csaw: ChIP-seq analysis with windows

User's Guide

Aaron Lun

First edition 15 August 2012

Last revised 16 May 2015

Contents

Chapter 1

Introduction

1.1 Scope

This document gives an overview of the Bioconductor package csaw for detecting differential binding (DB) in ChIP-seq experiments. Specifically, csaw uses sliding windows to identify significant changes in binding patterns for transcription factors (TFs) or histone marks across different biological conditions [Lun and Smyth, 2014]. However, it can also be applied to any sequencing technique where reads represent coverage of enriched genomic regions. The statistical methods described here are based upon those in the edgeR package [Robinson et al., 2010]. 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 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.

```
> require(csaw)
> param <- readParam(minq=50)
> data <- windowCounts(bam.files, ext=110, width=10, param=param)
```

2. Filtering out uninteresting regions.

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

3. Calculating normalization factors.

```
> binned <- windowCounts(bam.files, bin=TRUE, width=10000, param=param)
> normfacs <- normalize(binned)
```

4. Identifying DB windows.

```
> y <- asDGEList(data, norm.factors=normfacs)
> y <- estimateDisp(y, design)
> fit <- glmQLFit(y, design, robust=TRUE)
> results <- glmQLFTest(fit)
```

5. Correcting for multiple testing.

```
> merged <- mergeWindows(rowRanges(data), tol=1000L)
> tabcom <- combineTests(merged$id, results$table)
```

In this guide, the behaviour of each step will be demonstrated with some publicly available data. The dataset below focuses on changes in the binding profile of the NFYA protein between embryonic stem cells and terminal neurons [Tiwari et al., 2012]. This will be used as a case study for most of the code examples throughout the guide.

```
> bam.files <- c("es_1.bam", "es_2.bam", "tn_1.bam", "tn_2.bam")
> design <- model.matrix(~factor(c('es', 'es', 'tn', 'tn')))
> colnames(design) <- c("intercept", "cell.type")
```

A comprehensive listing of the datasets used in this guide is provided in Section 8.1, along with instructions on how to obtain and process them for entry into the csaw pipeline.

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. At this point, all we need are the `bam.files` that we defined in the introduction above.

2.1 Types of input data

Sorted and indexed BAM (i.e., binary SAM) files are required as input into the read counting functions in csaw. Sorting should be performed on the genomic position of the mapped read. For a given BAM file named `xxx.bam`, the corresponding index file should be named as `xxx.bam.bai` such that both files are in the same directory. Users should be aware that 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 new BAM file.

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. The number of fragments overlapping a genomic window is counted. 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.

```
> frag.len <- 110
> win.width <- 10
> param <- readParam(minq=50)
> 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`. The total number of reads in each library is stored as `totals` in the `colData`.

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

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

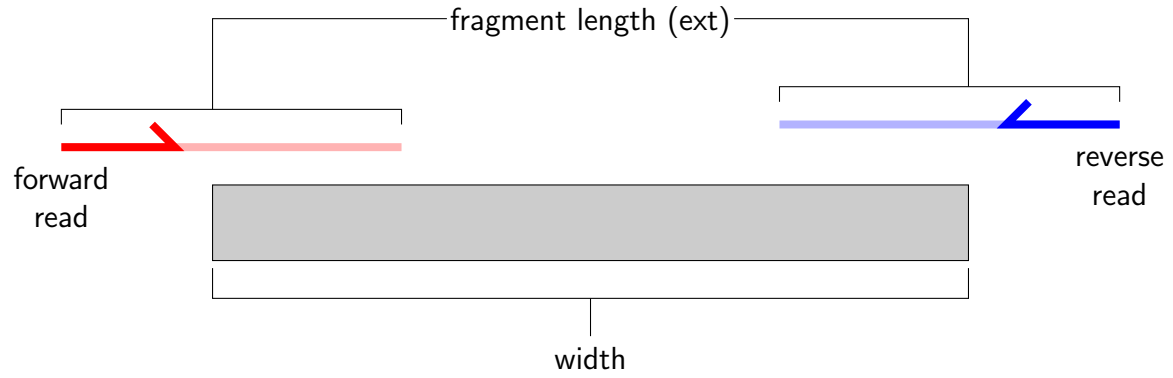
```
> head(rowRanges(data))
```

```
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 [3004051, 3004060]      *
  [5]     chr1 [3007551, 3007560]      *
  [6]     chr1 [3007651, 3007660]      *
  -------
  seqinfo: 66 sequences from an unspecified genome
```

```
> data$totals
```

```
[1] 17196528 20040530 23118590 22075821
```

For single-end data, suitable values for the average fragment length in `ext` can be estimated from the primary peak in a cross-correlation plot (see Section 2.4). 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 specified `width` of each window controls the compromise between spatial resolution and count size. Larger windows will yield higher read counts which 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.

The window size can be interpreted as a measure of the width of the binding site. Thus, TF analyses will typically use a small window size, e.g., 10 - 20 bp. This maximizes spatial resolution to allow optimal detection of narrow regions of enrichment. For histone marks, widths of at least 150 bp are recommended [Humburg et al., 2011]. This corresponds to the length of DNA wrapped up in each nucleosome, i.e., the smallest relevant unit for histone mark enrichment. For diffuse marks, the sizes of enriched regions are more variable and so the compromise between resolution and power is more arbitrary. Analyses with multiple widths can be combined to examine DB across several resolutions (see Section 6.5).

For TF analyses with small windows, the choice of spacing interval will also have an effect on 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 which specifies a number of extraction parameters. The idea is to define the `readParam` object once in the pipeline. It can then be reused for all relevant functions, to ensure 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 50
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 [Liao et al., 2013] was used to align the reads in this dataset, so a value of 50 might be appropriate.

Reads marked as PCR duplicates in the SAM flag can be ignored by setting `dedup=TRUE`. This can reduce the variability caused by inconsistent amplification between replicates, and avoid spurious duplicate-driven DB between groups. However, it also caps the number of reads at each position. This can lead to loss of 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. Thus, duplicate removal is not recommended for routine DB analyses. Of course, it may be unavoidable in some cases, e.g., involving libraries generated from low quantities of DNA. An example of counting with duplicate removal is shown below. Comparison to `data$totals` indicates that fewer reads are used from each library.

```
> dedup.param <- readParam(minq=50, dedup=TRUE)
> demo <- windowCounts(bam.files, ext=frag.len, width=win.width, param=dedup.param)
> demo$totals

[1] 13925800 13826445 16342981 18905912
```

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 [Consortium, 2012], and can be obtained by following this **link**. 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., for `mouse`) or they can be extracted from an appropriate masked `BSgenome` object.

Using `discard` is safer than simply ignoring windows that overlap the repeats. For example, a large window might contain both repeat regions 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
```

The `windowCounts` function will also accept a list of `readParam` objects. Different read extraction parameters can then be specified for each library. Use of library-specific settings is dangerous as spurious differences can be introduced between libraries. Nonetheless, it may be unavoidable, e.g., if the dataset is composed of a mixture of single- and paired-end libraries. A hypothetical example is shown below with one single- and one paired-end library. To mimic single-end data in the second library, only the first read of the pair is used.

```
> paramlist <- list(readParam(), readParam(pe="first"))
> checkList(paramlist)

[1] "pe"
```

For convenience, the `checkList` function is provided to identify those parameters that are different across the list. This can be useful to ensure that only the intended parameters are varied across libraries. Lists can also be manipulated with the `reformList` function, which sets parameters to their specified values in all elements of the list. In the code below, coercion of `pe` to a common value removes any differences between `readParam` objects.

```
> paramlist <- reformList(paramlist, pe="none")
> checkList(paramlist)
```

```
character(0)
```

Users are encouraged to construct their own `readParam` objects (or lists thereof) 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 [3008301, 3008310]      *
  [4]     chr1 [3008401, 3008410]      *
  [5]     chr1 [3009201, 3009210]      *
  [6]     chr1 [3010901, 3010910]      *
  -------
  seqinfo: 66 sequences from an unspecified genome
```

For analyses with large windows, it is worth 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 negligble when 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 can be filtered out using the `filter` argument. This improves memory efficiency by discarding the majority of low-abundance windows corresponding to uninteresting background regions. Specifically, any window is removed if the sum of counts across all libraries is below `filter`. 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,]    5   14    2    9
[2,]    5    2   19    6
[3,]    4    6   17    4
[4,]    6    8   16    5
[5,]    3   11   16    5
[6,]    5   11   13    2
```

Setting `bin=TRUE` will cause `windowCounts` to count reads into contiguous bins across the genome. Specifically, `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. No filtering is performed on the count sum, such that counts will be returned for each genomic bin. As a result, users should set `width` to a reasonably large value. This avoids problems with memory constraints when counts are returned for a large number of small bins.

```
> 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
```

2.3 Experiments involving paired-end data

2.3.1 Counting with proper pairs

ChIP experiments with paired-end sequencing can be accomodated by setting `pe="both"` in the `param` object supplied to `windowCounts`. Read extension is not required as the genomic

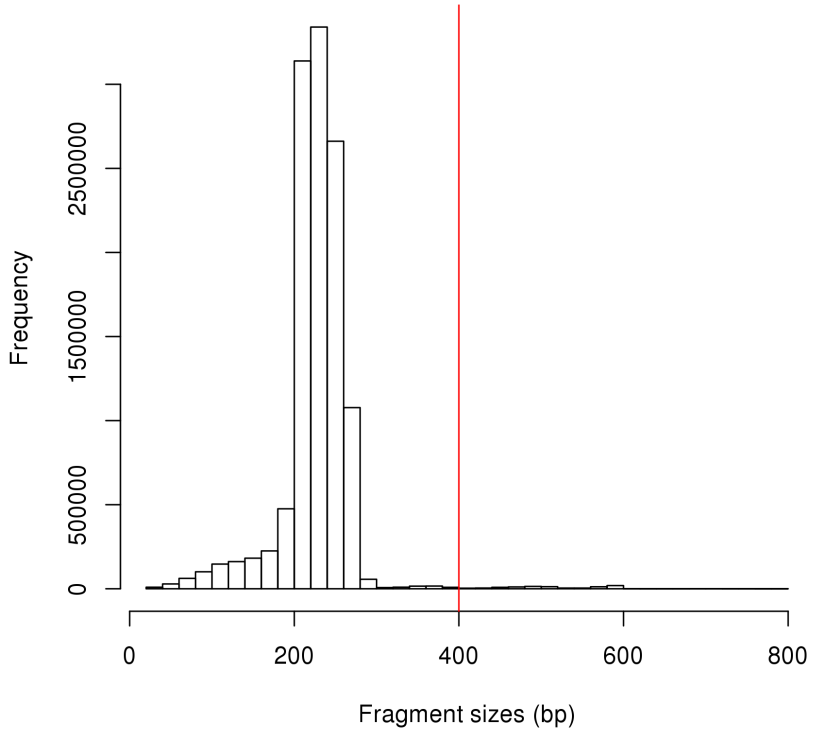
interval spanned by the originating fragment is explicitly defined between the 5′ positions of the paired reads. The number of fragments overlapping each window is then counted as described. By default, only proper pairs are used whereby the two reads are on the same chromosome, face inward and are no more than `max.frag` apart.

