted from that of its neighborhood, to reflect the effective size of the latter after subtraction of counts. Based on the fold-differences in widths, the abundance of the neighborhood is scaled down for a valid comparison to that of the corresponding window. Enrichment values are subsequently calculated from the differences in scaled abundances.

```
filter.stat <- filterWindowsLocal(data, wider)
summary(filter.stat$filter)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -6.2866  0.4377  0.5749  0.5882  0.7273  8.1494
```

Filtering can then be performed using a quantile- or fold change-based threshold on the enrichment values. In this scenario, a 3-fold increase in enrichment over the neighborhood abundance is required for retention of each window. This roughly mimics the behavior of single-sample peak-calling programs such as *MACS* [13].

```
keep <- filter.stat$filter > log2(3)
sum(keep)
```

```
## [1] 10701
```

Note that this procedure also assumes that no other enriched regions are present in each neighborhood. Otherwise, the local background will be overestimated and windows may be incorrectly filtered out. This may be problematic for diffuse histone marks or TFBS clusters where enrichment may be observed in both the window and its neighborhood.

If this seems too complicated, an alternative is to identify locally enriched regions using peak-callers like *MACS*. Filtering can then be performed to retain only windows within called peaks. However, peak calling must be done independently of the DB status of each window. If libraries are of similar size or biological variability is low, reads can be pooled into one library for single-sample peak calling [4]. This is equivalent to filtering on the average count and avoids loss of the type I error control from data snooping.

3.5.2 Identifying local maxima

Another strategy is to use the `findMaxima` function to identify local maxima in the read density across the genome. The code below will determine if each window is a local maximum, i.e., whether it has the highest average abundance within 1 kbp on either side. The data can then be filtered to retain only these locally maximal windows. This can also be combined with other filters to ensure that the retained windows have high absolute abundance.

```
maxed <- findMaxima(rowRanges(data), range=1000, metric=abundances)
summary(maxed)
```

```
##    Mode   FALSE    TRUE
## logical 3782693  921622
```

This approach is very aggressive and should only be used (sparingly) in datasets where binding is sharp, simple and isolated. Complex binding events involving diffuse enrichment or adjacent binding sites will not be handled well. For example, DB detection will fail if a low-abundance DB window is ignored in favor of a high-abundance non-DB neighbor.

3.5.3 With negative controls

Negative controls for ChIP-seq refer to input or IgG libraries where the IP step has been skipped or compromised with an irrelevant antibody, respectively. This accounts for sequencing/mapping biases in ChIP-seq data. IgG controls also quantify the amount of non-specific enrichment throughout the genome. These controls are mostly irrelevant when testing for DB

between ChIP samples. However, they can be used to filter out windows where the average abundance across the ChIP samples is below the abundance of the control. To illustrate, let us add an input library to our NF-YA data set in the code below.

```
tf.data <- NFYAData()
with.input <- tf.data$Path
in.demo <- windowCounts(with.input, ext=frag.len, param=param)
chip <- in.demo[,1:4] # All ChIP libraries
control <- in.demo[,5] # All control libraries
```

Some additional work is required to account for composition biases that are likely to be present when comparing ChIP to negative control samples (see Section 4.2). A simple strategy for normalization involves counting reads into large bins, which are used in `scaleControlFilter` to compute a normalization factor.

```
in.binned <- windowCounts(with.input, bin=TRUE, width=10000, param=param)
chip.binned <- in.binned[,1:4]
control.binned <- in.binned[,5]
scale.info <- scaleControlFilter(chip.binned, control.binned)
```

We use the `filterWindowsControl` function to compute the enrichment of the ChIP counts over the control counts for each window. This uses `scale.info` to adjust for composition biases between ChIP and control samples. A larger `prior.count` of 5 is also used to compute the average abundance. This protects against inflated log-fold changes when the count for the window in the control sample is near zero. By comparison, the global and local background estimates require less protection (`prior.count=2`, by default) as they are derived from larger bins with more counts.

```
filter.stat <- filterWindowsControl(chip, control,
    prior.count=5, scale.info=scale.info)
```

The log-fold enrichment of the ChIP sample over the control is then computed for each window, after normalizing for composition bias with the binned counts. The example below requires a 3-fold or greater increase in abundance over the control to retain each window.

```
keep <- filter.stat$filter > log2(3)
sum(keep)
```

```
## [1] 6657
```

As an aside, the *csaw* pipeline can also be applied to search for "DB" between ChIP libraries and control libraries. The ChIP and control libraries can be treated as separate groups, in which most "DB" events are expected to be enriched in the ChIP samples. If this is the case, the filtering procedure described above is inappropriate as it will select for windows with differences between ChIP and control samples. This compromises the assumption of the null hypothesis during testing, resulting in loss of type I error control.

3.6 By prior information

When only a subset of genomic regions are of interest, DB detection power can be improved by removing windows lying outside of these regions. Such regions could include promoters, enhancers, gene bodies or exons. Alternatively, sites could be defined from a previous ex-

periment or based on the genome sequence, e.g., TF motif matches. The example below retrieves the coordinates of the broad gene bodies from the mouse genome, including the 3 kbp region upstream of the TSS that represents the putative promoter region for each gene.

```
library(TxDb.Mmusculus.UCSC.mm10.knownGene)
broads <- genes(TxDb.Mmusculus.UCSC.mm10.knownGene)
broads <- resize(broads, width(broads)+3000, fix="end")
head(broads)
```

```
## GRanges object with 6 ranges and 1 metadata column:
##             seqnames              ranges strand |     gene_id
##                <Rle>           <IRanges>  <Rle> | <character>
##   100009600     chr9   21062393-21076096      - |   100009600
##   100009609     chr7   84935565-84967115      - |   100009609
##   100009614    chr10   77708457-77712009      + |   100009614
##   100009664    chr11   45805087-45841171      + |   100009664
##      100012     chr4 144157557-144165663      - |      100012
##      100017     chr4 134741554-134771024      - |      100017
##   -------
##   seqinfo: 66 sequences (1 circular) from mm10 genome
```

Windows can be filtered to only retain those which overlap with the regions of interest. Discerning users may wish to distinguish between full and partial overlaps, though this should not be a significant issue for small windows. This could also be combined with abundance filtering to retain windows that contain putative binding sites in the regions of interest.

```
suppressWarnings(keep <- overlapsAny(rowRanges(data), broads))
sum(keep)
```

```
## [1] 2700568
```

Any information used here should be independent of the DB status under the null in the current dataset. For example, DB calls from a separate dataset and/or independent annotation can be used without problems. However, using DB calls from the same dataset to filter regions would violate the null assumption and compromise type I error control.

In addition, this filter is unlike the others in that it does not operate on the abundance of the windows. It is possible that the set of retained windows may be very small, e.g., if no non-empty windows overlap the pre-defined regions of interest. Thus, it may be better to apply this filter before the multiplicity correction but after DB testing. This ensures that there are sufficient windows for stable estimation of the downstream statistics.

3.7 Some final comments about filtering

It should be stressed that these filtering strategies do not eliminate subjectivity. Some thought is still required in selecting an appropriate proportion of bound sites or minimum fold change above background for each method. Rather, these filters provide a relevant interpretation for what would otherwise be an arbitrary threshold on the abundance.

As a general rule, users should filter less aggressively if there is any uncertainty about the features of interest. In particular, the thresholds shown in this chapter for each filtering statistic are fairly mild. This ensures that more potentially DB windows are retained for

testing. Use of an aggressive filter risks the complete loss of detection for such windows, even if power is improved among those that are retained. Low numbers of retained windows may also lead to unstable estimates during, e.g., normalization, variance modelling.

Different filters can also be combined in more advanced applications, e.g., by running `data[keep1 & keep2,]` for filter vectors `keep1` and `keep2`. Any benefit will depend on the type of filters involved. The greatest effect is observed for filters that operate on different principles. For example, the low-count filter can be combined with others to ensure that all retained windows surpass some minimum count. This is especially relevant for the local background filters, where a large enrichment value does not guarantee a large count.

Chapter 4

Calculating normalization factors

This chapter will need to use the `bam.files` vector again (from the introduction). We'll also need the `param` object that we defined in Chapter 2, as well as the `filtered.data` object that we constructed in Chapter 3. You'll notice that character vectors containing paths to other BAM files are defined throughout this chapter. However, these are only present for demonstration purposes and aren't necessary for the main NF-YA example.

4.1 Overview

The complexity of the ChIP-seq technique gives rise to a number of different biases in the data. For a DB analysis, library-specific biases are of particular interest as they can introduce spurious differences between conditions. This includes composition biases, efficiency biases and trended biases. Thus, normalization between libraries is required to remove these biases prior to any statistical analysis. Several normalization strategies are presented here, though users should only pick **one** to use for any given analysis. Advice on choosing the most appropriate method is scattered throughout the chapter, so read carefully.

4.2 Eliminating composition biases

4.2.1 Using the TMM method on binned counts

As the name suggests, composition biases are formed when there are differences in the composition of sequences across libraries. Highly enriched regions consume more sequencing resources and thereby suppress the representation of other regions. Differences in the magnitude of suppression between libraries can lead to spurious DB calls. Scaling by library size fails to correct for this as composition biases can still occur in libraries of the same size.

To remove composition biases in *csaw*, reads are counted into large bins and the counts are used for normalization with the `normFactors` wrapper function. This uses the trimmed mean of M-values (TMM) method [14] to correct for any systematic fold change in the coverage of the bins. The assumption here is that most bins represent non-DB background regions, so any consistent difference across bins must be technical bias.

```
binned <- windowCounts(bam.files, bin=TRUE, width=10000, param=param)
filtered.data <- normFactors(binned, se.out=filtered.data)
filtered.data$norm.factors
```

```
## [1] 1.0084321 0.9751153 1.0136252 1.0032750
```

The TMM method trims away putative DB bins (i.e., those with extreme M-values) and computes normalization factors from the remainder to use in *edgeR*. The size of each library is scaled by the corresponding factor to obtain an effective library size for modelling. A larger normalization factor results in a larger effective library size and is conceptually equivalent to scaling each individual count downwards, given that the ratio of that count to the (effective) library size will be smaller. Check out the *edgeR* user's guide for more information.

In the above code, the `normFactors` call computes normalization factors from the bin-level counts in `binned` (see Section 4.2.2). The `se.out` argument directs the function to return a modified version of `filtered.data`, where the normalization factors are stored alongside the *window*-level counts for further analysis. Composition biases affect both bin- and window-level counts, so computing normalization factors from the former and applying them to the latter is valid – provided that the library sizes are the same between the two sets of counts, as the factors are interpreted with respect to the library sizes. (In *csaw*, separate calls to `windowCounts` with the same *readParam* object will always yield the same library sizes in `totals`.)

Note that `normFactors` skips the precision weighting step in the TMM method. Weighting aims to increase the contribution of bins with high counts, as these yield more precise M-values. However, high-abundance bins are more likely to contain binding sites and thus are more likely to be DB compared to background regions. If any DB regions should survive trimming, upweighting them would be counterproductive.

4.2.2 Motivating the use of large bins instead of windows

By definition, read coverage is low for background regions of the genome. This can result in a large number of zero counts and undefined M-values when reads are counted into small windows. Adding a prior count is only a superficial solution as the chosen prior will have undue influence on the estimate of the normalization factor when many counts are low. The variance of the fold change distribution is also higher for low counts, which reduces the effectiveness of the trimming procedure. These problems can be overcome by using large bins to increase the size of the counts, thus improving the precision of TMM normalization. The normalization factors computed from the bin-level counts are then applied to the window-level counts of interest.

Of course, this strategy requires the user to supply a bin size. If the bins are too large, background and enriched regions will be included in the same bin. This makes it difficult to trim away bins corresponding to enriched regions. On the other hand, the counts will be too low if the bins are too small. Testing multiple bin sizes is recommended to ensure that the estimates are robust to any changes. A value of 10 kbp is usually suitable for most datasets.

```
demo <- windowCounts(bam.files, bin=TRUE, width=5000, param=param)
normFactors(demo, se.out=FALSE)
```

```
## [1] 1.0083353 0.9778719 1.0109448 1.0031956
```

```
demo <- windowCounts(bam.files, bin=TRUE, width=15000, param=param)
```

```
normFactors(demo, se.out=FALSE)
```

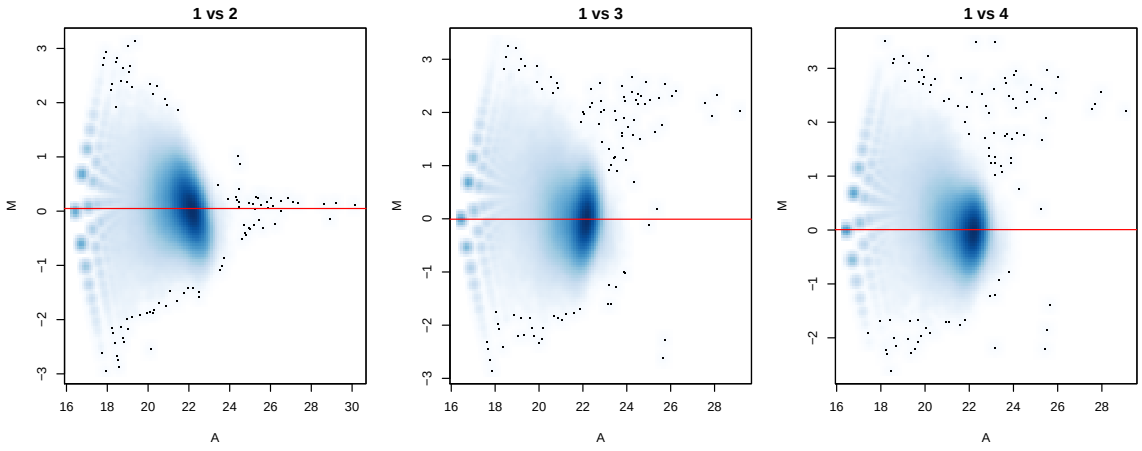
```
## [1] 1.0085125 0.9723214 1.0160058 1.0037202
```

We use `se.out=FALSE` to instruct `normFactors` to return the normalization factors directly. This is more convenient than returning a *RangedSummarizedExperiment* object and extracting the normalization factors with `...$norm.factors`. Here, the factors are consistently close to unity, which suggests that composition bias is negligble in this dataset. See Section 4.3.3 for some examples with greater bias.