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

```
[1] 11723681
```

A suitable value for `max.frag` can be chosen by examining the distribution of fragment sizes from the `getPESizes` function. In this example, a user might use a value of around 400 bp as it exceeds the vast majority of the fragment size distribution. 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="")
> abline(v=400, col="red")
```



The number of fragments exceeding the maximum size can be 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
     33498954      27420853             0       2000261         94786
    inter.chr     too.large
        761432        130397
```

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

Observant readers may notice that `ext` was specified in the above call to `windowCounts`. This is not strictly necessary for paired-end data, where the fragment length is explicitly calculated for each proper pair. However, it is recommended to set `ext` to the median fragment length across all pairs in each library - in this case, around 250 bp. This ensures consistency with some downstream functions, which assume that the `ext` field of the returned `RangedSummarizedExperiment` holds the average fragment length. Specification of `ext` is also necessary if fragments are to be coerced to the same length, as described in Section 2.4.2.

2.3.2 Handling low-quality paired-end data

In datasets where many read pairs are invalid, the reads in those pairs can be rescued by setting `rescue.ext` to a positive integer. For each invalid intra-chromosomal read pair, the read with the higher mapping quality score will be directionally extended by `rescue.ext` to impute the fragment. The other read in the pair is ignored. For inter-chromosomal read pairs, both reads are extended in this manner. Counting will then be performed with these fragments in addition to those from the valid pairs. An appropriate value of `rescue.ext` can be chosen based on the median of the fragment size distribution, above.

```
> rescue.param <- reform(pe.param, rescue.ext=250)
> demo <- windowCounts(pe.bam, ext=250, param=rescue.param)
> demo$totals
```

```
[1] 15469055
```

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] 13715351
```

2.3.3 Speeding up extraction for paired-end reads

Paired-end processing in csaw tends to be slow, as read names must be loaded and matched to identify proper pairs. This can be accelerated by performing the matching step once, and storing the identified pairs in a separate BAM file. Downstream applications can then directly use pairs in the new file without name matching. This pre-processing is performed by the `dumpPE` function, to produce a sorted/indexed BAM file with the specified `prefix`.

```
> new.bam <- dumpPE("example-pet.bam", prefix="new-pet", param=pe.param)
```

All read extraction functions can set `fast.pe=TRUE` in `readParam` to extract reads quickly from the new BAM file. This will identify each fragment based on the mapping position of the first read in each pair and the associated insert size. However, almost all of the other parameters in `readParam` will be ignored, e.g., `dedup`, `minq`, `discard`, `rescue.ext`. Any that are required for the analysis should be applied during `dumpPE`.

```
> fast.param <- reform(pe.param, fast.pe=TRUE)
> demo <- windowCounts(new.bam, ext=250, param=fast.param)
```

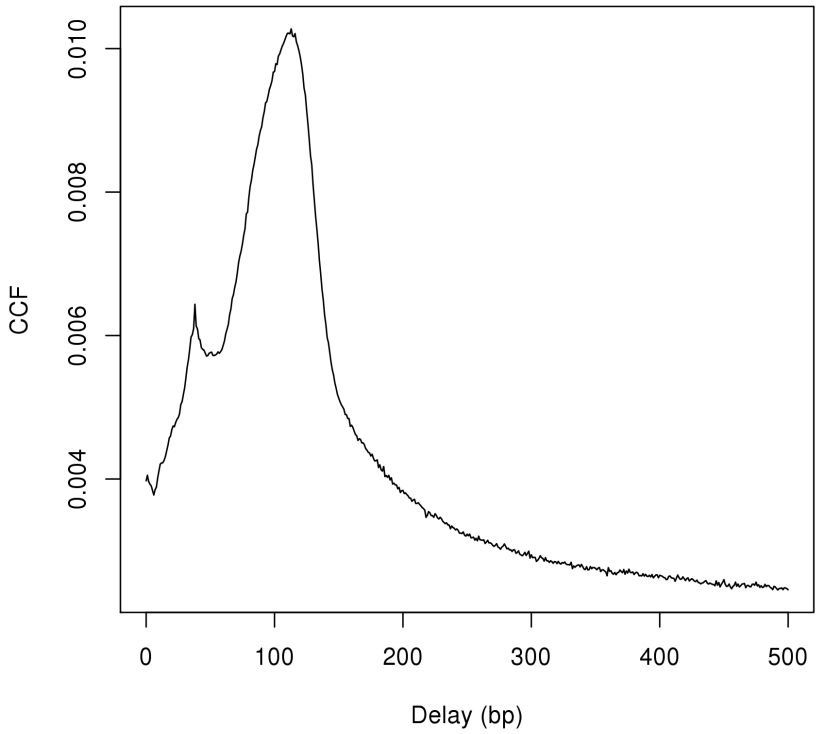
It is also possible to run `fast.pe=TRUE` on BAM files without first processing them with `dumpPE`. This should yield similar results as `fast.pe=FALSE` if none of the quality control parameters have been set.

2.4 Estimating the average fragment length

2.4.1 Using cross-correlation plots

Cross-correlation plots can be generated directly from BAM files using the `correlateReads` function. This provides a measure of the immunoprecipitation (IP) efficiency of a ChIP-seq experiment [Kharchenko et al., 2008]. 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. The location of the peak can then be used as an estimate of the fragment length for read extension in `windowCounts`. For this dataset, an estimate of $\sim$110 bp is obtained from the plot below.

```
> 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)")
```

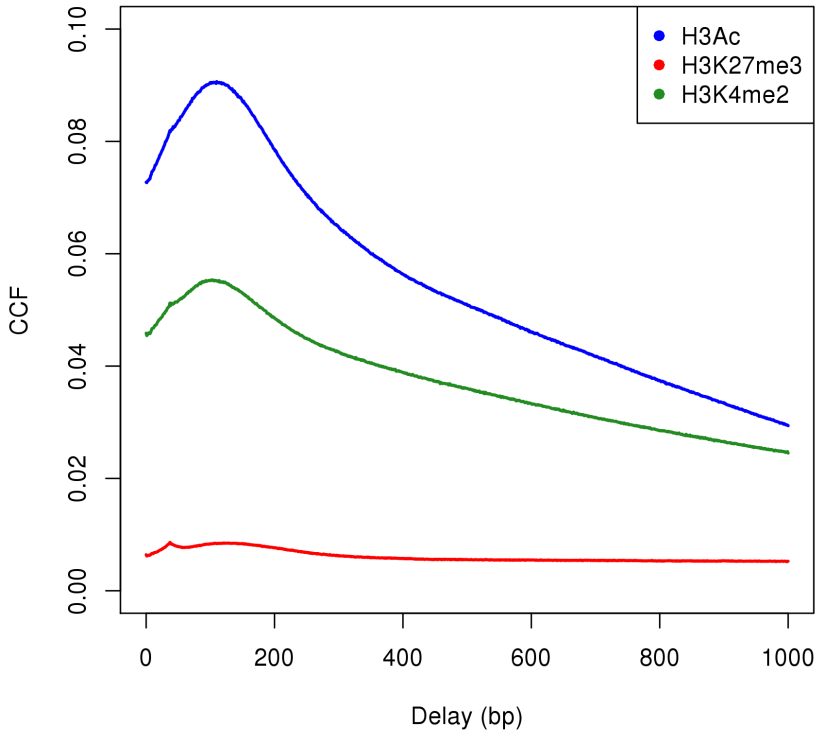


A sharp spike may also observed in the plot at a distance corresponding to the read length. This is thought to be an artifact, caused by the preference of aligners towards uniquely mapped reads. Duplicate removal is typically required here (i.e., set `dedup=TRUE` in `readParam`) to reduce the size of 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 [Landt et al., 2012]. 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
> h3ac <- correlateReads("h3ac.bam", n, param=dedup.on)
```

```
> h3k27me3 <- correlateReads("h3k27me3.bam", n, param=dedup.on)
> h3k4me2 <- correlateReads("h3k4me2.bam", n, param=dedup.on)
> plot(0:n, h3ac, 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, h3k4me2, col="forestgreen", pch=16, lwd=2)
> legend("topright", col=c("blue", "red", "forestgreen"),
+     c("H3Ac", "H3K27me3", "H3K4me2"), 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 larger fragment lengths will have wider peaks. The single-end reads in those peaks will then require more extension, in order to impute a fragment interval that covers the binding site. This is done by supplying a vector to the `ext` argument. Each entry specifies the average fragment length to be used for the corresponding library. The extension lengths are also stored in the `RangedSummarizedExperiment` output for future reference.

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

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

Caution is required to avoid detecting irrelevant DB from differences in peak widths. Some protection is provided in `windowCounts`, by scaling extended reads to the same length in all libraries. Consider a bimodal peak across several libraries. Scaling 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. This removes most of the differences in width between libraries. The final fragment length is taken from the `final.ext` attribute of the `ext` argument. This can be set with `makeExtVector`, and is defined as the mean of the fragment lengths by default. If `final.ext` is not present or is `NA`, no rescaling is performed.

```
> scaled.frag.lens <- makeExtVector(multi.frag.lens)
> attributes(scaled.frag.lens)$final.ext
```

```
[1] 175
```

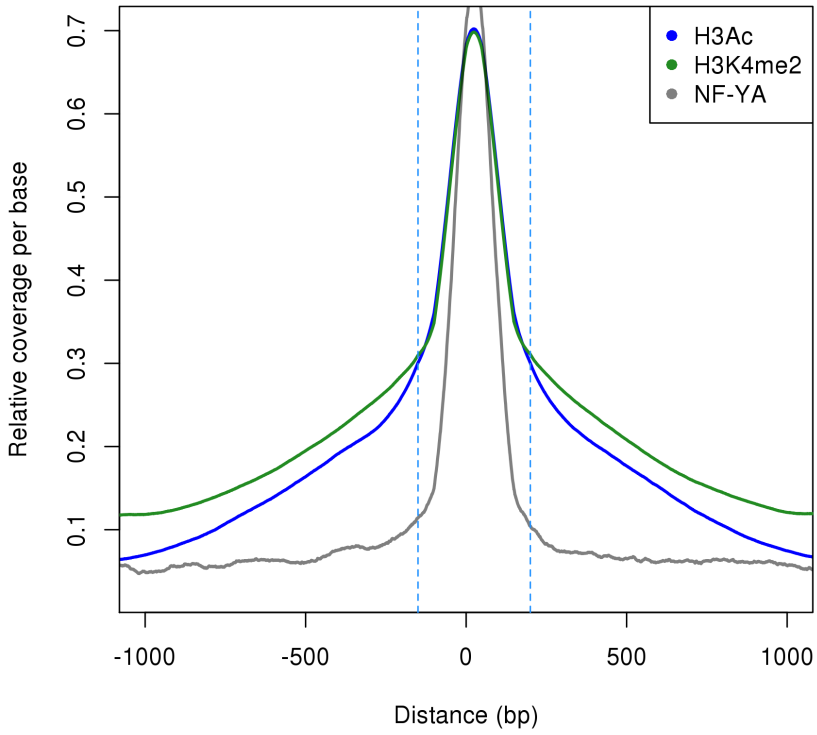
In general, use of different extension lengths is unnecessary in well-controlled datasets. Difference in lengths between libraries are usually smaller than 50 bp. This is less than the inherent variability in fragment lengths within each library (see the histogram for the paired-end data in Section 2.3). Such variability will affect the read coverage profile more than any difference in lengths, and is likely to mask the latter. Thus, an `ext` vector should only specified for datasets that exhibit large differences in the fragment sizes.

2.5 Choosing an appropriate window size

The coverage profile around potential binding sites can be obtained with the `profileSites` function. Here, the binding sites are defined by taking high-abundance 50 bp windows and identifying those that are locally maximal using `findMaxima`. For each selected window, `profileSites` records the coverage across the flanking regions as a function of the distance from the edge of the window. This is divided by the count for the window itself to obtain a relative coverage, based on the specification of `weight`. The values are then averaged across all windows to obtain an aggregated coverage profile for each library.

```
> all.bam <- c("h3ac.bam", "h3k4me2.bam", "es_1.bam")
> collected <- list()
> for (curbam in all.bam) {
+     windowed <- windowCounts(curbam, spacing=50, width=50, param=dedup.on, filter=20)
+     rwsms <- rowSums(assay(windowed))
+     maxed <- findMaxima(rowRanges(windowed), range=1000, metric=rwsms)
+     collected[[curbam]] <- profileSites(curbam, rowRanges(windowed)[maxed],
```

```
+         param=dedup.on, weight=1/rwsms[maxed])
+ }
> xranged <- as.integer(names(collected[[1]]))
> plot(xranged, collected[[1]], type="l", col="blue", xlim=c(-1000, 1000), lwd=2,
+     xlab="Distance (bp)", ylab="Relative coverage per base")
> lines(xranged, collected[[2]], col="forestgreen", lwd=2)
> lines(xranged, collected[[3]], col=rgb(0,0,0,0.5), lwd=2)
> legend("topright", col=c("blue", "forestgreen", rgb(0,0,0,0.5)),
+     c("H3Ac", "H3K4me2", "NF-YA"), pch=16)
> abline(v=c(-150,200), col="dodgerblue", lty=2)
```



In the example above, enrichment for the two histone marks is mostly contained within a 350 bp region around the maxima (dashed lines). This suggests that a window size of ∼150 bp is ideal, given that directional extension by 100 bp has been performed on both sides of the peak. Most of the coverage can then be captured without including too much background noise. In contrast, the NF-YA profile drops off more sharply. This indicates that a smaller window size ($<$ 50 bp) is probably adequate, consistent with sharp TF binding. See the `wwhm` function for selection of a window width from the coverage profile.

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. This manifests as a long tail in the coverage profile plot and suggests 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. Thus, it may be preferable to err on the side of smaller windows to maintain spatial resolution for such events.

The performance of this approach also deteriorates when background enrichment is prevalent. This makes it difficult to determine when the coverage becomes negligble. The problem is compounded for diffuse marks where the transition between enrichment and background is unclear. Indeed, local maxima are unlikely to be well-defined within diffuse regions. Thus, users are advised to exercise caution when picking a window size from these plots.

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,]   37   57  103  102
[2,]    0    0    1    0
[3,]   15   17   16    7
```

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. 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 655515 with 70 runs
  Lengths: 48361 46812 15963 15609 27617 ...   655  1289  1208    49    73
  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 the `frag.len`, `param` and `data` from the last chapter, as well as the `bam.files` vector we defined at the start. We'll also need the `aveLogCPM` function from edgeR, 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 [Bourgon et al., 2010]. 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 region.

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

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-2.211  -2.189  -2.110  -2.041  -1.972   9.813
```

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    NA's
logical 3673595   15073       0
```

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 to provide strong evidence for DB. 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    NA's
logical 3620807   67861       0
```

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 `filterWindows` 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 <- filterWindows(data, type="proportion")$filter > 0.999
> sum(keep)

[1] 54626
```

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 can be 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)
```

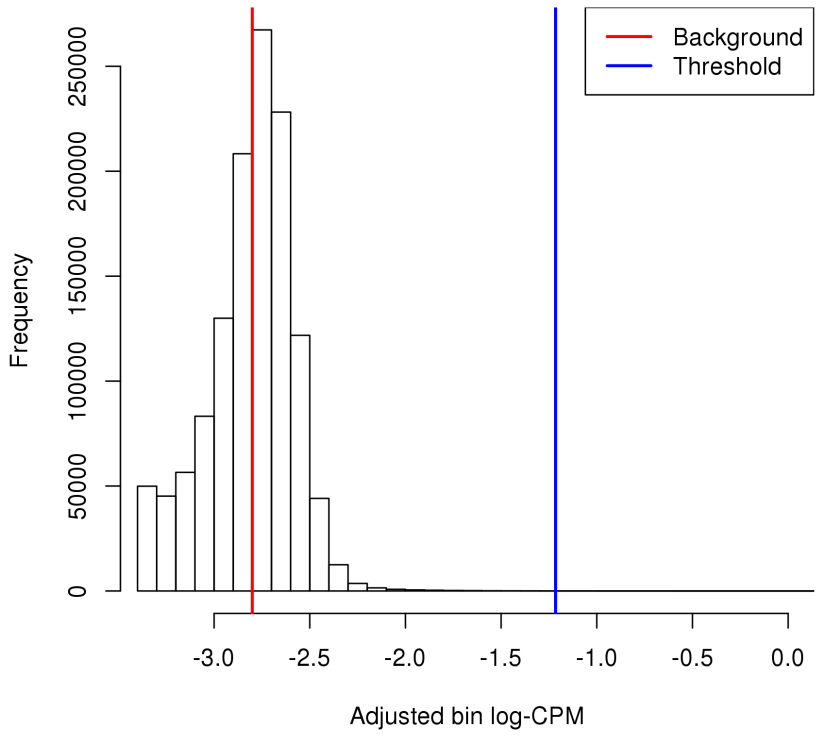
The median of the average abundances across all bins can be computed and used as a global estimate of the background coverage. This global background can then be compared to the window-based abundances. However, some care is required as the sizes of the regions used for read counting are different between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

With `type="global"`, the `filterWindows` function returns the increase in the abundance of each window over the global background. Windows can be filtered by setting some minimum threshold on this increase. 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 <- filterWindows(data, background=binned, type="global")
> keep <- filter.stat$filter > log2(3)
> sum(keep)

[1] 20571
```

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$back.abundances, xlab="Adjusted bin log-CPM", breaks=100, main="",
+     xlim=c(min(filter.stat$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. This 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 "neighbourhood" over which a local estimate of non-specific enrichment for that window can be obtained. The counts for this region can be obtained with the aptly-named `regionCounts` function. This should be synchronized with `windowCounts` by using the same `param`, if any non-default settings were used.

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

Counts for each window are subtracted from the counts for its neighbourhood. 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 to reflect the effective size of the neighbourhood. Again, the abundance of the neighbourhood is scaled down for a valid comparison to that of the corresponding window. This work is done by setting `type="local"` for `filterWindows`, which returns the enrichment values i.e., the increase in the abundance of each window over its neighbourhood.

```
> filter.stat <- filterWindows(data, wider, type="local")
```

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 neighbourhood abundance is required for retention of each window. This roughly mimics the behaviour of single-sample peak-calling programs such as MACS [Zhang et al., 2008].

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

[1] 9533
```

Note that this procedure also assumes that no other enriched regions are present in each neighbourhood. 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 neighbourhood.

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 [Lun and Smyth, 2014]. 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 approach uses 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    NA's
logical 2822727  865941       0
```

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 favour of a high-abundance non-DB neighbour.

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.

```
> in.demo <- windowCounts(c(bam.files, "input.bam"), ext=frag.len, param=param)
> chip <- in.demo[,1:4]
> control <- in.demo[,5]
```

Some additional work is required to account for composition biases (see Section 4.2.1). These are likely to be present when comparing ChIP to negative control samples. A simple strategy for normalization involves counting reads into large bins, as shown below.

```
> in.binned <- windowCounts(c(bam.files, "input.bam"), bin=TRUE, width=10000, param=param)
> chip.binned <- in.binned[,1:4]
> control.binned <- in.binned[,5]
```

The `filterWindows` function computes the enrichment of the ChIP counts over the control counts for each window. A larger `prior.count` of 5 is 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. Note that the global and local background estimates require less protection (`prior.count` of 2) as they are derived from larger bins with more counts.

```
> filter.stat <- filterWindows(chip, control, type="control", prior.count=5,
+     norm.fac=list(chip.binned, control.binned))
```

The binned counts are also supplied as a list to the `norm.fac` argument, where the first and second elements contain the counts for the ChIP and control samples, respectively. 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] 5918
```

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 experiment 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.

```
> require(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,  21078496]      - |   100009600
  100009609     chr7 [ 84940169,  84967009]      - |   100009609
  100009614    chr10 [ 77708446,  77712009]      + |   100009614
  100009664    chr11 [ 45805083,  45842878]      + |   100009664
     100012     chr4 [144157556, 144165651]      - |      100012
     100017     chr4 [134745412, 134771004]      - |      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] 2009551
```

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 the `bam.files` vector again (from the introduction). We'll also need the `param` object that we defined in Chapter 2. You'll notice that that a number of other BAM files are used in 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 in large bins and the counts are used for normalization with the `normalize` wrapper function. This uses the trimmed mean of M-values (TMM) method [Robinson and Oshlack, 2010] to correct for any sytematic 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 spurious.