4.2.3 Visualizing normalization outcomes with MA plots

The effectiveness of normalization can be examined using a MA plot. A single main cloud of points should be present, consisting primarily of background regions. Separation into multiple discrete points indicates that the counts are too low and that larger bin sizes should be used. Composition biases manifest as a vertical shift in the position of this cloud. Ideally, the log-ratios of the corresponding normalization factors should pass through the centre of the cloud. This indicates that undersampling has been identified and corrected.

```
par(mfrow=c(1, 3), mar=c(5, 4, 2, 1.5))
adj.counts <- cpm(asDGEList(binned), log=TRUE)
normfacs <- filtered.data$norm.factors
for (i in seq_len(length(bam.files)-1)) {
    cur.x <- adj.counts[,1]
    cur.y <- adj.counts[,1+i]
    smoothScatter(x=(cur.x+cur.y)/2+6*log2(10), y=cur.x-cur.y,
                  xlab="A", ylab="M", main=paste("1 vs", i+1))
    all.dist <- diff(log2(normfacs[c(i+1, 1)]))
    abline(h=all.dist, col="red")
}
```



4.3 Eliminating efficiency biases

4.3.1 Using the TMM method on high-abundance regions

Efficiency biases in ChIP-seq data refer to fold changes in enrichment that are introduced by variability in IP efficiencies between libraries. These technical differences are not biologically interesting and must be removed. This can be achieved by assuming that high-abundance windows contain binding sites. Consider the following H3K4me3 data set, where reads are counted into 150 bp windows.

```
k4data <- H3K4me3Data()
k4data
```

```
## DataFrame with 4 rows and 3 columns
##                   Name            Description      Path
##            <character>            <character>    <List>
## 1    h3k4me3-proB-8110   pro-B H3K4me3 (8110) <BamFile>
## 2    h3k4me3-proB-8115   pro-B H3K4me3 (8115) <BamFile>
## 3 h3k4me3-matureB-8070 mature B H3K4me3 (80.. <BamFile>
## 4 h3k4me3-matureB-8088 mature B H3K4me3 (80.. <BamFile>
```

```
me.files <- k4data$Path[c(1,3)]
me.demo <- windowCounts(me.files, width=150, param=param)
```

High-abundance windows are chosen using a global filtering approach described in Section 3.4. Here, the binned counts in `me.bin` are only used for defining the background abundance, *not* for computing normalization factors.

```
me.bin <- windowCounts(me.files, bin=TRUE, width=10000, param=param)
keep <- filterWindowsGlobal(me.demo, me.bin)$filter > log2(3)
filtered.me <- me.demo[keep,]
```

The TMM method is then applied to eliminate systematic differences across those windows. This also assumes that most binding sites in the genome are not DB. Thus, any systematic differences in coverage among the high-abundance windows must be caused by differences in IP efficiency between libraries or some other technical issue. Scaling by the normalization factors will subseqeuently remove these bises between libraries.

```
filtered.me <- normFactors(filtered.me, se.out=TRUE)
me.eff <- filtered.me$norm.factors
me.eff
```

```
## [1] 1.2654867 0.7902098
```

The above process seems rather involved, but this is only because we need to work our way through counting and normalization for a new data set. Only `normFactors` is actually performing the normalization step. We set `se.out=TRUE` so that the normalization factors are stored alongside the window counts for use in the downstream *edgeR* analysis. As a demonstration, we repeat this procedure on another data set involving H3 acetylation.

```
acdata <- H3K9acData()
acdata
```

```
## DataFrame with 4 rows and 3 columns
##                   Name            Description      Path
##            <character>            <character>    <List>
## 1    h3k9ac-proB-8113    pro-B H3K9ac (8113) <BamFile>
## 2    h3k9ac-proB-8108    pro-B H3K9ac (8108) <BamFile>
## 3 h3k9ac-matureB-8059 mature B H3K9ac (8059) <BamFile>
## 4 h3k9ac-matureB-8086 mature B H3K9ac (8086) <BamFile>
```

```
ac.files <- acdata$Path[c(1,2)]
ac.demo <- windowCounts(ac.files, width=150, param=param)
ac.bin <- windowCounts(ac.files, bin=TRUE, width=10000, param=param)
keep <- filterWindowsGlobal(ac.demo, ac.bin)$filter > log2(5)
summary(keep)
```

```
##    Mode   FALSE    TRUE
## logical  376632  438632
```

```
filtered.ac <- ac.demo[keep,]
filtered.ac <- normFactors(filtered.ac, se.out=TRUE)
ac.eff <- filtered.ac$norm.factors
ac.eff
```

```
## [1] 1.0745592 0.9306142
```

Normalization for efficiency biases assumes that most binding sites are not DB. Genuine biological differences may be removed when the assumption of a non-DB majority does not hold, e.g., overall binding is truly lower in one condition. In such cases, it is safer to normalize for composition biases – see Section 4.4 for a discussion of the choice between normalization methods.

4.3.2 Filtering windows prior to normalization

Normalization for efficiency biases is performed on window-level counts instead of bin counts. This is possible as filtering ensures that we only retain the high-abundance windows, i.e., those with counts that are large enough for stable calculation of normalization factors. It is not necessary to use larger windows or bins, and indeed, direct use of the windows of interest ensures removal of systematic differences in those windows prior to downstream analyses.

The filtering procedure needs to be stringent enough to avoid retaining windows from background regions. These will interfere with calculation of normalization factors from binding sites. This is due to the lower coverage for background regions, as well as the fact that they are not affected by efficiency bias (and cannot contribute to its estimation). Conversely, attempting to use the factors computed from high-abundance windows on windows from background regions will result in incorrect normalization of the latter. Thus, it is usually better to err on the side of caution and filter aggressively to ensure that background regions are not retained in downstream analyses. Obviously, though, retaining too few windows will result in unstable estimates of the normalization factors.

4.3.3 Checking normalization with MA plots

We again visualize the effect of normalization with MA plots. Plots are constructed using counts for 10 kbp bins, rather than with those from the windows. This is useful as the behavior of the entire genome can be examined, rather than just that of the high-abundance windows. It also allows calculation of and comparison to the factors for composition bias.

```
me.comp <- normFactors(me.bin, se.out=FALSE)
me.comp
```

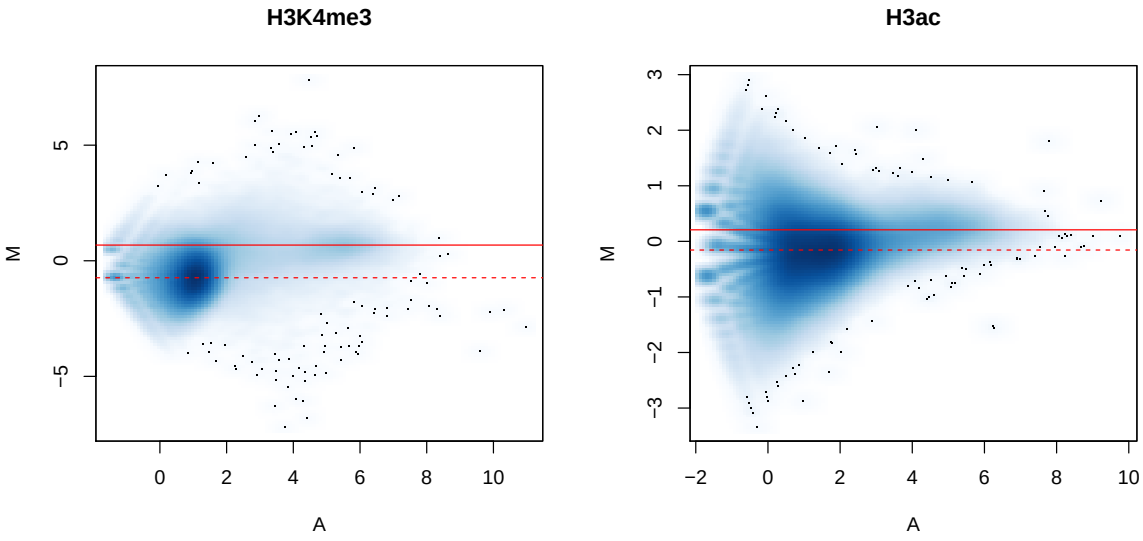
```
## [1] 0.7750992 1.2901574
```

```
ac.comp <- normFactors(ac.bin, se.out=FALSE)
ac.comp
```

```
## [1] 0.9473693 1.0555545
```

The clouds at low and high A-values represent the background and bound regions, respectively. The normalization factors from removal of composition bias (dashed) pass through the former, whereas the factors to remove efficiency bias (full) pass through the latter. A non-zero M-value location for the high A-value cloud represents a systematic difference between libraries for the bound regions, either due to genuine DB or variable IP efficiency. This also induces composition bias, leading to a non-zero M-value for the background cloud.

```
par(mfrow=c(1,2))
for (main in c("H3K4me3", "H3ac")) {
    if (main=="H3K4me3") {
        bins <- me.bin
        comp <- me.comp
        eff <- me.eff
    } else {
        bins <- ac.bin
        comp <- ac.comp
        eff <- ac.eff
    }
    adjc <- cpm(asDGEList(bins), log=TRUE)
    smoothScatter(x=rowMeans(adjc), y=adjc[,1]-adjc[,2],
                  xlab="A", ylab="M", main=main)
    abline(h=log2(eff[1]/eff[2]), col="red")
    abline(h=log2(comp[1]/comp[2]), col="red", lty=2)
}
```



4.4 Choosing between normalization strategies

The normalization strategies for composition and efficiency biases are mutually exclusive, as only one set of normalization factors will ultimately be used in *edgeR*. The choice between the two methods depends on whether one assumes that the systematic differences at high abundances represent genuine DB events. If so, the binned TMM method from Section 4.2 should be used to remove composition bias. This will preserve the assumed DB, at the cost of ignoring any efficiency biases that might be present. Otherwise, if the systematic differences are not genuine DB, they must represent efficiency bias and should be removed by applying the TMM method on high-abundance windows (Section 4.3). Some understanding of the biological context is useful in making this decision, e.g., comparing a wild-type against a knock-out for the target protein should result in systematic DB, while overall levels of histone marking are expected to be consistent in most conditions.

For the main NF-YA example, there is no expectation of constant binding between cell types. Thus, normalization factors will be computed to remove composition biases. This ensures that any genuine systematic changes in binding will still be picked up. In general, normalization for composition bias is a good starting point for any analysis. This can be considered as the "default" strategy unless there is evidence for a confounding efficiency bias.

4.5 Scaling normalization with spike-in chromatin

Several studies have used spike-in chromatin for scaling normalization of ChIP-seq data [15, 16]. Briefly, a constant amount of chromatin from a different species is added to each sample at the start of the ChIP-seq protocol. The mixture is processed and sequenced in the usual manner, using an antibody that can bind epitopes of interest from both species. The coverage of the spiked-in foreign chromatin is then quantified in each library. As the quantity of foreign chromatin should be constant in each sample, the coverage of binding sites on the foreign genome should also be the same between libraries. Any difference in coverage between libraries represents some technical bias that should be removed by scaling.

This normalization strategy can be implemented in *csaw* with some work. Assuming that the reference genome includes appropriate sequences from the foreign genome, coverage is quantified across genomic windows with `windowCounts`. Filtering is performed to select for high-abundance windows in the foreign genome (see Section 4.3), yielding counts for all enriched spike-in regions. (The filtered object is named `spike.data` in the code below.) Normalization factors are computed by applying the TMM method on these counts via `normFactors`. This aims to identify the fold-change in coverage between samples that is attributable to technical effects.

```
# Pretend chr1 is a spike-in, for demonstration purposes only!
is.1 <- seqnames(rowRanges(data))=="chr1"
spike.data <- data[is.1,]
endog.data <- data[!is.1,]

endog.data <- normFactors(spike.data, se.out=endog.data)
```

In the code above, the spike-in normalization factors are returned in a modified copy of `endog.data` for further analysis of the endogenous windows. We assume that the library sizes in `totals` are the same between `spike.data` and `endog.data`, which should be the case if they were formed by subsetting the output of a single `windowCounts` call. This ensures that the normalization factors computed from the spike-in windows are applicable to the endogenous windows.

Compared to the previous normalization methods, the spike-in approach does not distinguish between composition and efficiency biases. Instead, it uses the fold-differences in the coverage of spiked-in binding sites to empirically measure and remove the net bias between libraries. This avoids the need for assumptions regarding the origin of any systematic differences between libraries. That said, spike-in normalization involves some strong assumptions of its own. In particular, the ratio of spike-in chromatin to endogenous chromatin is assumed to be the same in each sample. This requires accurate quantitation of the chromatin in each sample, followed by precise addition of small spike-in quantities. Furthermore, the spike-in chromatin is assumed to behave in the same manner as endogenous chromatin throughout the ChIP-seq protocol. Whether these assumptions are reasonable will depend on the experimenter and the nature of the spike-in chromatin.