```
> binned <- windowCounts(bam.files, bin=TRUE, width=10000, param=param)
> normfacs <- normalize(binned)
> normfacs

[1] 1.007535 0.974537 1.013773 1.004618
```

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.

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

4.2.2 Choosing a bin size

By definition, read coverage is low for background regions. 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. This reduces the effectiveness of the trimming procedure during normalization. These problems can be overcome by using large bins to increase the size of the counts prior to TMM normalization.

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 suitable for most datasets.

```
> demo <- windowCounts(bam.files, bin=TRUE, width=5000, param=param)
> normalize(demo)

[1] 1.0090735 0.9754931 1.0106443 1.0052051

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

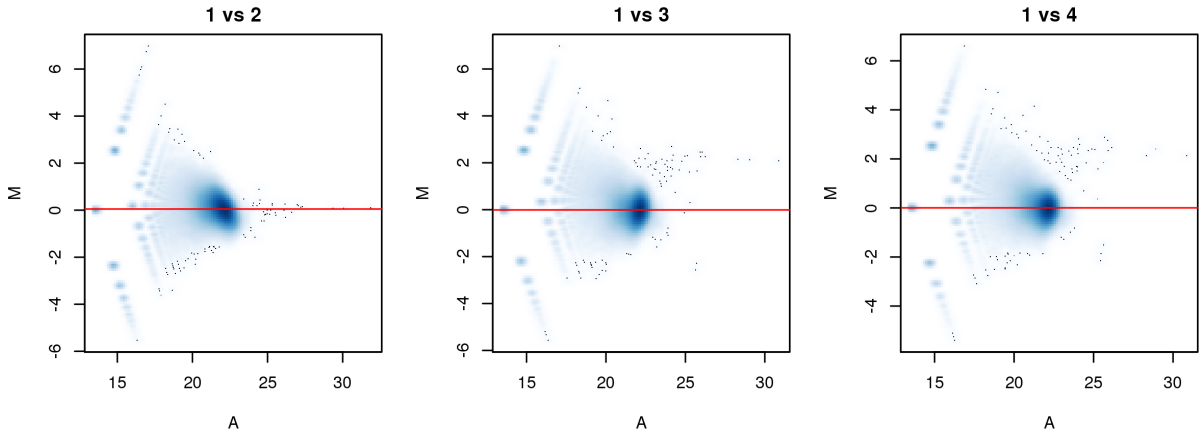
```
[1] 1.007548 0.972130 1.016127 1.004759
```

These factors are consistently close to unity, which suggests that composition bias is negligble in this dataset. See Section 4.3.2 for some examples with greater bias.

4.2.3 Visualizing normalization efforts 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)
> for (i in 1:(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 are commonly observed in ChIP-seq data. This refers 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. In the examples below, reads are counted into 150 bp windows for histone mark data. High-abundance windows are chosen using a global filtering approach described in Section 3.4. The TMM method can then be applied to eliminate systematic differences across those windows.

```
> me.files <- c("h3k4me3_mat.bam", "h3k4me3_pro.bam")
> me.demo <- windowCounts(me.files, width=150, param=param)
> me.bin <- windowCounts(me.files, bin=TRUE, width=10000, param=param)
> keep <- filterWindows(me.demo, me.bin, type="global")$filter > log2(3)
> me.eff <- normalize(me.demo[keep,])
> me.eff

[1] 0.7914877 1.2634435

> ac.files <- c("h3ac.bam", "h3ac_2.bam")
> ac.demo <- windowCounts(ac.files, width=150, param=param)
> ac.bin <- windowCounts(ac.files, bin=TRUE, width=10000, param=param)
> keep <- filterWindows(ac.demo, ac.bin, type="global")$filter > log2(5)
> ac.eff <- normalize(ac.demo[keep,])
> ac.eff

[1] 1.2012587 0.8324601
```

This method 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. However, 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. Also, care is required when filtering prior to normalization. Retaining too many windows will include background regions, while retaining too few will result in unstable estimates. See Chapter 3 for more details.

Note that TMM normalization is performed on window counts instead of bin counts. This is possible as the former should be large enough after filtering and retention of high-abundance candidates (Chapter 3). Direct use of the windows also ensures that there is no systematic difference in the counts between libraries during DB testing. In practice, the estimated normalization factors are usually robust to the choice of window or bin size.

4.3.2 Checking normalization with MA plots

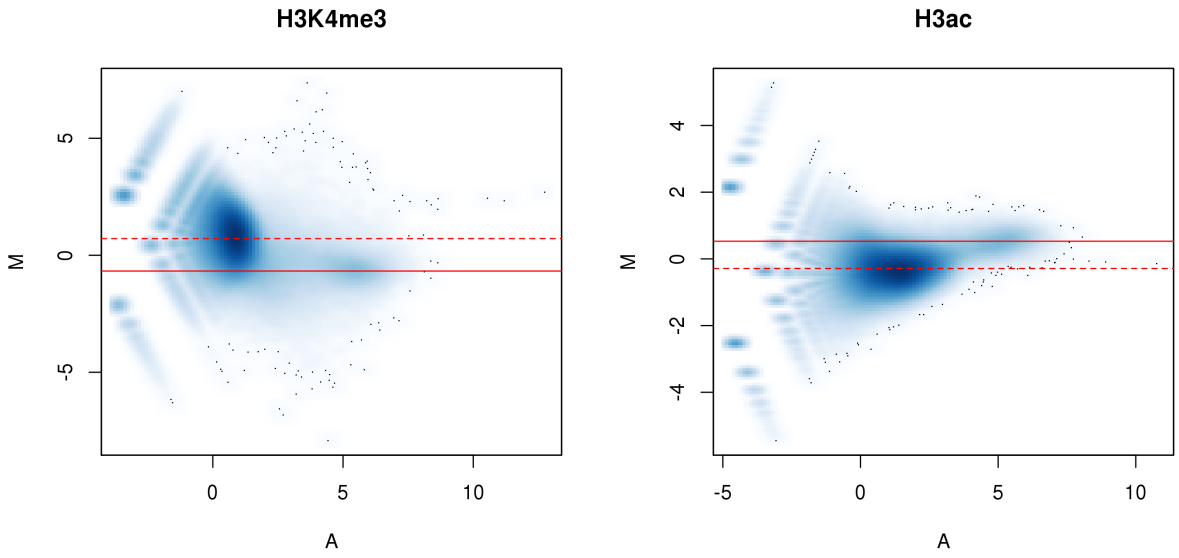
The effect of normalization can be visualized with MA plots. Plots are constructed using counts for 10 kbp bins, rather than with those from the windows. This is useful as the behaviour of the entire genome can be examined, rather than just that of the high-abundance windows. It also allows calculation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

```
> me.comp <- normalize(me.bin)
> me.comp
```

```
[1] 1.2801358 0.7811671

> ac.comp <- normalize(ac.bin)
> ac.comp

[1] 0.904996 1.104977
```

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)
+ }
```



4.3.3 Choosing between normalization strategies

The normalization strategies for composition and efficiency biases are mutually exclusive, as only one set of factors will ultimately be used. The choice between the two methods depends on whether one assumes that the systematic differences at high abundances represent genuine DB events. If so, composition bias should be removed to preserve the assumed DB. Otherwise, the differences must represent efficiency bias and should be removed. 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" unless there is evidence for a confounding efficiency bias.

4.4 Dealing with trended biases

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 is provided in `normalize` by setting `type="loess"`. This is based on the fast loess algorithm [Ballman et al., 2004] 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. Filtering prior to normalization is strongly recommended, to avoid inaccuracies in loess fitting at low abundances.

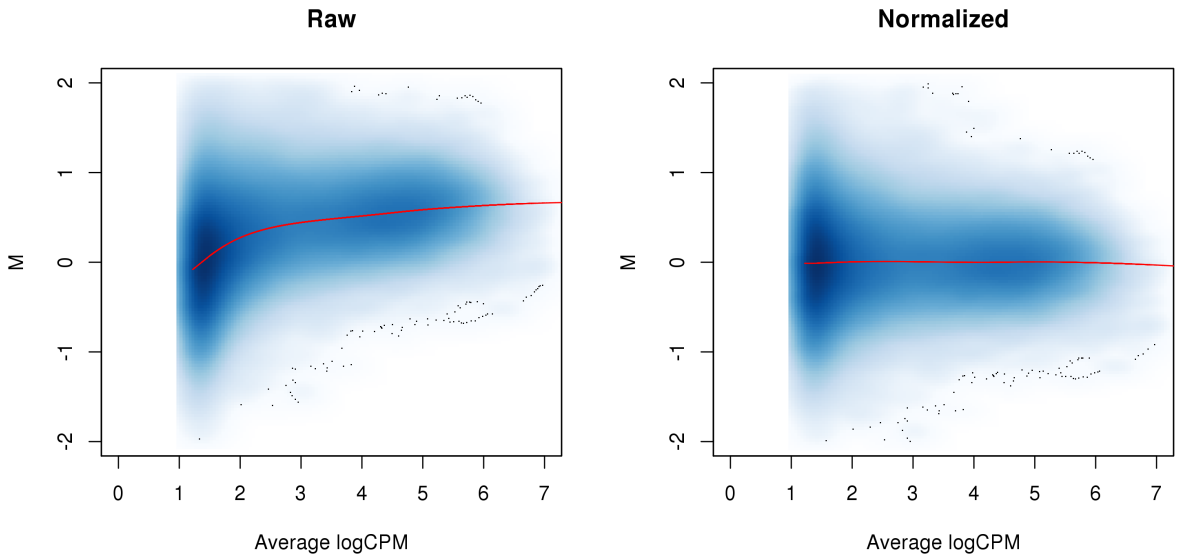
The example below operates on the filtered counts for 2 kbp windows. This window size is chosen purely for aesthetics in this demonstration, as the trend is less obvious at smaller widths. Obviously, users should pick a more appropriate value for their analysis.

```
> ac.demo2 <- windowCounts(ac.files, width=2000L, param=param)
> filtered <- filterWindows(ac.demo2, ac.bin, type="global")
> keep <- filtered$filter > log2(4)
> ac.demo2 <- ac.demo2[keep,]
> ac.off <- normalize(ac.demo2, type="loess")
> head(ac.off)

          [,1]       [,2]
[1,] 0.1382847 -0.1382847
```

```
[2,] 0.1382847 -0.1382847
[3,] 0.1673240 -0.1673240
[4,] 0.1805074 -0.1805074
[5,] 0.2021496 -0.2021496
[6,] 0.2144884 -0.2144884
```

MA plots can be examined to determine whether normalization was successful. Any abundance-dependent trend in the M-values should be eliminated. In addition, any filter statistic for the windows should be based on the average abundance from `aveLogCPM`. An average abundance threshold will act as a clean vertical cutoff in the plots below. Spurious trends that might affect normalization will not be introduced at the filter boundary.

```
> abval <- filtered$abundances[keep]
> o <- order(abval)
> adjc <- cpm(asDGEList(ac.demo2), log=TRUE)
> mval <- adjc[,1]-adjc[,2]
> fit <- loessFit(x=abval, y=mval)
> smoothScatter(abval, mval, ylab="M", xlab="Average logCPM",
+     main="Raw", ylim=c(-2,2), xlim=c(0, 7))
> lines(abval[o], fit$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")
```



Note that all non-linear methods assume that most bins/windows are not DB at each abundance. This is a stronger assumption than that for scaling methods, which only require a non-DB majority across all features. Removal of the trend may not be appropriate if it represents some genuine biological phenomenon, e.g., involving changes in overall binding. In addition, the computed offsets are not compatible with the normalization factors from the scaling methods. Only one of these sets of values will ultimately be used by edgeR.

4.5 A word on other biases

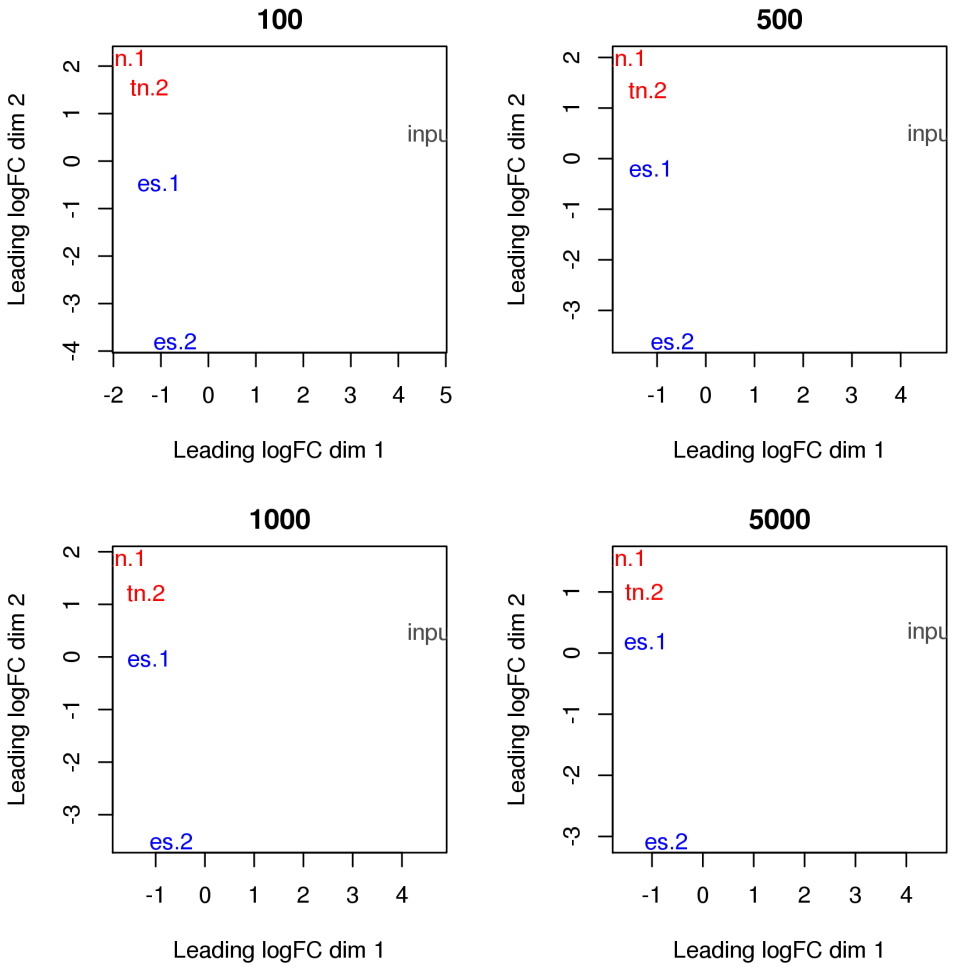
No normalization is performed to adjust for differences in mappability or sequencability between different regions of the genome. Region-specific biases are assumed to be constant between libraries. This is generally reasonable as the biases depend on fixed properties of the genome sequence such as GC 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.

4.6 Examining replicate similarity with MDS plots

On a semi-related note, the binned 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))
> binned <- windowCounts(c(bam.files, "input.bam"), bin=TRUE, width=2000L, param=param)
> adj.counts <- cpm(asDGEList(binned), log=TRUE)
> for (top in c(100, 500, 1000, 5000)) {
+     out <- plotMDS(adj.counts, main=top, col=c("blue", "blue", "red", "red", "grey30"),
+         labels=c("es.1", "es.2", "tn.1", "tn.2", "input"), top=top)
+ }
```



Replicates from different groups should form separate clusters in the MDS plot. This indicates that the results are reproducible and that the effect sizes are large. Such clustering is observed in the example above, along with separation from the input sample (note that the input is not essential for plotting, and is only included for demonstration purposes). 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, the removal of the corresponding libraries may be necessary to obtain sensible results.

Chapter 5

Testing for differential binding

Here, we'll need the `filtered.data` from Chapter 3 along with the original `data` from Chapter 2. We'll also use the `normfacs` vector from Chapter 4, as well as the `design` matrix from the introduction. 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 [Lund et al., 2012]. Counts are modelled using NB distributions that account for overdispersion between biological replicates [Robinson and Smyth, 2008]. 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 [Law et al., 2014]. This author's desire to increase his h-index may also be a factor [Chen et al., 2014].

5.1.2 Setting up the data

A `DGEList` object is first formed from the count matrix, library sizes and normalization factors. Here, the `normfacs` vector from TMM normalization of background bins is used to remove composition bias. If an offset matrix was generated (e.g., from non-linear normalization), this can be assigned into `y$offset` for later use in the various edgeR functions.

```
> y <- asDGEList(filtered.data, norm.factors=normfacs)
```

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 [McCarthy et al., 2012]. 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.

```
> 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(c("es", "es", "tn", "tn"))`
[1] "contr.treatment"
```

5.2 Estimating the dispersions

5.2.1 Stabilising estimates with empirical Bayes

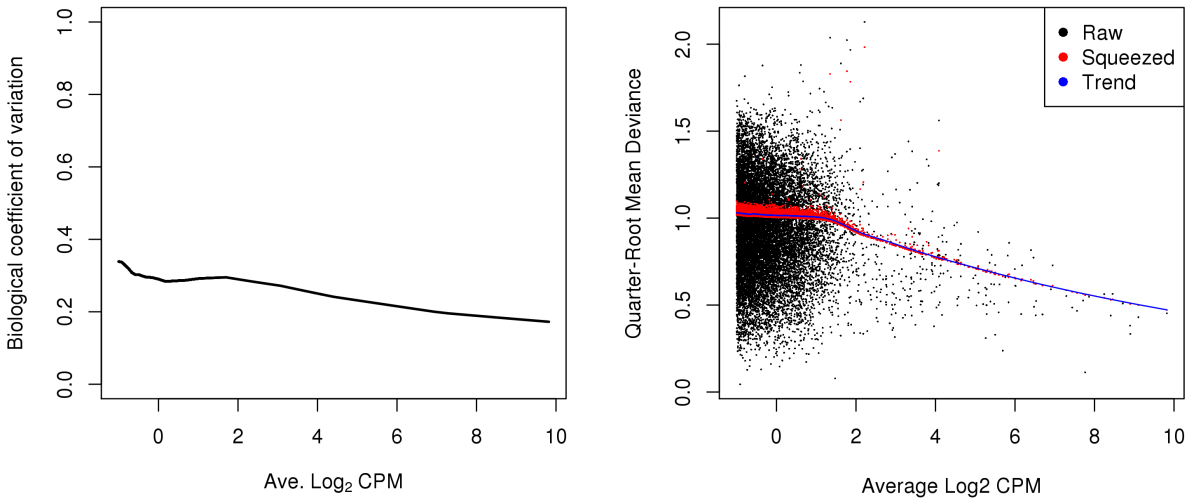
Under the QL framework, both the QL and NB dispersions are used to model biological variability in the data [Lund et al., 2012]. The former ensures that the NB mean-variance relationship is properly specified with appropriate contributions from the Poisson and Gamma components. The latter accounts for 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 [McCarthy et al., 2012]. 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 [Lund et al., 2012]. The ideal amount of shrinkage is determined from the variability of the dispersions.

```
> par(mfrow=c(1,2))
> y <- estimateDisp(y, design)
> 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"))
> fit <- glmQLFit(y, design, robust=TRUE)
> plotQLDisp(fit)
```

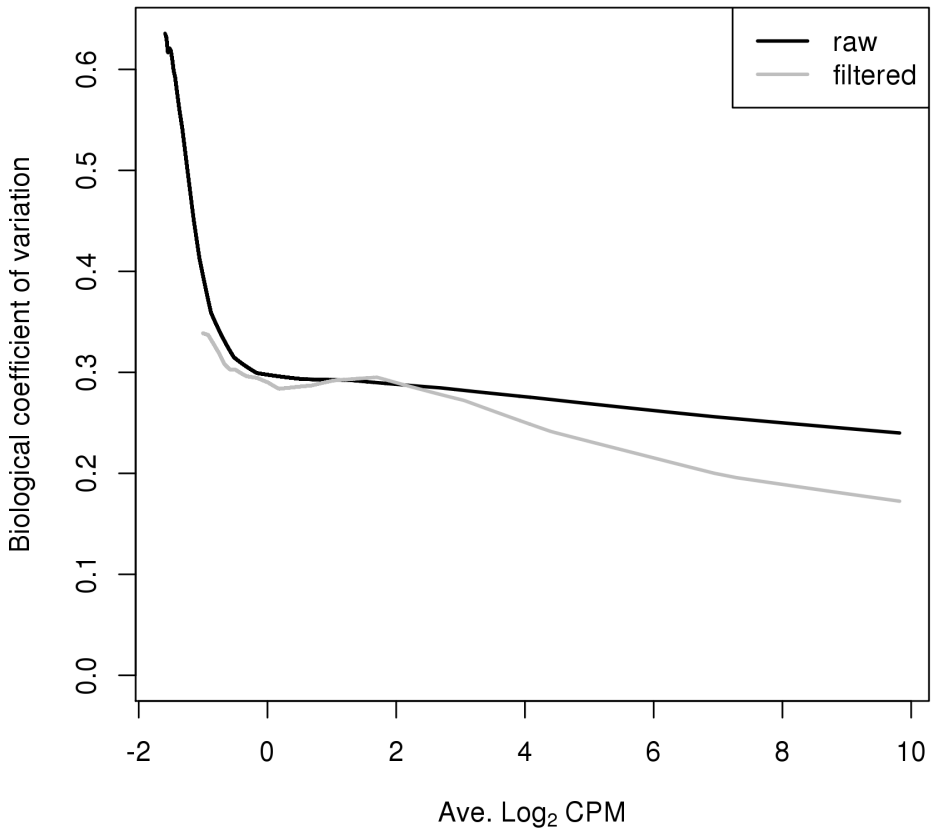
The effect of EB stabilisation can be visualized by examining the biological coefficient of variation (for the NB dispersion) and the quarter-root deviance (for the QL dispersion). These plots can also be used to decide whether the fitted trend is appropriate. Sudden irregularities may be indicative 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.