4.6 Dealing with trended biases

4.6.1 Applying loess-based normalization to the counts

In more extreme cases, the bias may vary with the average abundance to form a trend. One possible explanation is that changes in IP efficiency will have little effect at low-abundance background regions and more effect at high-abundance binding sites. Thus, the magnitude of the bias between libraries will change with abundance. The trend cannot be corrected with scaling methods as no single scaling factor will remove differences at all abundances. Rather, non-linear methods are required, such as cyclic loess or quantile normalization.

One such implementation of a non-linear normalization method is provided in `normOffsets`. This is based on the fast loess algorithm [17] with minor adaptations to handle low counts. A matrix is produced that contains an offset term for each bin/window in each library. This offset matrix can then be directly used in *edgeR*, assuming that the bins or windows used in

normalization are also the ones to be tested for DB. We demonstrate this procedure below, using filtered counts for 2 kbp windows in the H3 acetylation data set. (This window size is chosen purely for aesthetics in this demonstration, as the trend is less obvious at smaller widths. Obviously, users should pick a more appropriate value for their analysis.)

```
ac.demo2 <- windowCounts(ac.files, width=2000L, param=param)
filtered <- filterWindowsGlobal(ac.demo2, ac.bin)
keep <- filtered$filter > log2(4)
ac.demo2 <- ac.demo2[keep,]
ac.demo2 <- normOffsets(ac.demo2, se.out=TRUE)
ac.off <- assay(ac.demo2, "offset")
head(ac.off)
```

```
##            [,1]        [,2]
## [1,] 0.07604146 -0.07604146
## [2,] 0.07604146 -0.07604146
## [3,] 0.07604146 -0.07604146
## [4,] 0.07637039 -0.07637039
## [5,] 0.07668889 -0.07668889
## [6,] 0.07739743 -0.07739743
```

When `se.out=TRUE`, the offsets are stored in the *RangedSummarizedExperiment* object as an `"offsets"` entry in the `assays` slot. Each offset represents the log-transformed scaling factor that needs to be applied to the corresponding entry of the count matrix for its normalization. Any operations like subsetting that are applied to modify the object will also be applied to the offsets, allowing for synchronised processing.

4.6.2 Characteristics of loess normalization

Loess normalization of trended biases is quite similar to TMM normalization for efficiency biases described in Section 4.3. Both methods assume a non-DB majority across features, and will not be appropriate if there is a change in overall binding. Loess normalization involves a slightly stronger assumption of a non-DB majority at every abundance, not just across all bound regions. This is necessary to remove trended biases but may also discard genuine changes, such as a subset of DB sites at very high abundances.

Compared to TMM normalization, the accuracy of loess normalization is less dependent on stringent filtering. This is because the use of a trend accommodates changes in the bias between high-abundance binding sites and low-abundance background regions. Nonetheless, some filtering is still necessary to avoid inaccuracies in loess fitting at low abundances. Any filter statistic for the windows should be based on the average abundance from `aveLogCPM`, such as those calculated using `filterWindowsGlobal` or equivalents. An average abundance threshold will act as a clean vertical cutoff in the MA plots above. This avoids introducing spurious trends at the filter boundary that might affect normalization.

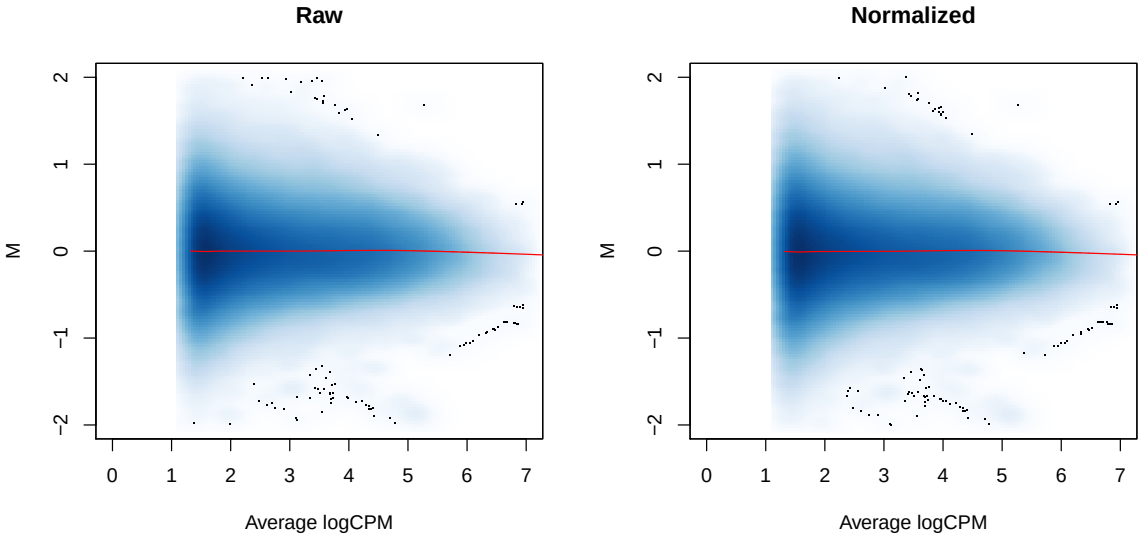
4.6.3 Checking normalization with MA plots

We examine the MA plots to determine whether normalization was successful. Any abundance-dependent trend in the M-values should be eliminated after applying the offsets to the log-counts. This is done by subtraction, though note that the offsets are base e while most log-values in *edgeR* are reported as base 2.

```
par(mfrow=c(1,2))

# MA plot without normalization.
ac.y <- asDGEList(ac.demo2)
lib.size <- ac.y$samples$lib.size
adjc <- cpm(ac.y, log=TRUE)
abval <- aveLogCPM(ac.y)
mval <- adjc[,1]-adjc[,2]
fit <- loessFit(x=abval, y=mval)
smoothScatter(abval, mval, ylab="M", xlab="Average logCPM",
              main="Raw", ylim=c(-2,2), xlim=c(0, 7))
o <- order(abval)
lines(abval[o], fit$fitted[o], col="red")

# Repeating after normalization.
re.adjc <- log2(assay(ac.demo2)+0.5) - ac.off/log(2)
mval <- re.adjc[,1]-re.adjc[,2]
fit <- loessFit(x=abval, y=mval)
smoothScatter(abval, re.adjc[,1]-re.adjc[,2], ylab="M", xlab="Average logCPM",
              main="Normalized", ylim=c(-2,2), xlim=c(0, 7))
lines(abval[o], fit$fitted[o], col="red")
```



4.7 A word on other biases

No normalization is performed to adjust for differences in mappability or sequencability between different regions of the genome. Region-specific biases are assumed to be constant between libraries. This is generally reasonable as the biases depend on fixed properties of the genome sequence such as GC content. Thus, biases should cancel out during DB comparisons. Any variability between samples will just be absorbed into the dispersion estimate.

That said, explicit normalization to correct these biases can improve results for some datasets. Procedures like GC correction could decrease the observed variability by removing systematic differences between replicates. Of course, this also assumes that the targeted differences have no biological relevance. Detection power may be lost if this is not true. For example, differences in the GC content distribution can be driven by technical bias as well as biology, e.g., when protein binding is associated with a specific GC composition.

Chapter 5

Testing for differential binding

Here, we'll need the `filtered.data` generated in Chapter 3 (and modified in Chapter 4), and the `design` matrix from the introduction. We'll also need the original `data` from Chapter 2, but just for a demonstration. Finally, we'll be executing a number of *edgeR* functions, so make sure the *edgeR* package is loaded if you've been skipping chapters.

5.1 Introduction to *edgeR*

5.1.1 Overview

Low counts per window are typically observed in ChIP-seq datasets, even for genuine binding sites. Any statistical analysis to identify DB sites must be able to handle discreteness in the data. Count-based models are ideal for this purpose. In this guide, the quasi-likelihood (QL) framework in the *edgeR* package is used [18]. Counts are modelled using NB distributions that account for overdispersion between biological replicates [19]. Each window can then be tested for significant DB between conditions.

Of course, any statistical method can be used if it is able to accept a count matrix and a vector of normalization factors (or more generally, a matrix of offsets). The choice of *edgeR* is primarily motivated by its performance relative to some published alternatives [20]. This author's desire to increase his h-index may also be a factor [21].

5.1.2 Setting up the data

A *DGEList* object is first constructed from the count matrix in `filtered.data`. If normalization factors or offsets are present in the *RangedSummarizedExperiment* object – see Chapter 4 – they will automatically be extracted and used to construct the *DGEList* object. Otherwise, they can be manually passed to the `asDGEList` function. If offsets are available, they will generally override the normalization factors in the downstream *edgeR* analysis.

```
y <- asDGEList(filtered.data)
```

The experimental design is described by a design matrix. In this case, the only relevant factor is the cell type of each sample. A generalized linear model (GLM) will be fitted to the counts for each window using the specified design [22]. This provides a general framework for the analysis of complex experiments with multiple factors. Readers are referred to the user's guide in *edgeR* for more details on parametrization.

```
tf.data <- NFYAData()
cell.type <- sub("NF-YA ([^ ]+) .*", "\\1", head(tf.data$Description, -1))
cell.type
```

```
## [1] "ESC" "ESC" "TN"  "TN"
```

```
design <- model.matrix(~factor(cell.type))
colnames(design) <- c("intercept", "cell.type")
design
```

```
##   intercept cell.type
## 1         1         0
## 2         1         0
## 3         1         1
## 4         1         1
## attr(,"assign")
## [1] 0 1
## attr(,"contrasts")
## attr(,"contrasts")$`factor(cell.type)`
## [1] "contr.treatment"
```

5.2 Estimating the dispersions

5.2.1 Stabilising estimates with empirical Bayes

Under the QL framework, both the QL and NB dispersions are used to model biological variability in the data [18]. The former ensures that the NB mean-variance relationship is properly specified with appropriate contributions from the Poisson and Gamma components. The latter accounts for variability and uncertainty in the dispersion estimate. However, limited replication in most ChIP-seq experiments means that each window does not contain enough information for precise estimation of either dispersion.

This problem is overcome in *edgeR* by sharing information across windows. For the NB dispersions, a mean-dispersion trend is fitted across all windows to model the mean-variance relationship [22]. The raw QL dispersion for each window is estimated after fitting a GLM with the trended NB dispersion. Another mean-dependent trend is fitted to the raw QL estimates. An empirical Bayes (EB) strategy is then used to stabilize the raw QL dispersion estimates by shrinking them towards the second trend [18]. The ideal amount of shrinkage is determined from the variability of the dispersions.

```
y <- estimateDisp(y, design)
summary(y$trended.dispersion)
```

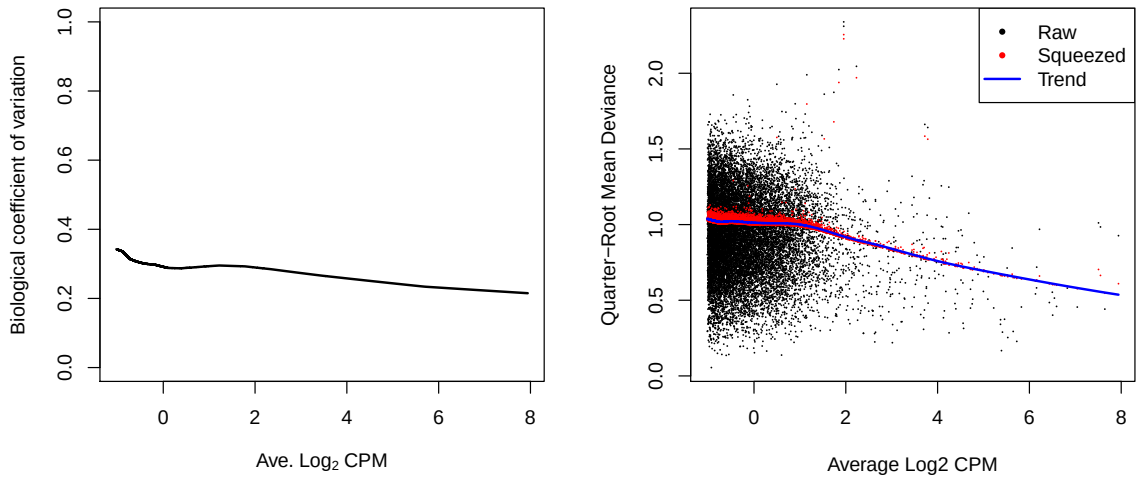
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.04627 0.08514 0.09030 0.09384 0.09978 0.11725
```

```
fit <- glmQLFit(y, design, robust=TRUE)
summary(fit$var.post)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##  0.08929  1.02731  1.05983  1.06655  1.10610 25.87131
```

The effect of EB stabilisation can be visualized by examining the biological coefficient of variation (for the NB dispersion) and the quarter-root deviance (for the QL dispersion). These plots can also be used to decide whether the fitted trend is appropriate. Sudden irregularities may be indicative of an underlying structure in the data which cannot be modelled with the mean-dispersion trend. Discrete patterns in the raw dispersions are indicative of low counts and suggest that more aggressive filtering is required.

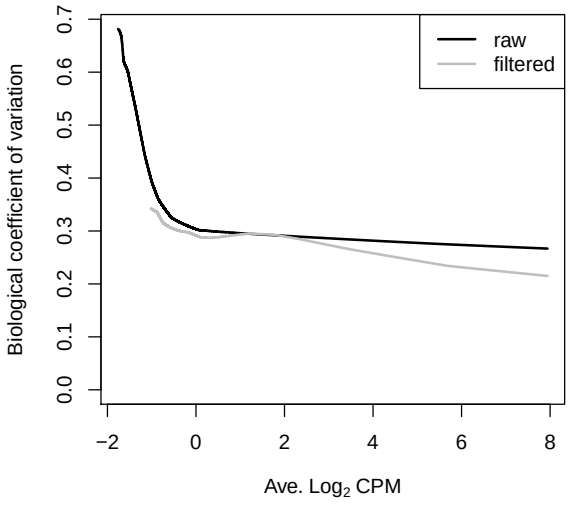
```
par(mfrow=c(1,2))
o <- order(y$AveLogCPM)
plot(y$AveLogCPM[o], sqrt(y$trended.dispersion[o]), type="l", lwd=2,
     ylim=c(0, 1), xlab=expression("Ave."~Log[2]~"CPM"),
     ylab=("Biological coefficient of variation"))
plotQLDisp(fit)
```



A strong trend may also be observed where the NB dispersion drops sharply with increasing average abundance. This is due to the disproportionate impact of artifacts such as mapping errors and PCR duplicates at low counts. It is difficult to accurately fit an empirical curve to these strong trends. As a consequence, the dispersions at high abundances may be overestimated. Filtering of low-abundance regions (as described in Chapter 3) provides some protection by removing the strongest part of the trend. Users can compare raw and filtered results to see whether it makes any difference. Filtering has an additional benefit of removing those tests that have low power due to the magnitude of the dispersions.

```
relevant <- rowSums(assay(data)) >= 20 # weaker filtering than 'filtered.data'
yo <- asDGEList(data[relevant], norm.factors=normfacs)
yo <- estimateDisp(yo, design)
oo <- order(yo$AveLogCPM)
plot(yo$AveLogCPM[oo], sqrt(yo$trended.dispersion[oo]), type="l", lwd=2,
     ylim=c(0, max(sqrt(yo$trended))), xlab=expression("Ave."~Log[2]~"CPM"),
     ylab=("Biological coefficient of variation"))
lines(y$AveLogCPM[o], sqrt(y$trended[o]), lwd=2, col="grey")
```

```
legend("topright", c("raw", "filtered"), col=c("black", "grey"), lwd=2)
```



5.2.2 Modelling variable dispersions between windows

Any variability in the dispersions across windows is modelled in *edgeR* by the prior degrees of freedom (d.f.). A large value for the prior d.f. indicates that the variability is low. This means that more EB shrinkage can be performed to reduce uncertainty and maximize power. However, strong shrinkage is not appropriate if the dispersions are highly variable. Fewer prior degrees of freedom (and less shrinkage) are required to maintain type I error control.

```
summary(fit$df.prior)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.3249 37.0011 37.0011 36.9438 37.0011 37.0011
```

On occasion, the estimated prior degrees of freedom will be infinite. This is indicative of a strong batch effect where the dispersions are consistently large. A typical example involves uncorrected differences in IP efficiency across replicates. In severe cases, the trend may fail to pass through the bulk of points as the variability is too low to be properly modelled in the QL framework. This problem is usually resolved with appropriate normalization.

Note that the prior degrees of freedom should be robustly estimated [23]. Obviously, this protects against large positive outliers (e.g., highly variable windows) but it also protects against near-zero dispersions at low counts. These will manifest as large negative outliers after a log transformation step during estimation [24]. Without robustness, incorporation of these outliers will inflate the observed variability in the dispersions. This results in a lower estimated prior d.f. and reduced DB detection power.

5.3 Testing for DB windows

The effect of specific factors can be tested to identify windows with significant differential binding. In the QL framework, p-values are computed using the QL F-test [18]. This is more appropriate than using the likelihood ratio test as the F-test accounts for uncertainty in the dispersion estimates. Associated statistics such as log-fold changes and log-counts per million are also computed for each window.

```
results <- glmQLFTest(fit, contrast=c(0, 1))
head(results$table)
```

```
##       logFC     logCPM        F     PValue
## 1 1.8295648 -0.9631454 5.669892 0.02223713
## 2 0.9413170 -0.8081324 1.907469 0.17510744
## 3 0.9233936  0.4965227 3.407509 0.07250075
## 4 1.0986454  1.3167989 5.784324 0.02101349
## 5 1.2664250  1.2826709 7.192270 0.01067931
## 6 0.7935887  0.7798144 2.436384 0.12662798
```

The null hypothesis here is that the cell type has no effect. The `contrast` argument in the `glmQLFTest` function specifies which factors are of interest. In this case, a contrast of `c(0, 1)` defines the null hypothesis as `0*intercept + 1*cell.type = 0`, i.e., that the log-fold change between cell types is zero. DB windows can then be identified by rejecting the null. Specification of the contrast is explained in greater depth in the *edgeR* user's manual.

Once the significance statistics have been calculated, they can be stored in row metadata of the *RangedSummarizedExperiment* object. This ensures that the statistics and coordinates are processed together, e.g., when subsetting to select certain windows.

```
rowData(filtered.data) <- cbind(rowData(filtered.data), results$table)
```

5.4 What to do without replicates

Designing a ChIP-seq experiment without any replicates is strongly discouraged. Without replication, the reproducibility of any findings cannot be determined. Nonetheless, it may be helpful to salvage some information from datasets that lack replicates. This is done by supplying a "reasonable" value for the NB dispersion during GLM fitting (e.g., 0.05 - 0.1, based on past experience). DB windows are then identified using the likelihood ratio test.

```
fit.norep <- glmFit(y, design, dispersion=0.05)
results.norep <- glmLRT(fit.norep, contrast=c(0, 1))
head(results.norep$table)
```

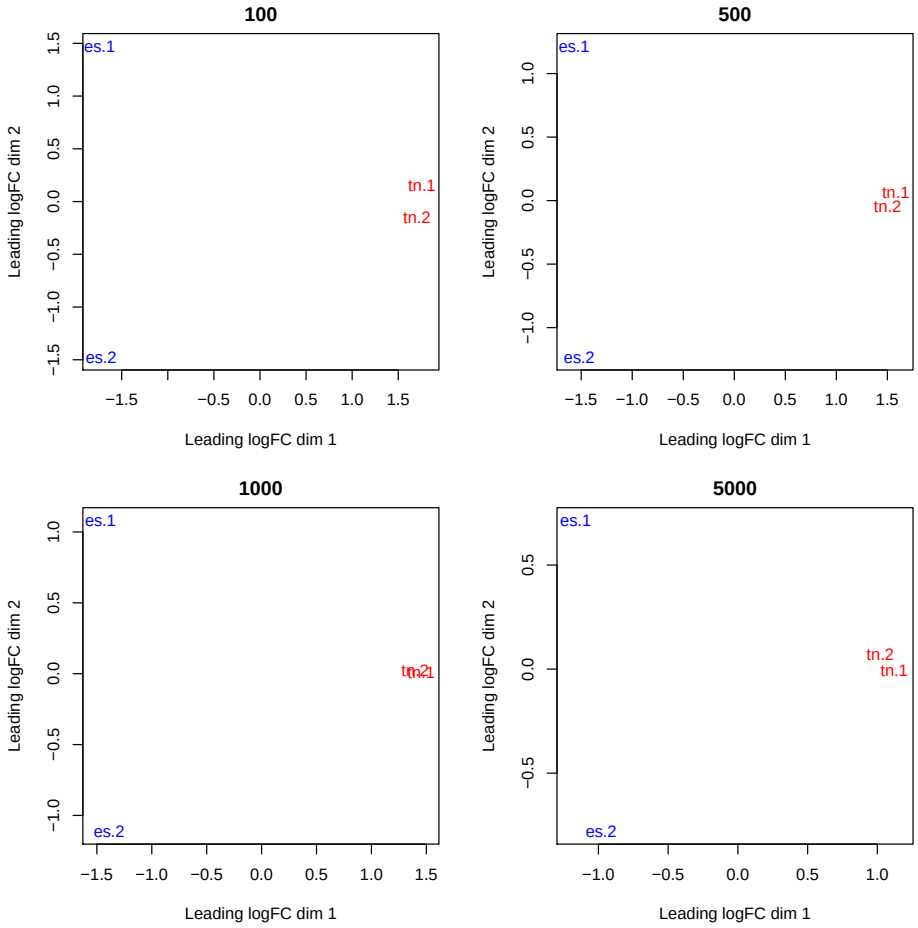
```
##       logFC     logCPM        LR      PValue
## 1 1.8335902 -0.9631454  8.576795 0.003404741
## 2 0.9335664 -0.8081324  2.793258 0.094661575
## 3 0.9217906  0.4965227  4.653039 0.030998626
## 4 1.0960626  1.3167989  8.004891 0.004665117
## 5 1.2631123  1.2826709 10.406617 0.001255646
## 6 0.7878871  0.7798144  3.743536 0.053012161
```

Obviously, this approach has a number of pitfalls. The lack of replicates means that the biological variability in the data cannot be modelled. Thus, it becomes impossible to gauge the sensibility of the supplied NB dispersions in the analysis. Another problem is spurious DB due to inconsistent PCR duplication between libraries. Normally, inconsistent duplication results in a large QL dispersion for the affected window, such that significance is downweighted. However, estimation of the QL dispersion is not possible without replicates. This means that duplicates may need to be removed to protect against false positives.

5.5 Examining replicate similarity with MDS plots

As a quality control measure, the window counts can be used to examine the similarity of replicates through multi-dimensional scaling (MDS) plots. The distance between each pair of libraries is computed as the square root of the mean squared log-fold change across the top set of bins with the highest absolute log-fold changes. A small top set visualizes the most extreme differences whereas a large set visualizes overall differences. Checking a range of `top` values may be useful when the scope of DB is unknown. Again, counting with large bins is recommended as fold changes will be undefined in the presence of zero counts.

```
par(mfrow=c(2,2), mar=c(5,4,2,2))
adj.counts <- cpm(y, log=TRUE)
for (top in c(100, 500, 1000, 5000)) {
    out <- plotMDS(adj.counts, main=top, col=c("blue", "blue", "red", "red"),
                   labels=c("es.1", "es.2", "tn.1", "tn.2"), top=top)
}
```



Replicates from different groups should form separate clusters in the MDS plot, as observed above. This indicates that the results are reproducible and that the effect sizes are large. Mixing between replicates of different conditions indicates that the biological difference has no effect on protein binding, or that the data is too variable for any effect to manifest. Any outliers should also be noted as their presence may confound the downstream analysis. In the worst case, outlier samples may need to be removed to obtain sensible results.

Chapter 6

Correction for multiple testing

All right, we're almost there. This chapter needs the `results` object from the last chapter, as well as `filtered.data` from Chapter 3. A few other things are also required for some of the optional sections below – namely, `broads` from Chapter 3, and `ac.files` and `param` from Chapters 4 and 2, respectively.

6.1 Problems with false discovery rate control

6.1.1 Overview

The false discovery rate (FDR) is usually the most appropriate measure of error for high-throughput experiments. Control of the FDR can be provided by applying the Benjamini-Hochberg (BH) method [25] to a set of p-values. This is less conservative than the alternatives (e.g., Bonferroni) yet still provides some measure of error control. The most obvious approach is to apply the BH method to the set of p-values across all windows. This will control the FDR across the set of putative DB windows.

However, the FDR across all detected windows is not necessarily the most relevant error rate. Interpretation of ChIP-seq experiments is more concerned with regions of the genome in which (differential) protein binding is found, rather than the individual windows. In other words, the FDR across all detected DB regions is usually desired. This is not equivalent to that across all DB windows as each region will often consist of multiple overlapping windows. Control of one will not guarantee control of the other [4].

To illustrate this difference, consider an analysis where the FDR across all window positions is controlled at 10%. In the results, there are 18 adjacent window positions in one region and 2 windows in a separate region. The first set of windows is a truly DB region whereas the second set is a false positive. A window-based interpretation of the FDR is correct as only 2 of the 20 window positions are false positives. However, a region-based interpretation results in an actual FDR of 50%.

To avoid misinterpretation of the FDR, *csaw* provides a number of strategies to obtain region-level results. This involves defining the regions of interest - possibly from the windows themselves - and converting per-window statistics into a p-value for each region. Application of the BH method to the per-region p-values will then control the relevant FDR across regions. These strategies are demonstrated below using the NF-YA data.

6.2 Grouping windows into regions

6.2.1 Quick and dirty clustering

The `mergeWindows` function provides a simple single-linkage algorithm to cluster windows into regions. Windows that are less than `tol` apart are considered to be adjacent and are grouped into the same cluster. The chosen `tol` represents the minimum distance at which two binding events are treated as separate sites. Large values (500 - 1000 bp) reduce redundancy and favor a region-based interpretation of the results, while smaller values ($<$ 200 bp) allow resolution of individual binding sites.

```
merged <- mergeWindows(filtered.data, tol=1000L)
merged$regions
```

```
## GRanges object with 4433 ranges and 0 metadata columns:
##          seqnames            ranges strand
##             <Rle>         <IRanges>  <Rle>
##      [1]     chr1   7088951-7088960      *
##      [2]     chr1   7397851-7398110      *
##      [3]     chr1   9541401-9541510      *
##      [4]     chr1   9545251-9545360      *
##      [5]     chr1   9748451-9748460      *
##      ...      ...               ...    ...
##   [4429]     chrY   4065001-4065010      *
##   [4430]     chrY   4149901-4149910      *
##   [4431]     chrY   4160201-4160310      *
##   [4432]     chrY 90808801-90808860      *
##   [4433]     chrY 90812051-90812910      *
##   -------
##   seqinfo: 66 sequences from an unspecified genome
```

If many adjacent windows are present, very large clusters may be formed that are difficult to interpret. We perform a simple check below to determine whether most clusters are of an acceptable size. Huge clusters indicate that more aggressive filtering from Chapter 3 is required. This mitigates chaining effects by reducing the density of windows in the genome.