A strong trend may also be observed where the 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 this makes any difference. Such 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 # some filtering; otherwise, it takes too long.
> 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.
 0.6825 40.3800 40.3800 40.3400 40.3800 40.3800
```

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 [Phipson et al., 2013]. 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 [Smyth, 2004]. 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 [Lund et al., 2012]. 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 0.2658709 -0.9478874 0.1329712 0.71718466
2 0.2821774 -0.9519483 0.1428751 0.70732624
3 1.7985794 -0.8667826 5.7293640 0.02118629
4 0.7674033 -0.9870740 1.0429882 0.31292280
5 0.9151180  0.3674874 3.1213088 0.08447706
6 1.0754719  1.1797634 5.1014896 0.02911051
```

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.

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 0.2684871 -0.9478874 0.2120472 0.645167508
2 0.2780220 -0.9519483 0.2272559 0.633566015
3 1.8018229 -0.8667826 8.2162847 0.004151611
4 0.7577310 -0.9870740 1.5873122 0.207710563
5 0.9143507  0.3674874 4.2230561 0.039878274
6 1.0745323  1.1797634 7.2749300 0.006992362
```

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.

Chapter 6

Correction for multiple testing

All right, we're almost there. This chapter needs the `results` object from the last chapter. We also need a bunch of things from Chapter 3 - namely, the `filtered.data` list, the `broads` object and the `keep.simple` and `maxed` vectors. Finally, we'll need `ac.files` and `param` from Chapters 4 and 2, respectively, for a little demonstration.

6.1 Problems with false discovery rate control

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 [Benjamini and Hochberg, 1995] 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 [Lun and Smyth, 2014].

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 forming one cluster and 2 windows forming a separate cluster. Each cluster represents a 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%.

6.2 Restoring FDR control with clustered windows

Misinterpretation of the FDR can be avoided by obtaining a single p-value for each region. In particular, several strategies can be used to cluster adjacent windows into regions. A combined p-value can then be computed for each cluster, based on the p-values of the constituent windows [Simes, 1986]. This tests the joint null hypothesis for each cluster, i.e., that no enrichment is observed across any sites within the corresponding region. The combined p-values are then adjusted using the BH method to control the region-level FDR.

An alternative approach is to choose a single window to represent each cluster/region. For example, the window with the highest average abundance in each cluster can be used. This is sensible for analyses involving sharp binding events, where each cluster is expected to be small and contain no more than one binding site. Thus, a single window (and p-value) can reasonably be used as a representative of the entire region. The BH method can then be applied to the corresponding p-values of the representative windows from all clusters.

Both approaches are available in the csaw package. The combining procedure is known as Simes' method and is implemented in the `combineTests` function. Similarly, selection of a representative window can be performed using the `getBestTest` function. Examples of their usage are shown below, along with demonstrations of the different clustering strategies.

6.3 Clustering with external information

Combined p-values can be computed for a pre-defined set of regions based on the windows overlapping those regions. The most obvious source of pre-defined 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 [Lun and Smyth, 2014]. In either case, the `findOverlaps` function from the GenomicRanges package can be used to identify all windows in or overlapping each specified region.

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

Hits object with 15447 hits and 0 metadata columns:
          queryHits subjectHits
          <integer>   <integer>
      [1]         7        8690
      [2]         7        8691
      [3]         7        8692
      [4]        18       10380
      [5]        18       10381
      ...       ...         ...
  [15443]     23646        8502
  [15444]     23646        8503
  [15445]     23649        8216
  [15446]     23649        8217
```

```
  [15447]     23649        8218
  -------
  queryLength: 23653
  subjectLength: 15073
```

The `combineOverlaps` function can be used to 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. The number of windows in each region that change by more than $\sqrt{2}$ in either direction are also reported. 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),])
```

```
   nWindows logFC.up logFC.down       PValue         FDR
7         3        3          0 0.0001795767 0.003158611
18        4        4          0 0.0002309644 0.003687220
22        1        1          0 0.1736536776 0.244063724
23        5        5          0 0.0020820527 0.012592020
25        4        0          0 0.9859862430 0.989442610
28        3        3          0 0.0056620470 0.024264301
```

At this point, one might imagine that it would be simpler to just collect and analyze counts over the pre-defined 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 boundaries for called peaks can lead to loss of DB detection power due to "contamination" by reads from background regions. In both cases, window-based methods may be more robust as each interval of the promoter/peak region is examined separately [Lun and Smyth, 2014].

6.4 Quick and dirty clustering

Clustering can also be performed with a simple single-linkage algorithm in the `mergeWindows` function. This approach avoids potential problems with the other clustering strategies, e.g., peak-calling errors, incorrect or incomplete annotation. 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 favour a region-based interpretation of the results, while smaller values ($<$ 200 bp) allow resolution of individual binding sites.

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

```
GRanges object with 4422 ranges and 0 metadata columns:
         seqnames               ranges strand
            <Rle>            <IRanges>  <Rle>
     [1]     chr1     [6466701, 6466760]      *
     [2]     chr1     [7088951, 7088960]      *
     [3]     chr1     [7397851, 7398110]      *
     [4]     chr1     [9541401, 9541510]      *
     [5]     chr1     [9545251, 9545360]      *
     ...      ...                    ...    ...
  [4418]     chrX [164076151, 164076560]      *
  [4419]     chrX [166170351, 166170410]      *
  [4420]     chrY [   259151,    259360]      *
  [4421]     chrY [ 90808801,  90808860]      *
  [4422]     chrY [ 90812101,  90812910]      *
  -------
  seqinfo: 66 sequences from an unspecified genome
```

A combined p-value is computed for each cluster with the `combineTests` function. The BH method is then applied to control the FDR across all detected clusters. The number of `up` or `down` windows can be used to gauge whether binding increases or decreases across the cluster. A complex DB event may be present if both `up` and `down` are substantially 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).

```
> tabcom <- combineTests(merged$id, results$table)
> head(tabcom)

  nWindows logFC.up logFC.down     PValue        FDR
1        2        0          0 0.71718466 0.78170830
2        1        1          0 0.02118629 0.05790222
3        6        6          0 0.08733154 0.15428688
4        3        3          0 0.04513129 0.09569469
5        3        3          0 0.05843050 0.11442856
6        2        2          0 0.01414122 0.04413018
```

If many adjacent windows are present, very large clusters may be formed that are difficult to interpret. A simple check can be used 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$region))

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   10.0    60.0   110.0   159.4   160.0 15710.0
```

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 neighbours, 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(rowRanges(filtered.data), tol=1000L, max.width=5000L)
> summary(width(merged.max$region))

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   10.0    60.0   110.0   158.3   160.0  4410.0
```

6.5 Integrating results from multiple window sizes

The sensitivity of the analysis to the choice of window size can be mitigated by testing a range of different widths. DB results from each width can be integrated by clustering adjacent windows together (even if they are of differing sizes), 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)
> 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 `consolidateSizes` function can then be applied to combine these results. This merges windows of all sizes into a single set of clusters, and computes a combined p-value from the associated p-values for each cluster. However, if a cluster contains many small windows, the DB results for the small window size will contribute most to the combined p-value. This is not ideal when results from all window sizes are of equal interest. Equal contributions from each window size can be enforced by setting `equiweight=TRUE`, whereby a weighted version of Simes' method [Benjamini and Hochberg, 1997] is used. The weight assigned to each window is inversely proportional to the number of windows of that size in the same cluster.

```
> cons <- consolidateSizes(data.list=list(ac.small, ac.large),
+     result.list=list(mock.small, mock.large), equiweight=TRUE)
> cons$region

GRanges object with 29893 ranges and 0 metadata columns:
          seqnames               ranges strand
             <Rle>            <IRanges>  <Rle>
      [1]     chr1   [4571001, 4572500]      *
```

```
      [2]     chr1   [4783501, 4787000]      *
      [3]     chr1   [4806501, 4809500]      *
      [4]     chr1   [4856501, 4859500]      *
      [5]     chr1   [5082501, 5084500]      *
      ...      ...                   ...    ...
  [29889]     chrY [  897001,   899000]      *
  [29890]     chrY [ 1010001,  1012000]      *
  [29891]     chrY [ 1244001,  1246500]      *
  [29892]     chrY [ 1285001,  1287000]      *
  [29893]     chrY [90738501, 90740500]      *
  -------
  seqinfo: 66 sequences from an unspecified genome
```

The same function can also be used to consolidate windows of varying size that overlap pre-specified regions. This is done by specifying the `region` argument, as shown below.