```
summary(width(merged$regions))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    10.0    60.0   110.0   172.1   160.0 15660.0
```

Alternatively, chaining can be limited by setting `max.width` to restrict the size of the merged intervals. Clusters substantially larger than `max.width` are split into several smaller subclusters of roughly equal size. The chosen value should be small enough so as to separate DB regions

from unchanged neighbors, yet large enough to avoid misinterpretation of the FDR. Any value from 2000 to 10000 bp is recommended. This paramater can also interpreted as the maximum distance at which two binding sites are considered part of the same event.

```
merged.max <- mergeWindows(filtered.data, tol=1000L, max.width=5000L)
summary(width(merged.max$regions))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    10.0    60.0   110.0   170.6   160.0  4960.0
```

6.2.2 Using external information

Another approach is to group together windows that overlap with a pre-specified region of interest. The most obvious source of pre-specified regions is that of annotated features such as promoters or gene bodies. Alternatively, called peaks can be used provided that sufficient care has been taken to avoid loss of error control from data snooping [4]. Regardless of how they are specified, each region of interest corresponds to a group that contains all overlapping windows, as identified by the `findOverlaps` function from the *GenomicRanges* package.

```
olap <- findOverlaps(broads, rowRanges(filtered.data))
olap
```

```
## Hits object with 16384 hits and 0 metadata columns:
##           queryHits subjectHits
##           <integer>   <integer>
##       [1]         7        8922
##       [2]         7        8923
##       [3]         7        8924
##       [4]        18       10619
##       [5]        18       10620
##       ...       ...         ...
##   [16380]     24521        8725
##   [16381]     24521        8726
##   [16382]     24524        8438
##   [16383]     24524        8439
##   [16384]     24524        8440
##   -------
##   queryLength: 24528 / subjectLength: 15708
```

At this point, one might imagine that it would be simpler to just collect and analyze counts over the pre-specified regions. This is a valid strategy but will yield different results. Consider a promoter containing two separate sites that are identically DB in opposite directions. Counting reads across the promoter will give equal counts for each condition so changes within the promoter will not be detected. Similarly, imprecise peak boundaries can lead to loss of detection power due to "contamination" by reads in background regions. Window-based methods may be more robust as each interval of the promoter/peak region is examined separately [4], avoiding potential problems with peak-calling errors and incorrect/incomplete annotation.

6.3 Obtaining per-region p-value

6.3.1 Combining window-level p-values

We compute a combined p-value for each region based on the p-values of the constituent windows [26]. This tests the joint null hypothesis for each region, i.e., that no enrichment is observed across any of its windows. Any DB within the region will reject the joint null and yield a low p-value for the entire region. The combined p-values are then adjusted using the BH method to control the region-level FDR.

```
tabcom <- combineTests(merged$ids, results$table)
is.sig.region <- tabcom$FDR <= 0.05
summary(is.sig.region)
```

```
##    Mode   FALSE    TRUE
## logical    2945    1488
```

Summarizing the direction of DB for each cluster requires some care as the direction of DB can differ between constituent windows. The `num.up.tests` and `num.down.tests` fields contain the number of windows that change in each direction, and can be used to gauge whether binding increases or decreases across the cluster. A complex DB event may be present if both `num.up.tests` and `num.down.tests` are non-zero (i.e., opposing changes within the region) or if the total number of windows is much larger than either number (e.g., interval of constant binding adjacent to the DB interval).

Alternatively, the `direction` field specifies which DB direction contributes to the combined p-value. If `"up"`, the combined p-value for this cluster is driven by p-values of windows with positive log-fold changes. If `"down"`, the combined p-value is driven by windows with negative log-fold changes. If `"mixed"`, windows with both positive and negative log-fold changes are involved. This allows the dominant DB in significant clusters to be quickly summarized, as shown below.

```
table(tabcom$direction[is.sig.region])
```

```
##
## down   up
##  246 1242
```

For pre-specified regions, the `combineOverlaps` function will combine the p-values for all windows in each region. This is a wrapper around `combineTests` for *Hits* objects. It returns a single combined p-value (and its BH-adjusted value) for each region. Regions that do not overlap any windows have values of `NA` in all fields for the corresponding rows.

```
tabbroad <- combineOverlaps(olap, results$table)
head(tabbroad[!is.na(tabbroad$PValue),])
```

```
## DataFrame with 6 rows and 8 columns
##    num.tests num.up.logFC num.down.logFC      PValue        FDR   direction
##    <integer>    <integer>      <integer>   <numeric>  <numeric> <character>
## 7          3            3              0 0.000354345 0.00461264          up
## 18         4            4              0 0.000831695 0.00733820          up
## 23         4            0              0 0.084562340 0.14567298          up
## 25         4            0              0 0.866537670 0.89772103       mixed
```

```
## 28         3            3              0 0.000553808 0.00585212          up
## 36         4            3              0 0.016347528 0.04758053          up
##     rep.test rep.logFC
##    <integer> <numeric>
## 7       8923 3.0804339
## 18     10622 3.1398280
## 23       413 0.9810843
## 25     12754 0.0932641
## 28     11210 3.1944940
## 36      3508 1.7984907
```

```
is.sig.gene <- tabcom$FDR <= 0.05
table(tabbroad$direction[is.sig.gene])
```

```
##
##  down mixed    up
##   108    54  1318
```

6.3.2 Based on the most significant window

Another approach is to use the single window with the strongest DB as a representative of the entire region. This is useful when a log-fold change is required for each cluster, e.g., for plotting. (In contrast, taking the average log-fold change across all windows in a region will understate the magnitude of DB, especially if the region includes some non-DB background intervals of the genome.) Identification of the most significant (i.e., "best") window is performed using the `getBestTest` function. This reports the index of the window with the lowest p-value in each cluster as well as the associated statistics.

```
tab.best <- getBestTest(merged$ids, results$table)
head(tab.best)
```

```
## DataFrame with 6 rows and 8 columns
##   num.tests num.up.logFC num.down.logFC    PValue       FDR   direction
##   <integer>    <integer>      <integer> <numeric> <numeric> <character>
## 1         1            1              0 0.0222371 0.0628280          up
## 2         6            0              0 0.0640759 0.1316257          up
## 3         3            2              0 0.0369508 0.0893632          up
## 4         3            0              0 0.1316684 0.2210931          up
## 5         1            1              0 0.0146024 0.0470966          up
## 6         2            0              0 0.4323988 0.5538352          up
##    rep.test rep.logFC
##   <integer> <numeric>
## 1         1  1.829565
## 2         5  1.266425
## 3        10  1.681574
## 4        13  1.378296
## 5        14  1.922191
## 6        15  0.818257
```

A Bonferroni correction is applied to the p-value of the best window in each region, based on the number of constituent windows in that region. This is necessary to account for the implicit multiple testing across all windows in each region. The corrected p-value is reported as `PValue` in `tab.best`, and can be used for correction across regions using the BH method to control the region-level FDR.

In addition, it is often useful to report the start location of the best window within each cluster. This allows users to easily identify a relevant DB subinterval in large regions. For example, the sequence of the DB subinterval can be extracted for motif discovery.

```
tabcom$rep.start <- start(rowRanges(filtered.data))[tab.best$rep.test]
head(tabcom[,c("rep.logFC", "rep.start")])
```

```
## DataFrame with 6 rows and 2 columns
##   rep.logFC rep.start
##   <numeric> <integer>
## 1   1.82956   7088951
## 2   1.09865   7398001
## 3   1.51933   9541501
## 4   1.28005   9545351
## 5   1.92219   9748451
## 6   0.72853  10007401
```

The same approach can be applied to the overlaps between windows and pre-specified regions, using the `getBestOverlaps` wrapper function. This is demonstrated below for the broad gene body example. As with `combineOverlaps`, regions with no windows are assigned `NA` in the output table, but these are removed here to show some actual results.

```
tab.best.broad <- getBestOverlaps(olap, results$table)
tabbroad$rep.start <- start(rowRanges(filtered.data))[tab.best.broad$rep.test]
head(tabbroad[!is.na(tabbroad$PValue),c("rep.logFC", "rep.start")])
```

```
## DataFrame with 6 rows and 2 columns
##    rep.logFC rep.start
##    <numeric> <integer>
## 7  3.0804339  32657101
## 18 3.1398280   8259301
## 23 0.9810843  92934601
## 25 0.0932641  71596001
## 28 3.1944940   4137001
## 36 1.7984907 100187601
```

6.3.3 Wrapper functions

For convenience, the steps of merging windows and computing statistics are implemented into a single wrapper function. This simply calls `mergeWindows` followed by `combineTests` and `getBestTest`.

```
merge.res <- mergeResults(filtered.data, results$table, tol=100,
    merge.args=list(max.width=5000))
names(merge.res)
```

```
## [1] "regions"  "combined" "best"
```

An equivalent wrapper function is also available for handling overlaps to pre-specified regions. This simply calls `findOverlaps` followed by `combineOverlaps` and `getBestOverlaps`.

```
broad.res <- overlapResults(filtered.data, regions=broads,
    tab=results$table)
names(broad.res)
```

```
## [1] "regions"  "combined" "best"
```

6.4 Squeezing out more detection power

6.4.1 Integrating results from multiple window sizes

Repeating the analysis with different window sizes may uncover new DB events at different resolutions. Multiple sets of DB results are integrated by clustering adjacent windows together (even if they differ in size) and combining p-values within each of the resulting clusters. The example below uses the H3 acetylation data from Chapter 4. Some filtering is performed to avoid excessive chaining in this demonstration. Corresponding tables of DB results should also be obtained – for brevity, mock results are used here.

```
ac.small <- windowCounts(ac.files, width=150L, spacing=100L,
    filter=25, param=param)
ac.large <- windowCounts(ac.files, width=1000L, spacing=500L,
    filter=35, param=param)

# Mocking up results for demonstration purposes.
ns <- nrow(ac.small)
mock.small <- data.frame(logFC=rnorm(ns), logCPM=0, PValue=runif(ns))
nl <- nrow(ac.large)
mock.large <- data.frame(logFC=rnorm(nl), logCPM=0, PValue=runif(nl))
```

The `mergeResultsList` function merges windows of all sizes into a single set of regions, and computes a combined p-value from the associated p-values for each region. Equal contributions from each window size are enforced by setting `equiweight=TRUE`, which uses a weighted version of Simes' method [27]. The weight assigned to each window is inversely proportional to the number of windows of that size in the same cluster. This avoids the situation where, if a cluster contains many small windows, the DB results for the analysis with the small window size contribute most to the combined p-value. This is not ideal when results from all window sizes are of equal interest.

```
cons.res <- mergeResultsList(list(ac.small, ac.large),
    tab.list=list(mock.small, mock.large),
    equiweight=TRUE, tol=1000)
cons.res$regions
```

```
## GRanges object with 21726 ranges and 0 metadata columns:
##                       seqnames          ranges strand
##                          <Rle>       <IRanges>  <Rle>
##       [1]                 chr1 4774501-4776000      *
##       [2]                 chr1 4784501-4787000      *
```

```
##       [3]                 chr1 4807001-4809000      *
##       [4]                 chr1 4856501-4859500      *
##       [5]                 chr1 5082501-5084500      *
##       ...                  ...             ...    ...
##   [21722] chrX_GL456233_random   336001-336933      *
##   [21723]                 chrY   897501-898500      *
##   [21724]                 chrY 1010001-1011500      *
##   [21725]                 chrY 1244501-1246000      *
##   [21726]                 chrY 1285501-1287000      *
##   -------
##   seqinfo: 66 sequences from an unspecified genome
```

```
cons.res$combined
```

```
## DataFrame with 21726 rows and 8 columns
##       num.tests num.up.logFC num.down.logFC    PValue       FDR   direction
##       <integer>    <integer>      <integer> <numeric> <numeric> <character>
## 1             2            0              0 0.4127682  0.997611          up
## 2            13            0              0 0.4800261  0.997611        down
## 3             7            0              0 0.2114586  0.985842        down
## 4            18            0              0 0.0971028  0.983147          up
## 5             7            0              0 0.9308619  0.997611       mixed
## ...         ...          ...            ...       ...       ...         ...
## 21722         5            0              0  0.788422  0.997611       mixed
## 21723         1            0              0  0.949679  0.998493        down
## 21724         3            0              0  0.962915  0.999487       mixed
## 21725         4            0              0  0.744565  0.997611       mixed
## 21726         4            0              0  0.468441  0.997611          up
##        rep.test rep.logFC
##       <integer> <numeric>
## 1        179205  0.819856
## 2             8 -0.364467
## 3            13 -1.648852
## 4            17  0.601116
## 5            30 -0.447841
## ...         ...       ...
## 21722    280929 -0.228905
## 21723    280930 -0.528653
## 21724    280931  0.296111
## 21725    179200  0.175919
## 21726    280935  0.121840
```

Similarly, the `overlapResultsList` function is used to merge windows of varying size that overlap pre-specified regions.

```
cons.broad <- overlapResultsList(list(ac.small, ac.large),
    tab.list=list(mock.small, mock.large),
    equiweight=TRUE, region=broads)
cons.broad$regions
```

```
## GRanges object with 24528 ranges and 1 metadata column:
##             seqnames              ranges strand |     gene_id
##                <Rle>           <IRanges>  <Rle> | <character>
```

```
##   100009600     chr9   21062393-21076096      - |   100009600
##   100009609     chr7   84935565-84967115      - |   100009609
##   100009614    chr10   77708457-77712009      + |   100009614
##   100009664    chr11   45805087-45841171      + |   100009664
##      100012     chr4 144157557-144165663      - |      100012
##         ...      ...                 ...    ... .         ...
##       99889     chr3   84496093-85890516      - |       99889
##       99890     chr3 110246109-110253998      - |       99890
##       99899     chr3 151730922-151752960      - |       99899
##       99929     chr3   65525410-65555518      + |       99929
##       99982     chr4 136550540-136605723      - |       99982
##   -------
##   seqinfo: 66 sequences (1 circular) from mm10 genome
```

```
cons.res$combined
```

```
## DataFrame with 21726 rows and 8 columns
##       num.tests num.up.logFC num.down.logFC    PValue       FDR   direction
##       <integer>    <integer>      <integer> <numeric> <numeric> <character>
## 1             2            0              0 0.4127682  0.997611          up
## 2            13            0              0 0.4800261  0.997611        down
## 3             7            0              0 0.2114586  0.985842        down
## 4            18            0              0 0.0971028  0.983147          up
## 5             7            0              0 0.9308619  0.997611       mixed
## ...         ...          ...            ...       ...       ...         ...
## 21722         5            0              0  0.788422  0.997611       mixed
## 21723         1            0              0  0.949679  0.998493        down
## 21724         3            0              0  0.962915  0.999487       mixed
## 21725         4            0              0  0.744565  0.997611       mixed
## 21726         4            0              0  0.468441  0.997611          up
##        rep.test rep.logFC
##       <integer> <numeric>
## 1        179205  0.819856
## 2             8 -0.364467
## 3            13 -1.648852
## 4            17  0.601116
## 5            30 -0.447841
## ...         ...       ...
## 21722    280929 -0.228905
## 21723    280930 -0.528653
## 21724    280931  0.296111
## 21725    179200  0.175919
## 21726    280935  0.121840
```

In this manner, DB results from multiple window widths can be gathered together and reported as a single set of regions. Consolidation is most useful for histone marks and other analyses involving diffuse regions of enrichment. For such studies, the ideal window size is not known or may not even exist, e.g., if the widths of the enriched regions or DB subintervals are variable.

6.4.2 Weighting windows on abundance

Windows that are more likely to be DB can be upweighted to improve detection power. For example, in TF ChIP-seq data, the window of highest abundance within each enriched region probably contains the binding site. It is reasonable to assume that this window will also have the strongest DB. To improve power, the weight assigned to the most abundant window is increased relative to that of other windows in the same cluster. This means that the p-value of this window will have a greater influence on the final combined p-value.

Weights are computed in a manner to minimize conservativeness relative to the optimal unweighted approaches in each possible scenario. If the strongest DB event is at the most abundant window, the weighted approach will yield a combined p-value that is no larger than twice the p-value of the most abundant window. (Here, the optimal approach would be to use the p-value of the most abundance window directly as a proxy for the p-value of the cluster.) If the strongest DB event is *not* at the most abundant window, the weighted approach will yield a combined p-value that is no larger than twice the combined p-value without wweighting (which is optimal as all windows have equal probabilities of containing the strongest DB). All windows have non-zero weights, which ensures that any DB events in the other windows will still be considered when the p-values are combined.

The application of this weighting scheme is demonstrated in the example below. First, the `getBestTest` function with `by.pval=FALSE` is used to identify the most abundant window in each cluster. Window-specific weights are then computed using the `upweightSummits` function, and supplied to `combineTests` to use in computing combined p-values.

```
tab.ave <- getBestTest(merged$id, results$table, by.pval=FALSE)
weights <- upweightSummit(merged$id, tab.ave$rep.test)
head(weights)
```

```
## [1] 1 1 1 6 1 1
```

```
tabcom.w <- combineTests(merged$id, results$table, weight=weights)
head(tabcom.w)
```

```
## DataFrame with 6 rows and 8 columns
##   num.tests num.up.logFC num.down.logFC    PValue       FDR   direction
##   <integer>    <integer>      <integer> <numeric> <numeric> <character>
## 1         1            1              0 0.0222371 0.0595633          up
## 2         6            2              0 0.0330212 0.0785274          up
## 3         3            2              0 0.0223809 0.0598038          up
## 4         3            0              0 0.0853882 0.1522629          up
## 5         1            1              0 0.0146024 0.0442498          up
## 6         2            0              0 0.2600941 0.3505617          up
##    rep.test rep.logFC
##   <integer> <numeric>
## 1         1   1.82956
## 2         4   1.09865
## 3         9   1.51933
## 4        12   1.28005
## 5        14   1.92219
## 6        16   0.72853
```

The weighting approach can also be applied to the clusters from the broad gene body example. This is done by replacing the call to `getBestTest` with one to `getBestOverlaps`, as before. Similarly, `upweightSummit` can be replaced with `summitOverlaps`. These wrappers are designed to minimize book-keeping problems when one window overlaps multiple regions.

```
broad.best <- getBestOverlaps(olap, results$table, by.pval=FALSE)
head(broad.best[!is.na(broad.best$PValue),])
```

```
## DataFrame with 6 rows and 8 columns
##    num.tests num.up.logFC num.down.logFC      PValue        FDR   direction
##    <integer>    <integer>      <integer>   <numeric>  <numeric> <character>
## 7          3            1              0 0.000118115 0.00217479          up
## 18         4            1              0 0.015705062 0.04689590          up
## 23         4            0              0 0.084562340 0.14823164          up
## 25         4            0              0 0.865505096 0.88784454        down
## 28         3            1              0 0.000184603 0.00285879          up
## 36         4            1              0 0.008016993 0.03020496          up
##     rep.test  rep.logFC
##    <integer>  <numeric>
## 7       8923  3.0804339
## 18     10620  1.3982113
## 23       413  0.9810843
## 25     12753 -0.0962083
## 28     11210  3.1944940
## 36      3509  1.6380682
```

```
broad.weights <- summitOverlaps(olap, region.best=broad.best$rep.test)
tabbroad.w <- combineOverlaps(olap, results$table, o.weight=broad.weights)
```

6.4.3 Filtering after testing but before correction

Most of the filters in Chapter 3 are applied before the statistical analysis. However, some of the approaches may be too aggressive, e.g., filtering to retain only local maxima or based on pre-defined regions. In such cases, it may be preferable to initially apply one of the other, milder filters. This ensures that sufficient windows are retained for stable normalization and/or EB shrinkage. The aggressive filters can then be applied after the window-level statistics have been calculated, but before clustering into regions and calculation of cluster-level statistics. This is still beneficial as it removes irrelevant windows that would increase the severity of the BH correction. It may also reduce chaining effects during clustering.

6.5 FDR control in difficult situations

6.5.1 Clustering only on DB windows for diffuse marks

The clustering procedures described above rely on independent filtering to remove irrelevant windows. This ensures that the regions of interest are reasonably narrow and can be easily interpreted, which is typically the case for most protein targets, e.g., TFs, narrow histone

marks. However, enriched regions may be very large for more diffuse marks. Such regions may be difficult to interpret when only the DB subinterval is of interest. To overcome this, a *post-hoc* analysis can be performed whereby only significant windows are used for clustering.

```
postclust <- clusterWindows(rowRanges(filtered.data), results$table,
                            target=0.05, tol=100, max.width=1000)
postclust$FDR
```

```
## [1] 0.04985163
```

```
postclust$region
```

```
## GRanges object with 1685 ranges and 0 metadata columns:
##                       seqnames              ranges strand
##                          <Rle>           <IRanges>  <Rle>
##      [1]                 chr1   10037951-10037960      *
##      [2]                 chr1   15805551-15805610      *
##      [3]                 chr1   33565901-33566010      *
##      [4]                 chr1   35985451-35985510      *
##      [5]                 chr1   36684301-36684360      *
##      ...                  ...                 ...    ...
##   [1681]                 chrX 102157051-102157060      *
##   [1682]                 chrX 104482701-104482760      *
##   [1683]                 chrX 143483001-143483060      *
##   [1684] chrX_GL456233_random       336801-336910      *
##   [1685]                 chrY     4160201-4160310      *
##   -------
##   seqinfo: 66 sequences from an unspecified genome
```

This will define and cluster significant windows in a manner that controls the cluster-level FDR at 5%. The clustering step itself is performed using `mergeWindows` with the specified parameters. Each cluster consists entirely of DB windows and can be directly interpreted as a DB region or a DB subinterval of a larger enriched region. This reduces the pressure on abundance filtering to obtain well-separated regions prior to clustering, e.g., for diffuse marks or in data sets with weak IP signal. That said, users should be aware that calculation of the cluster-level FDR is not entirely rigorous. As such, independent clustering and FDR control via Simes' method should be considered as the default for routine analyses.

6.5.2 Using the empirical FDR for noisy data

Some analyses involve comparisons of ChIP samples to negative controls. In such cases, any region exhibiting enrichment in the negative control over the ChIP samples must be a false positive. The number of significant regions that change in the "wrong" direction can be used as an estimate of the number of false positives at any given p-value threshold. Division by the number of discoveries changing in the "right" direction yields an estimate of the FDR, i.e., the empirical FDR [13]. This strategy is implemented in the `empiricalFDR` function, which controls the empirical FDR across clusters based on their combined p-values. Its use is demonstrated below, though the output is not meaningful in this situation as genuine changes in binding can be present in both directions.

```
empres <- empiricalFDR(merged$id, results$table)
```

The empirical FDR is useful for analyses of noisy data with high levels of non-specific binding. This is because the estimate of the number of false positives adapts to the observed number of regions exhibiting enrichment in the negative controls. In contrast, the standard BH method in `combineTests` relies on proper type I error control during hypothesis testing. As non-specific binding events tend to be condition-specific, they are indistinguishable from DB events and assigned low p-values, resulting in loss of FDR control. Thus, for noisy data, use of the empirical FDR may be more appropriate to control the proportion of "experimental" false positives. However, calculation of the empirical FDR is not as statistically rigorous as that of the BH method, so users are advised to only apply it when necessary.

6.5.3 Detecting complex DB

Complex DB events involve changes to the shape of the binding profile, not just a scaling increase/decrease to binding intensity. Such regions may contain multiple sites that change in binding strength in opposite directions, or peaks that change in width or position between conditions. This often manifests as DB in opposite directions in different subintervals of a region. Some of these events can be identified using the `mixedTests` function.

```
tab.mixed <- mixedTests(merged$ids, results$table)
tab.mixed
```

```
## DataFrame with 4433 rows and 10 columns
##      num.tests num.up.logFC num.down.logFC    PValue       FDR   direction
##      <integer>    <integer>      <integer> <numeric> <numeric> <character>
## 1            1            1              0  0.988881         1       mixed
## 2            6            2              0  0.994660         1       mixed
## 3            3            2              0  0.993842         1       mixed
## 4            3            0              0  0.978055         1       mixed
## 5            1            1              0  0.992699         1       mixed
## ...        ...          ...            ...       ...       ...         ...
## 4429         1            0              1  0.994546         1       mixed
## 4430         1            0              0  0.940947         1       mixed
## 4431         2            0              2  0.999984         1       mixed
## 4432         2            0              0  0.942395         1       mixed
## 4433         6            0              0  0.892397         1       mixed
##      rep.up.test rep.up.logFC rep.down.test rep.down.logFC
##        <integer>    <numeric>     <integer>      <numeric>
## 1              1      1.82956             1        1.82956
## 2              4      1.09865             5        1.26642
## 3              9      1.51933            10        1.68157
## 4             12      1.28005            13        1.37830
## 5             14      1.92219            14        1.92219
## ...          ...          ...           ...            ...
## 4429       15697    -1.896735         15697      -1.896735
## 4430       15698    -1.142843         15698      -1.142843
## 4431       15700    -3.880663         15700      -3.880663
## 4432       15702     0.918049         15701       1.083159
## 4433       15707    -0.761103         15704      -0.196601
```

`mixedTests` converts the p-value for each window into two one-sided p-values. The one-sided p-values in each direction are combined using Simes' method, and the two one-sided combined p-values are themselves combined using an intersection-union test [28]. The resulting p-value is only low if a region contains strong DB in both directions.

`combineTests` also computes some statistics for informal detection of complex DB. For example, the `num.up.tests` and `num.down.tests` fields can be used to identify regions with changes in both directions. The `direction` field will also label some regions as `"mixed"`, though this is not comprehensive. Indeed, regions labelled as `"up"` or `"down"` in the 'direction' field may also correspond to complex DB events, but will not be labelled as `"mixed"` if the significance calculations are dominated by windows changing in only one direction.

6.5.4 Enforcing a minimal number of DB windows

On occasion, we may be interested in genomic regions that contain at least a minimal number or proportion of DB windows. This is motivated by the desire to avoid detecting DB regions where only a small subinterval exhibits a change, instead favoring more systematic changes throughout the region that are easier to interpret. We can identify these regions using the `minimalTests` function.