```
> cons.broad <- consolidateSizes(data.list=list(ac.small, ac.large),
+     result.list=list(mock.small, mock.large), region=broads, equiweight=TRUE)
```

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 are variable.

6.6 Choosing a single representative window

6.6.1 Based on differential binding

In some cases, it may be necessary to also report the window in which the strongest DB is found. This information can be useful for identifying the change in binding within large clusters, or to narrow down the relevant sequence for motif discovery. Identification of the most significant (i.e., "best") window can be performed using the `getBestTest` function. This reports the index of the window with the lowest p-value in each cluster.

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

```
  best     logFC      logCPM         F     PValue        FDR
1    2 0.2821774 -0.95194826 0.1428751 1.00000000 1.00000000
2    3 1.7985794 -0.86678263 5.7293640 0.02118629 0.06067733
3    7 1.2158831  1.15706343 6.2131300 0.10003304 0.17772041
4   11 1.4410731  0.01763535 5.2094162 0.08264182 0.15484836
5   15 1.5000536 -0.77378156 4.5573423 0.11581199 0.19739876
6   17 2.1479461 -0.80663321 8.0137840 0.01414122 0.04642351
```

A common use of the `best` index is to obtain the log-fold change of the best window in each cluster. This value is useful as it focuses on the most DB interval within the cluster. In contrast, taking the average log-fold change across all windows in a cluster will understate the magnitude of DB. It may also be useful to report the start location of the best window. This allows investigators to easily identify the DB subinterval within large clusters.

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

```
  best.logFC best.start
1  0.2821774    6466751
2  1.7985794    7088951
3  1.2158831    7398001
4  1.4410731    9541451
5  1.5000536    9545351
6  2.1479461    9748451
```

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

```
   best.logFC best.start
7   4.5768433   32657051
18  3.9835039    8259301
22  0.9871556  118937801
23  3.6001595   92934401
25 -0.3390825   71596001
28  2.4195175    4137001
```

Typically, the best window will only be used as a descriptive measure for each cluster. However, more statistical rigour is required if these windows are treated as the features of interest over which the error rate is to be controlled. A Bonferroni correction is applied to the p-value of each best window to obtain the corresponding `PValue` in `tab.best`. This is necessary to account for the implicit multiple testing across all windows in each cluster.

6.6.2 Based on average abundance

Alternatively, the best window can be defined as the one with the highest average abundance in each cluster. This represents the window with the strongest binding for the target protein, though not necessarily the strongest DB. As the average abundance is (roughly) independent of the p-value, there is no need to correct for multiple testing within each cluster.

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

```
  best     logFC      logCPM         F      PValue         FDR
1    1 0.2658709 -0.94788742 0.1329712 0.717184659 0.76807715
2    3 1.7985794 -0.86678263 5.7293640 0.021186295 0.05719524
3    6 1.0754719  1.17976338 5.1014896 0.029110514 0.07076783
4   11 1.4410731  0.01763535 5.2094162 0.027547273 0.06840928
5   14 1.4995852 -0.66575935 4.5401771 0.038953670 0.08701439
6   17 2.1479461 -0.80663321 8.0137840 0.007070609 0.02804146
```

For sharp binding events, it may be preferable to restrict the DB analysis to these windows. This will usually provide more power as it avoids the conservativeness of Simes' method. The use of the highest-abundance window will also favour detection of simple DB events, where DB occurs at the most strongly bound loci in each cluster. This may be preferable for some studies, given that complex DB events are often difficult to interpret.

The obvious drawback is that information is lost when the analysis is restricted to a single window. Detection will fail for complex DB events, e.g., clusters of TF binding sites, spreading of diffuse marks. Loss of information also increases the sensitivity of the analysis to the clustering procedure. For example, a DB site will be ignored if it is clustered alongside a stronger non-DB site, as the window covering the latter will represent the cluster.

A less formal approach is to simply report the log-fold change of the most abundant window in each cluster. This will indicate whether any DB is present at the strongest binding site within the corresponding genomic interval. If not, the DB event may be complex, e.g., involving multiple peaks or peaks that change in shape or position between conditions. These require more careful interpretation than simpler "binary" (i.e., on/off) changes.

```
> tabcom$mab.logFC <- tab.ave$logFC # 'mab' stands for most abundant.
```

6.6.3 Weighting windows on abundance

A more graduated approach uses the weighted version of Simes' method, where the window with the highest abundance is upweighted. In the example below, the weight assigned to the top window is increased relative to that of other windows in the same cluster. This means that the behaviour of the top window will have a greater influence on the final combined p-value. That said, the other windows in the cluster still have unity weights. Any DB events in those windows will still be considered when the p-values are combined.

```
> weights <- upweightSummit(merged$id, tab.ave$best)
> head(weights)
```

```
[1] 2 1 1 1 1 6
```

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

```
  nWindows logFC.up logFC.down     PValue        FDR
1        2        0          0 0.71718466 0.77749217
2        1        1          0 0.02118629 0.05744071
3        6        6          0 0.04574509 0.09655599
4        3        3          0 0.03760941 0.08424393
5        3        3          0 0.04869209 0.10042743
6        2        2          0 0.01060591 0.03607642
```

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),])
```

```
    best      logFC     logCPM           F       PValue         FDR
7   8691  3.2094739 -0.4991505 17.95159644 0.0001197178 0.002124181
18 10381  1.3722229  0.7281035  6.63557489 0.0135703725 0.041924249
22  4457  0.9871556 -0.8557656  1.91501281 0.1736536776 0.246289293
23   404  1.1650371  0.4128078  4.47883138 0.0402318308 0.084767165
25 12508 -0.1803523 -0.2788300  0.09146378 0.7638039864 0.799944955
28 10967  2.4195175 -0.6229089 10.98598694 0.0018873490 0.011627370
```

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

6.7 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, filtering based on pre-defined regions. In such cases, it may be preferable to apply one of the other, milder filters first. This ensures that sufficient windows are retained for stable normalization and/or EB shrinkage. The aggressive filters can then be applied after the statistics have been calculated. 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.

For example, say the user wants to focus on local maxima. Assume that the test statistics are computed from `filtered.data`, as generated with the mild `keep.simple` filter in Chapter 3. These filtered windows are re-filtered using the `maxed` vector from Section 3.5.2, to identify those that are local maxima. The re-filtered windows and their corresponding test statistics can then be used for all FDR-controlling procedures described in this chapter.

```
> keep.new <- maxed[keep.simple]
> refilt.data <- filtered.data[keep.new,]
> refilt.res <- results[keep.new,]
```

Any filter vector from Chapter 3 can be used in place of `maxed`, so long as it was generated with the original `data`. Generating the filter vectors with `data` instead of `filtered.data` is generally recommended, as it avoids any unforseen interactions between filters. In particular, the local maxima filter depends on the abundance of adjacent windows. It may behave incorrectly if it is applied after another filter that removes high-abundance windows.

Chapter 7

Post-processing steps

This is where we bring it all together. We'll need the `merged` list and the `tabcom` table 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 `bam.files` that we started off with.

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 upand 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.

```
> require(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|0-1|+"
[3] ""                                ""
[5] "Rrs1|0|+,Adhfe1|0|+"             "Vcpip1|0|-,1700034P13Rik|0-1|+"

> head(anno$left)

[1] ""                "" ""                ""
[5] ""                "Vcpip1|1|-[19]"

> head(anno$right)
```

```
[1] ""                                  ""
[3] ""                                  "Rrs1|1|+[3898]"
[5] "Rrs1|1|+[48],Adhfe1|1-2|+[2686]" "1700034P13Rik|2|+[3989]"
```

Character vectors of compact string representations are provided to summarize the features overlapped by each supplied region. Each pattern contains `GENE|EXONS|STRAND` to describe the strand and overlapped exons of that gene. Promoters are labelled as exon `0` whereas introns are labelled as `I`. For `left` and `right`, an additional `[DISTANCE]` field is included. This indicates the gap between the annotated feature and the supplied region.

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 286315 ranges and 3 metadata columns:
            seqnames                ranges strand   |      symbol      exon
               <Rle>             <IRanges>  <Rle>   | <character> <integer>
  100009600     chr9   [21062393, 21062717]      -   |       Zglp1         7
  100009600     chr9   [21062894, 21062987]      -   |       Zglp1         6
  100009600     chr9   [21063314, 21063396]      -   |       Zglp1         5
  100009600     chr9   [21066024, 21066377]      -   |       Zglp1         4
  100009600     chr9   [21066940, 21067925]      -   |       Zglp1         3
        ...      ...                   ...    ... ...         ...       ...
      99889     chr3 [ 85785218,  85887518]      -   |      Arfip1        -1
      99890     chr3 [110246104, 110250999]      -   |       Prmt6        -1
      99899     chr3 [151730923, 151749959]      -   |       Ifi44        -1
      99929     chr3 [ 65528447,  65555518]      +   |      Tiparp        -1
      99982     chr4 [136550533, 136602723]      -   |       Kdm1a        -1
             internal
            <integer>
  100009600         1
  100009600         1
  100009600         1
  100009600         1
  100009600         1
        ...       ...
      99889     23881
      99890     23882
      99899     23883
      99929     23884
      99982     23885
  -------
  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.

```
> expanded <- suppressWarnings(resize(merged$region, fix="center",
+     width=width(merged$region)+50))
> sbm.score <- checkBimodality(bam.files, expanded, width=frag.len)
> head(sbm.score)

[1] 1.272727 1.388889 1.544554 2.045455 1.461538 1.400000
```

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 effectively 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.gz", open="w")
> write.table(data.frame(as.data.frame(merged$region)[,1:3], tabcom, anno),
+     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 <- tabcom$FDR <= 0.05
> require(rtracklayer)
> test <- merged$region[is.sig]
```

```
> test$score <- -10*log10(tabcom$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  9748400  9748460 region1 13.55264  .
2 chr1 15805500 15805660 region2 16.55359  .
3 chr1 23762950 23762960 region3 14.80023  .
4 chr1 32172600 32172710 region4 13.23200  .
5 chr1 33565950 33566010 region5 25.31317  .
6 chr1 35985450 35985560 region6 26.23550  .
```

Another useful format is to store the results as a `GRanges` object. Statistics are stored in the various metadata fields, which is convenient as it synchronizes the subsetting of statistics and genomic coordinates. This object can also be saved to file and reloaded later for direct manipulation in the R environment, e.g., to find overlaps with other regions of interest.

```
> output.ranges <- merged$region
> elementMetadata(output.ranges) <- tabcom
> saveRDS(output.ranges, "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.