```
tab.min <- minimalTests(merged$ids, results$table,
    min.sig.n=3, min.sig.prop=0.5)
tab.min
```

```
## DataFrame with 4433 rows and 8 columns
##      num.tests num.up.logFC num.down.logFC      PValue       FDR   direction
##      <integer>    <integer>      <integer>   <numeric> <numeric> <character>
## 1            1            1              0   0.0222371 0.0978546          up
## 2            6            0              0   0.2900030 0.4877023          up
## 3            3            2              0   0.2185117 0.4085933          up
## 4            3            0              0   0.2052412 0.3911583          up
## 5            1            1              0   0.0146024 0.0748410          up
## ...        ...          ...            ...         ...       ...         ...
## 4429         1            0              1 0.010907924 0.0646455        down
## 4430         1            0              0 0.118105203 0.2747388        down
## 4431         2            0              2 0.000475973 0.0107120        down
## 4432         2            0              0 0.230418616 0.4244419          up
## 4433         6            0              0 1.000000000 1.0000000       mixed
##       rep.test rep.logFC
##      <integer> <numeric>
## 1            1  1.829565
## 2            3  0.923394
## 3            8  0.800016
## 4           11  0.988726
## 5           14  1.922191
## ...        ...       ...
## 4429     15697 -1.896735
## 4430     15698 -1.142843
## 4431     15699 -3.028362
## 4432     15702  0.918049
## 4433     15705  0.440353
```

`minimalTests` applies a Holm-Bonferroni correction to all windows in the same cluster and picks the x^{th}-smallest adjusted p-value (where x is defined from 'min.sig.n' and 'min.sig.prop'). This tests the joint null hypothesis that the per-window null hypothesis is false for fewer than x windows in the cluster. If the x^{th}-smallest p-value is low, this provides strong evidence against the joint null for that cluster.

As an aside, this function also has some utility outside of ChIP-seq contexts. For example, we might want to obtain a single p-value for a gene set based on the presence of a minimal percentage of differentially expressed genes. Alternatively, we may be interested in ranking genes in loss-of-function screens based on a minimal number of shRNA/CRISPR guides that exhibit a significant effect. These problems are equivalent to that of identifying a genomic region with a minimal number of DB windows.

6.6 Further points on data management

Technically, `results$table` was not required in any of the calls performed in this chapter. Recall that the per-window significance statistics were stored in the row metadata of `filtered.data`. Thus, `rowData(filtered.data)` could be used instead of `results$table`. The former may be more convenient as it avoids the need to keep track of a separate object.

On a similar note, it is possible to store the results for each region in the per-element metadata of the corresponding *GRanges* object. This synchronises the storage and handling of the statistics and coordinates for each region. Thus, data management throughout the rest of the analysis can be simplified. This is demonstrated below for `tabcom` and `tabbroad`.

```
mcols(broads) <- tabbroad
mcols(merged$region) <- tabcom
```

Chapter 7

Post-processing steps

This is where we bring it all together. We'll need the `merged` list from the previous chapter – oh, and the `org.Mm.eg.db` object that we loaded in Chapter 3. There's a bit about visualization at the end where we need the `data` and `param` objects from Chapter 2, along with the original `bam.files` that we started with in the introduction.

7.1 Adding gene-based annotation

Annotation can be added to a given set of regions using the `detailRanges` function. This will identify overlaps between the regions and annotated genomic features such as exons, introns and promoters. Here, the promoter region of each gene is defined as some interval 3 kbp upstream and 1 kbp downstream of the TSS for that gene. Any exonic features within `dist` on the left or right side of each supplied region will also be reported.

```
library(org.Mm.eg.db)
anno <- detailRanges(merged$region, txdb=TxDb.Mmusculus.UCSC.mm10.knownGene,
                     orgdb=org.Mm.eg.db, promoter=c(3000, 1000), dist=5000)
head(anno$overlap)
```

```
## [1] "Pcmtd1:+:PE"                   ""
## [3] ""                              "Rrs1:+:P,Adhfe1:+:P"
## [5] "Vcpip1:-:P,1700034P13Rik:+:PE" "Ppp1r42:-:I"
```

```
head(anno$left)
```

```
## [1] ""               ""               ""               ""
## [5] "Vcpip1:-:69"    "Ppp1r42:-:3948"
```

```
head(anno$right)
```

```
## [1] "Pcmtd1:+:144"           ""
## [3] "Rrs1:+:3898"            "Rrs1:+:48,Adhfe1:+:2588"
## [5] "1700034P13Rik:+:21"     "Ppp1r42:-:1612"
```

Character vectors of compact string representations are provided to summarize the features overlapped by each supplied region. Each pattern contains `GENE|STRAND|TYPE` to describe the strand and overlapped features of that gene. Exons are labelled as `E`, promoters are `P`

and introns are `I`. For `left` and `right`, `TYPE` is replaced by `DISTANCE`. This indicates the gap (in base pairs) between the supplied region and the closest non-overlapping exon of the annotated feature. All of this annotation can be stored in the metadata of the `GRanges` object for later use.

```
merged$region$overlap <- anno$overlap
merged$region$left <- anno$left
merged$region$right <- anno$right
```

While the string representation saves space in the output, it is not easy to work with. If the annotation needs to manipulated directly, users can obtain it from the `detailRanges` command by not specifying the regions of interest. This can then be used for interactive manipulation, e.g., to identify all genes where the promoter contains DB sites.

```
anno.ranges <- detailRanges(txdb=TxDb.Mmusculus.UCSC.mm10.knownGene,
                            orgdb=org.Mm.eg.db)
anno.ranges
```

```
## GRanges object with 505738 ranges and 2 metadata columns:
##             seqnames              ranges strand |      symbol        type
##                <Rle>           <IRanges>  <Rle> | <character> <character>
##   100009600     chr9   21062393-21062717      - |       Zglp1           E
##   100009600     chr9   21062400-21062717      - |       Zglp1           E
##   100009600     chr9   21062894-21062987      - |       Zglp1           E
##   100009600     chr9   21063314-21063396      - |       Zglp1           E
##   100009600     chr9   21066024-21066377      - |       Zglp1           E
##         ...      ...                 ...    ... .         ...         ...
##       99982     chr4 136554325-136558324      - |       Kdm1a           P
##       99982     chr4 136560502-136564501      - |       Kdm1a           P
##       99982     chr4 136567608-136571607      - |       Kdm1a           P
##       99982     chr4 136576159-136580158      - |       Kdm1a           P
##       99982     chr4 136550540-136602723      - |       Kdm1a           G
##   -------
##   seqinfo: 66 sequences (1 circular) from mm10 genome
```

7.2 Checking bimodality for TF studies

For TF experiments, a simple measure of strand bimodality can be reported as a diagnostic. Given a set of regions, the `checkBimodality` function will return the maximum bimodality score across all base positions in each region. The bimodality score at each base position is defined as the minimum of the ratio of the number of forward- to reverse-stranded reads to the left of that position, and the ratio of the reverse- to forward-stranded reads to the right. A high score is only possible if both ratios are large, i.e., strand bimodality is present.

```
spacing <- metadata(data)$spacing
expanded <- resize(merged$region, fix="center",
                   width=width(merged$region)+spacing)
sbm.score <- checkBimodality(bam.files, expanded, width=frag.len)
head(sbm.score)
```

```
## [1] 1.400000 1.455696 2.263158 1.363636 1.222222 5.333333
```

In the above code, all regions are expanded by `spacing`, i.e., 50 bp. This ensures that the optimal bimodality score can be computed for the centre of the binding site, even if that position is not captured by a window. The `width` argument specifies the span with which to count reads for the score calculation. This should be set to the average fragment length. If multiple `bam.files` are provided, they will be pooled during counting.

For typical TF binding sites, bimodality scores can be considered to be "high" if they are larger than 4. This allows users to distinguish between genuine binding sites and high-abundance artifacts such as repeats or read stacks. However, caution is still required as some high scores may be driven by the stochastic distribution of reads. Obviously, the concept of strand bimodality is less relevant for diffuse targets like histone marks.

7.3 Saving the results to file

It is a simple matter to save the results for later perusal. This is done here in the '`*.tsv`' format where all detail is preserved. Compression is used to reduce the file size.

```
ofile <- gzfile("clusters.tsv.gz", open="w")
write.table(as.data.frame(merged$region), file=ofile,
    row.names=FALSE, quote=FALSE, sep="\t")
close(ofile)
```

Of course, other formats can be used depending on the purpose of the file. For example, significantly DB regions can be exported to BED files through the *rtracklayer* package for visual inspection with genomic browsers. A transformed FDR is used here for the score field.

```
is.sig <- merged$region$FDR <= 0.05
library(rtracklayer)
test <- merged$region[is.sig]
test$score <- -10*log10(merged$region$FDR[is.sig])
names(test) <- paste0("region", 1:sum(is.sig))
export(test, "clusters.bed")
head(read.table("clusters.bed"))
```

```
##     V1       V2       V3      V4       V5 V6
## 1 chr1  9748450  9748460 region1 13.45862  .
## 2 chr1 10037950 10038010 region2 16.40853  .
## 3 chr1 15805500 15805660 region3 19.17791  .
## 4 chr1 23762950 23762960 region4 14.69600  .
## 5 chr1 33565900 33566010 region5 27.55389  .
## 6 chr1 35985450 35985510 region6 24.81004  .
```

Alternatively, the *GRanges* object can be directly saved to file and reloaded later for direct manipulation in the R environment, e.g., to find overlaps with other regions of interest.

```
saveRDS(merged$region, "ranges.rds")
```

7.4 Simple visualization of genomic coverage

Visualization of the read depth around interesting features is often desired. This is facilitated by the `extractReads` function, which pulls out the reads from the BAM file. The returned *GRanges* object can then be used to plot the sequencing coverage or any other statistic of interest. Note that the `extractReads` function also accepts a *readParam* object. This ensures that the same reads used in the analysis will be pulled out during visualization.

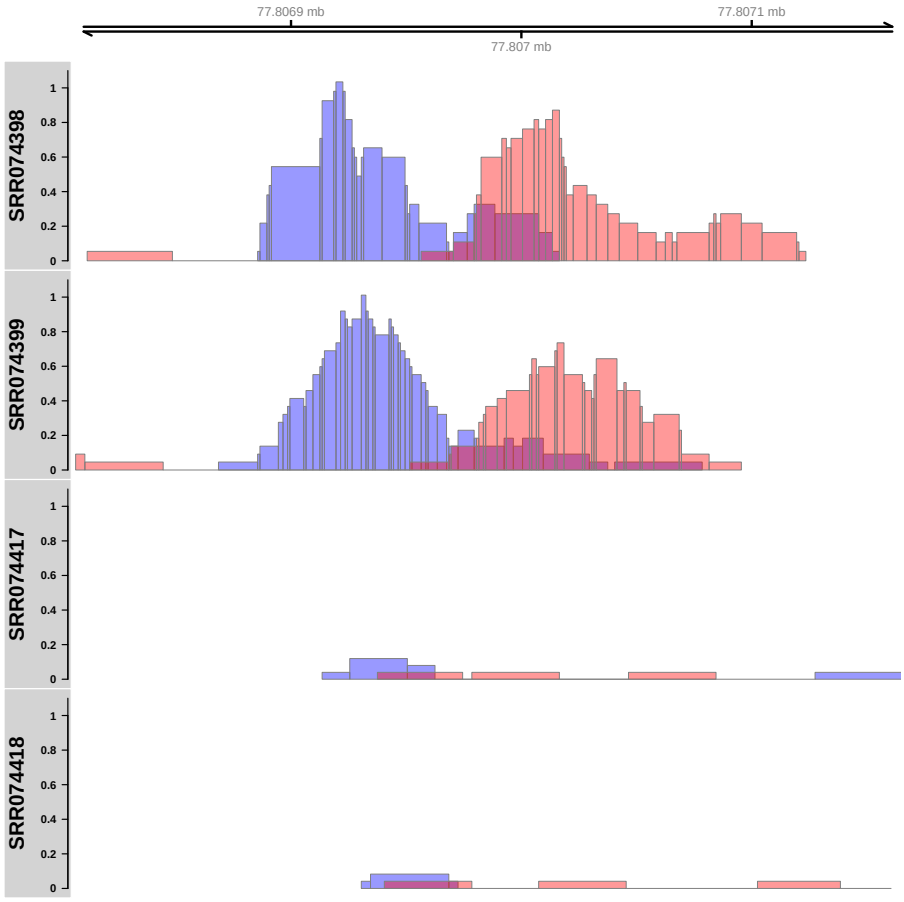
```
cur.region <- GRanges("chr18", IRanges(77806807, 77807165))
extractReads(bam.files[[1]], cur.region, param=param)
```

```
## GRanges object with 55 ranges and 0 metadata columns:
##        seqnames            ranges strand
##           <Rle>         <IRanges>  <Rle>
##    [1]    chr18 77806886-77806922      +
##    [2]    chr18 77806887-77806923      +
##    [3]    chr18 77806887-77806923      +
##    [4]    chr18 77806887-77806923      +
##    [5]    chr18 77806890-77806926      +
##    ...      ...               ...    ...
##   [51]    chr18 77807063-77807095      -
##   [52]    chr18 77807068-77807104      -
##   [53]    chr18 77807082-77807119      -
##   [54]    chr18 77807084-77807120      -
##   [55]    chr18 77807087-77807123      -
##   -------
##   seqinfo: 1 sequence from an unspecified genome
```

Here, coverage is visualized as the number of reads covering each base pair in the interval of interest. Specifically, the reads-per-million is shown to allow comparisons between libraries of different size. The plots themselves are constructed using methods from the *Gviz* package. The blue and red tracks represent the coverage on the forward and reverse strands, respectively. Strong strand bimodality is consistent with a genuine TF binding site. For paired-end data, coverage can be similarly plotted for fragments, i.e., proper read pairs.

```
library(Gviz)
collected <- vector("list", length(bam.files))
for (i in seq_along(bam.files)) {
    reads <- extractReads(bam.files[[i]], cur.region, param=param)
    adj.total <- data$totals[i]/1e6
    pcov <- as(coverage(reads[strand(reads)=="+"])/adj.total, "GRanges")
    ncov <- as(coverage(reads[strand(reads)=="-"])/adj.total, "GRanges")
    ptrack <- DataTrack(pcov, type="histogram", lwd=0, fill=rgb(0,0,1,.4),
                        ylim=c(0,1.1), name=tf.data$Name[i], col.axis="black",
                        col.title="black")
    ntrack <- DataTrack(ncov, type="histogram", lwd=0, fill=rgb(1,0,0,.4),
                        ylim=c(0,1.1))
```

```
        collected[[i]] <- OverlayTrack(trackList=list(ptrack,ntrack))
    }
    gax <- GenomeAxisTrack(col="black")
    plotTracks(c(gax, collected), from=start(cur.region), to=end(cur.region))
```



Chapter 8

Epilogue

Congratulations on getting to the end. Here's a poem for your efforts.

There once was a man named Will
Who never ate less than his fill.
He ate meat and bread
Until he was fed
But died when he saw the bill.

8.1 Session information

```
sessionInfo()
```

```
## R version 4.0.3 (2020-10-10)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 18.04.5 LTS
##
## Matrix products: default
## BLAS:   /home/biocbuild/bbs-3.12-bioc/R/lib/libRblas.so
## LAPACK: /home/biocbuild/bbs-3.12-bioc/R/lib/libRlapack.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
##  [1] grid      parallel  stats4    stats     graphics  grDevices utils
##  [8] datasets  methods   base
##
## other attached packages:
##  [1] Gviz_1.34.0
```

```
##  [2] rtracklayer_1.50.0
##  [3] org.Mm.eg.db_3.12.0
##  [4] TxDb.Mmusculus.UCSC.mm10.knownGene_3.10.0
##  [5] GenomicFeatures_1.42.0
##  [6] AnnotationDbi_1.52.0
##  [7] edgeR_3.32.0
##  [8] limma_3.46.0
##  [9] csaw_1.24.0
## [10] SummarizedExperiment_1.20.0
## [11] Biobase_2.50.0
## [12] MatrixGenerics_1.2.0
## [13] matrixStats_0.57.0
## [14] GenomicRanges_1.42.0
## [15] GenomeInfoDb_1.26.0
## [16] IRanges_2.24.0
## [17] S4Vectors_0.28.0
## [18] BiocGenerics_0.36.0
## [19] chipseqDBData_1.6.0
##
## loaded via a namespace (and not attached):
##   [1] colorspace_1.4-1             ellipsis_0.3.1
##   [3] htmlTable_2.1.0              biovizBase_1.38.0
##   [5] XVector_0.30.0               base64enc_0.1-3
##   [7] dichromat_2.0-0              rstudioapi_0.11
##   [9] bit64_4.0.5                  interactiveDisplayBase_1.28.0
##  [11] xml2_1.3.2                   codetools_0.2-16
##  [13] splines_4.0.3                knitr_1.30
##  [15] Formula_1.2-4                Rsamtools_2.6.0
##  [17] cluster_2.1.0                dbplyr_1.4.4
##  [19] png_0.1-7                    shiny_1.5.0
##  [21] BiocManager_1.30.10          compiler_4.0.3
##  [23] httr_1.4.2                   backports_1.1.10
##  [25] lazyeval_0.2.2               assertthat_0.2.1
##  [27] Matrix_1.2-18                fastmap_1.0.1
##  [29] later_1.1.0.1                htmltools_0.5.0
##  [31] prettyunits_1.1.1            tools_4.0.3
##  [33] gtable_0.3.0                 glue_1.4.2
##  [35] GenomeInfoDbData_1.2.4       dplyr_1.0.2
##  [37] rappdirs_0.3.1               Rcpp_1.0.5
##  [39] vctrs_0.3.4                  Biostrings_2.58.0
##  [41] ExperimentHub_1.16.0         xfun_0.18
##  [43] stringr_1.4.0                mime_0.9
##  [45] lifecycle_0.2.0              ensembldb_2.14.0
##  [47] statmod_1.4.35               XML_3.99-0.5
##  [49] AnnotationHub_2.22.0         zlibbioc_1.36.0
##  [51] scales_1.1.1                 BiocStyle_2.18.0
##  [53] BSgenome_1.58.0              VariantAnnotation_1.36.0
##  [55] ProtGenerics_1.22.0          hms_0.5.3
##  [57] promises_1.1.1               AnnotationFilter_1.14.0
##  [59] RColorBrewer_1.1-2           yaml_2.2.1
##  [61] curl_4.3                     gridExtra_2.3
```

```
##  [63] memoise_1.1.0                ggplot2_3.3.2
##  [65] rpart_4.1-15                 biomaRt_2.46.0
##  [67] latticeExtra_0.6-29          stringi_1.5.3
##  [69] RSQLite_2.2.1                BiocVersion_3.12.0
##  [71] highr_0.8                    checkmate_2.0.0
##  [73] BiocParallel_1.24.0          rlang_0.4.8
##  [75] pkgconfig_2.0.3              bitops_1.0-6
##  [77] evaluate_0.14                lattice_0.20-41
##  [79] purrr_0.3.4                  htmlwidgets_1.5.2
##  [81] GenomicAlignments_1.26.0     bit_4.0.4
##  [83] tidyselect_1.1.0             magrittr_1.5
##  [85] R6_2.5.0                     generics_0.0.2
##  [87] Hmisc_4.4-1                  DelayedArray_0.16.0
##  [89] DBI_1.1.0                    pillar_1.4.6
##  [91] foreign_0.8-80               nnet_7.3-14
##  [93] survival_3.2-7               RCurl_1.98-1.2
##  [95] tibble_3.0.4                 crayon_1.3.4
##  [97] KernSmooth_2.23-18           BiocFileCache_1.14.0
##  [99] rmarkdown_2.5                jpeg_0.1-8.1
## [101] progress_1.2.2               locfit_1.5-9.4
## [103] data.table_1.13.2            blob_1.2.1
## [105] digest_0.6.27                xtable_1.8-4
## [107] httpuv_1.5.4                 openssl_1.4.3
## [109] munsell_0.5.0                askpass_1.1
```

Bibliography

[1] A. T. Lun and G. K. Smyth. csaw: a Bioconductor package for differential binding analysis of ChIP-seq data using sliding windows. *Nucleic Acids Res.*, 44(5):e45, Mar 2016.

[2] A. T. L. Lun and G. K. Smyth. From reads to regions: a Bioconductor workflow to detect differential binding in ChIP-seq data. *F1000Research*, 4, 2015.

[3] M. D. Robinson, D. J. McCarthy, and G. K. Smyth. edgeR: a Bioconductor package for differential expression analysis of digital gene expression data. *Bioinformatics*, 26(1):139–140, Jan 2010.

[4] A. T. Lun and G. K. Smyth. De novo detection of differentially bound regions for ChIP-seq data using peaks and windows: controlling error rates correctly. *Nucleic Acids Res.*, 42(11):e95, Jul 2014.

[5] 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.

[6] V. K. Tiwari, M. B. Stadler, C. Wirbelauer, R. Paro, D. Schubeler, and C. Beisel. A chromatin-modifying function of JNK during stem cell differentiation. *Nat. Genet.*, 44(1):94–100, Jan 2012.

[7] 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.

[8] ENCODE Project Consortium. An integrated encyclopedia of DNA elements in the human genome. *Nature*, 489(7414):57–74, Sep 2012.

[9] P. V. Kharchenko, M. Y. Tolstorukov, and P. J. Park. Design and analysis of ChIP-seq experiments for DNA-binding proteins. *Nat. Biotechnol.*, 26(12):1351–1359, Dec 2008.

[10] S. G. Landt, G. K. Marinov, A. Kundaje, P. Kheradpour, F. Pauli, S. Batzoglou, B. E. Bernstein, P. Bickel, J. B. Brown, P. Cayting, Y. Chen, G. Desalvo, C. Epstein, K. I. Fisher-Aylor, G. Euskirchen, M. Gerstein, J. Gertz, A. J. Hartemink, M. M. Hoffman, V. R. Iyer, Y. L. Jung, S. Karmakar, M. Kellis, P. V. Kharchenko, Q. Li, T. Liu, X. S. Liu, L. Ma, A. Milosavljevic, R. M. Myers, P. J. Park, M. J. Pazin, M. D. Perry, D. Raha, T. E. Reddy, J. Rozowsky, N. Shoresh, A. Sidow, M. Slattery, J. A. Stamatoyannopoulos, M. Y. Tolstorukov, K. P. White, S. Xi, P. J. Farnham, J. D. Lieb, B. J. Wold, and M. Snyder. ChIP-seq guidelines and practices of the ENCODE and modENCODE consortia. *Genome Res.*, 22(9):1813–1831, Sep 2012.

[11] P. Humburg, C. A. Helliwell, D. Bulger, and G. Stone. ChIPseqR: analysis of ChIP-seq experiments. *BMC Bioinformatics*, 12:39, 2011.

[12] R. Bourgon, R. Gentleman, and W. Huber. Independent filtering increases detection power for high-throughput experiments. *Proc. Natl. Acad. Sci. U.S.A.*, 107(21):9546–9551, May 2010.

[13] Y. Zhang, T. Liu, C. A. Meyer, J. Eeckhoute, D. S. Johnson, B. E. Bernstein, C. Nusbaum, R. M. Myers, M. Brown, W. Li, and X. S. Liu. Model-based analysis of ChIP-Seq (MACS). *Genome Biol.*, 9(9):R137, 2008.

[14] M. D. Robinson and A. Oshlack. A scaling normalization method for differential expression analysis of RNA-seq data. *Genome Biol.*, 11(3):R25, 2010.

[15] N. Bonhoure, G. Bounova, D. Bernasconi, V. Praz, F. Lammers, D. Canella, I. M. Willis, W. Herr, N. Hernandez, M. Delorenzi, N. Hernandez, M. Delorenzi, B. Deplancke, B. Desvergne, N. Guex, W. Herr, F. Naef, J. Rougemont, U. Schibler, T. Andersin, P. Cousin, F. Gilardi, P. Gos, F. Lammers, S. Raghav, D. Villeneuve, R. Fabbretti, V. Vlegel, I. Xenarios, E. Migliavacca, V. Praz, F. David, Y. Jarosz, D. Kuznetsov, R. Liechti, O. Martin, J. Delafontaine, J. Cajan, K. Gustafson, I. Krier, M. Leleu, N. Molina, A. Naldi, L. Rib, L. Symul, and G. Bounova. Quantifying ChIP-seq data: a spiking method providing an internal reference for sample-to-sample normalization. *Genome Res.*, 24(7):1157–1168, Jul 2014.

[16] D. A. Orlando, M. W. Chen, V. E. Brown, S. Solanki, Y. J. Choi, E. R. Olson, C. C. Fritz, J. E. Bradner, and M. G. Guenther. Quantitative ChIP-Seq normalization reveals global modulation of the epigenome. *Cell Rep.*, 9(3):1163–1170, Nov 2014.

[17] K. V. Ballman, D. E. Grill, A. L. Oberg, and T. M. Therneau. Faster cyclic loess: normalizing RNA arrays via linear models. *Bioinformatics*, 20(16):2778–2786, Nov 2004.

[18] S. P. Lund, D. Nettleton, D. J. McCarthy, and G. K. Smyth. Detecting differential expression in RNA-sequence data using quasi-likelihood with shrunken dispersion estimates. *Stat. Appl. Genet. Mol. Biol.*, 11(5), 2012.

[19] M. D. Robinson and G. K. Smyth. Small-sample estimation of negative binomial dispersion, with applications to SAGE data. *Biostatistics*, 9(2):321–332, Apr 2008.

[20] C. W. Law, Y. Chen, W. Shi, and G. K. Smyth. Voom: precision weights unlock linear model analysis tools for RNA-seq read counts. *Genome Biol.*, 15(2):R29, Feb 2014.

[21] Y. Chen, A. T. L. Lun, and G. K. Smyth. Differential expression analysis of complex RNA-seq experiments using edgeR. In S. Datta and D. S. Nettleton, editors, *Statistical Analysis of Next Generation Sequence Data*. Springer, New York, 2014.

[22] D. J. McCarthy, Y. Chen, and G. K. Smyth. Differential expression analysis of multifactor RNA-Seq experiments with respect to biological variation. *Nucleic Acids Res.*, 40(10):4288–4297, May 2012.

[23] B. Phipson, S. Lee, I. J. Majewski, W. S. Alexander, and G. K. Smyth. Robust hyperparameter estimation protects against hypervariable genes and improves power to detect differential expression. *Ann. Appl. Stat.*, 10(2):946–963, 2016.

[24] G. K. Smyth. Linear models and empirical bayes methods for assessing differential expression in microarray experiments. *Stat. Appl. Genet. Mol. Biol.*, 3:Article 3, 2004.

[25] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *J. R. Stat. Soc. Series B*, 57:289–300, 1995.

[26] R. J. Simes. An improved Bonferroni procedure for multiple tests of significance. *Biometrika*, 73(3):751–754, 1986.

[27] Y. Benjamini and Y. Hochberg. Multiple hypotheses testing with weights. *Scand. J. Stat.*, 24:407–418, 1997.

[28] R. L. Berger and J. C. Hsu. Bioequivalence trials, intersection-union tests and equivalence confidence sets. *Statist. Sci.*, 11(4):283–319, 11 1996.