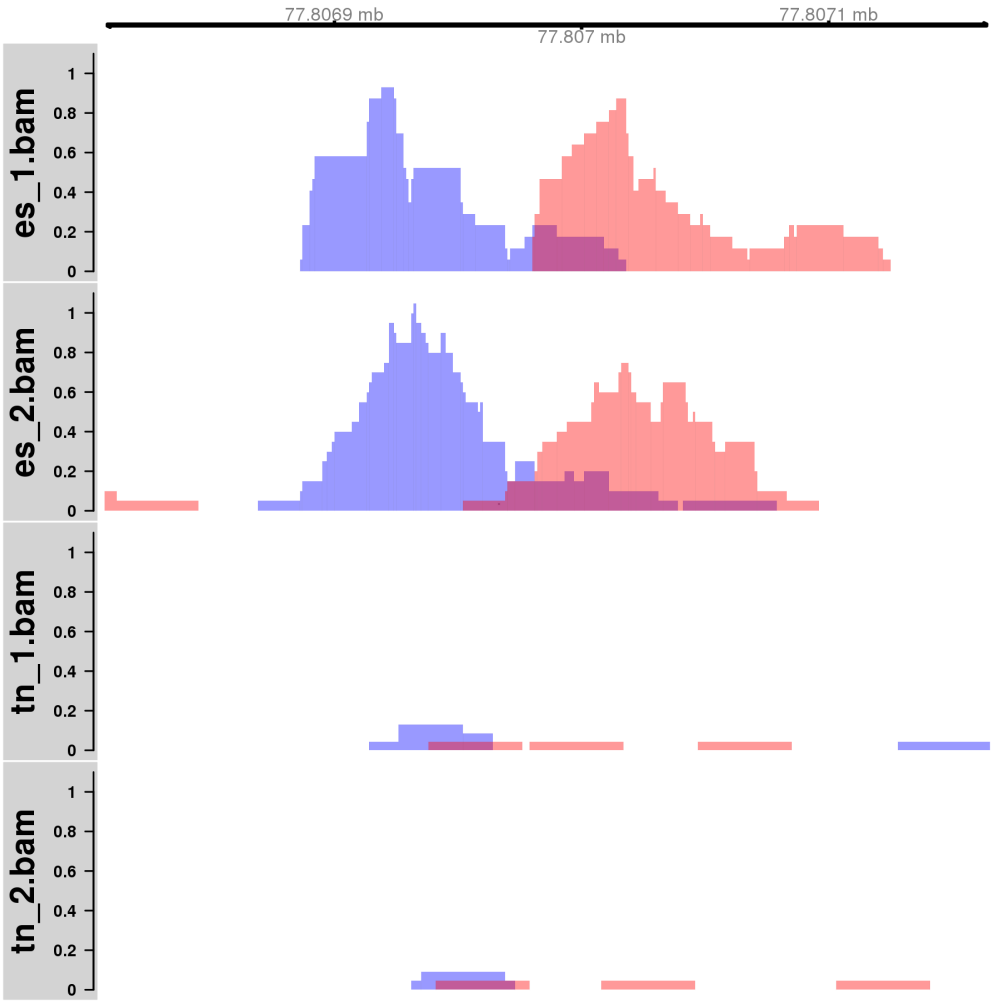
```
> cur.region <- GRanges("chr18", IRanges(77806807, 77807165))
> extractReads(bam.files[1], cur.region, param=param)
```

```
GRanges object with 45 ranges and 0 metadata columns:
       seqnames              ranges strand
          <Rle>           <IRanges>  <Rle>
   [1]    chr18 [77806886, 77806923]      +
   [2]    chr18 [77806887, 77806924]      +
   [3]    chr18 [77806887, 77806924]      +
   [4]    chr18 [77806887, 77806924]      +
   [5]    chr18 [77806890, 77806927]      +
   ...      ...                 ...    ...
  [41]    chr18 [77807048, 77807085]      -
  [42]    chr18 [77807068, 77807105]      -
  [43]    chr18 [77807082, 77807119]      -
```

```
  [44]    chr18 [77807084, 77807121]      -
  [45]    chr18 [77807087, 77807124]      -
  -------
  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.

```
> require(Gviz)
> collected <- list()
> for (i in 1:length(bam.files)) {
+     reads <- extractReads(bam.files[i], cur.region, param=param)
+     pcov <- as(coverage(reads[strand(reads)=="+"])/data$totals[i]*1e6, "GRanges")
+     ncov <- as(coverage(reads[strand(reads)=="-"])/data$totals[i]*1e6, "GRanges")
+     ptrack <- DataTrack(pcov, type="histogram", lwd=0, fill=rgb(0,0,1,.4), ylim=c(0,1.1),
+         name=bam.files[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))
```


77.8069 mb
77.807 mb
77.8071 mb
es_1.bam
es_2.bam
tn_1.bam
tn_2.bam
0
0.2
0.4
0.6
0.8
1

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 Datasets

8.1.1 Obtaining the FastQ files

The NFYA dataset used throughout the guide was first mentioned in Section 1.3. This was generated by Tiwari et al. [2012] and is available from the NCBI Gene Expression Omnibus (GEO) with the accession number GSE25532. FastQ files werer obtained from the Sequence Read Archive (SRA) with accessions of SRR074398 for `es_1.bam`, SRR074399 for `es_2.bam`, SRR074417 for `tn_1.bam`, SRR074418 for `tn_2.bam` and SRR074401 for `input.bam`.

The paired-end dataset used in Section 2.3 was generated by Pal et al. [2013] and is available from the NCBI GEO under the accession GSE43212. The lone FastQ file can be obtained from the SRA with the accession SRR642390 for `example-pet.bam`.

All libraries used in Section 2.4.1 were generated by Zhang et al. [2012] and are available from the NCBI GEO under the accession GSE31233. FastQ files can be obtained from the SRA under the accessions SRR330784 and SRR330785 for `h3ac.bam`; SRR330800 and SRR330801 for `h3k4me2.bam`; and SRR330814, SRR330815 and SRR330816 for `h3k27me3.bam`. Multiple FastQ files represent technical replicates that were merged into a single BAM file.

Finally, the H3K4me3 dataset in Section 4.3.1 was generated by Revilla-I-Domingo et al. [2012] and is available under the accession GSE38046. FastQ files can be obtained from the

SRA under the accessions SRR499732 and SRR499733 for `h3k4me3_pro.bam`, and SRR499716 and SRR499717 for `h3k4me3_mat.bam`. Again, technical replicates were merged together. For H3ac, the FastQ file at SRR330786 was also downloaded and used as `h3ac_2.bam`.

8.1.2 Alignment and processing to produce BAM files

Technically, each of the libraries described above are downloaded in the SRA format. These can be unpacked to yield FastQ files using the `fastq-dump` program from the SRA Toolkit (`http://www.ncbi.nlm.nih.gov/Traces/sra/?view=software`). For the lone paired-end library, users will need to specify `fastq-dump -split-files` to ensure that two separate files are produced, i.e., containing sequences from either end of each fragment.

Reads in the FastQ files were then aligned to the mm10 build of the mouse genome using subread v1.4.6 [Liao et al., 2013]. The subread software can be obtained from Bioconductor as the Rsubread package, or as a standalone C program from `http://subread.sourceforge.net`. The consensus threshold for alignment was set at 2 to accommodate short read lengths ($<$ 45 bp in all datasets). Only unique alignments were reported, and any tied alignments were split by Hamming distance. Default values were used for all other parameters. Paired-end data was aligned by supplying both FastQ files to subread within the same run.

Once aligned, SAM files were converted to BAM files using SAMtools v0.1.19 [Li et al., 2009]. BAM files were position-sorted with the `samtools sort` command, and duplicate reads were marked using the `MarkDuplicates` command from the Picard suite v1.117 (`http://broadinstitute.github.io/picard`). Any technical replicates were merged together using `samtools merge` to form a single library. Indexing was performed using `samtools index`.

If all relevant SRA files have been obtained, a set of explicit commands can be used to produce each BAM file. These commands are stored in a Bash file, accessible by running:

```
> system.file("doc", "sra2bam.sh", package = "csaw")
```

8.2 Session information

```
> sessionInfo()

R version 3.2.0 (2015-04-16)
Platform: x86_64-unknown-linux-gnu (64-bit)
Running under: CentOS release 6.4 (Final)

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [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      stats4    parallel  stats     graphics  grDevices utils
 [8] datasets  methods   base

other attached packages:
 [1] Gviz_1.13.3
 [2] rtracklayer_1.29.10
 [3] org.Mm.eg.db_3.1.2
 [4] RSQLite_1.0.0
 [5] DBI_0.3.1
 [6] XVector_0.9.1
 [7] TxDb.Mmusculus.UCSC.mm10.knownGene_3.1.3
 [8] GenomicFeatures_1.21.13
 [9] AnnotationDbi_1.31.17
[10] edgeR_3.11.7
[11] limma_3.25.9
[12] csaw_1.3.7
[13] SummarizedExperiment_0.2.1
[14] Biobase_2.29.1
[15] GenomicRanges_1.21.15
[16] GenomeInfoDb_1.5.8
[17] IRanges_2.3.12
[18] S4Vectors_0.7.6
[19] BiocGenerics_0.15.2

loaded via a namespace (and not attached):
 [1] statmod_1.4.21            locfit_1.5-9.1
 [3] VariantAnnotation_1.15.14 reshape2_1.4.1
 [5] splines_3.2.0             lattice_0.20-31
 [7] colorspace_1.2-6          XML_3.98-1.2
 [9] survival_2.38-2           foreign_0.8-63
[11] BiocParallel_1.3.26       RColorBrewer_1.1-2
[13] lambda.r_1.1.7            matrixStats_0.14.0
[15] plyr_1.8.3                stringr_1.0.0
[17] zlibbioc_1.15.0           Biostrings_2.37.2
[19] munsell_0.4.2             gtable_0.1.2
[21] futile.logger_1.4.1       latticeExtra_0.6-26
[23] biomaRt_2.25.1            proto_0.3-10
[25] Rcpp_0.11.6               acepack_1.3-3.3
[27] KernSmooth_2.23-14        BSgenome_1.37.2
[29] scales_0.2.5              Hmisc_3.16-0
[31] gridExtra_0.9.1           Rsamtools_1.21.8
[33] ggplot2_1.0.1             digest_0.6.8
[35] stringi_0.4-1             biovizBase_1.17.1
[37] tools_3.2.0               bitops_1.0-6
[39] magrittr_1.5              RCurl_1.95-4.6
[41] dichromat_2.0-0           Formula_1.2-1
```

```
[43] cluster_2.0.2            futile.options_1.0.0
[45] MASS_7.3-41              rpart_4.1-9
[47] nnet_7.3-9               GenomicAlignments_1.5.9
```

8.3 References

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.

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.

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

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.

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.

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

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

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.

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.

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.

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.

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.

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.

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.

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.

B. Pal, T. Bouras, W. Shi, F. Vaillant, J. M. Sheridan, N. Fu, K. Breslin, K. Jiang, M. E. Ritchie, M. Young, G. J. Lindeman, G. K. Smyth, and J. E. Visvader. Global changes in the mammary epigenome are induced by hormonal cues and coordinated by Ezh2. *Cell Rep.*, 3(2):411–426, Feb 2013.

B. Phipson, S. Lee, I. J. Majewski, W. S. Alexander, and G. K. Smyth. Empirical Bayes in the presence of exceptional cases, with application to microarray data. Technical report, Bioinformatics Division, Walter and Eliza Hall Institute of Medical Research, 2013.

R. Revilla-I-Domingo, I. Bilic, B. Vilagos, H. Tagoh, A. Ebert, I. M. Tamir, L. Smeenk, J. Trupke, A. Sommer, M. Jaritz, and M. Busslinger. The B-cell identity factor Pax5 regulates distinct transcriptional programmes in early and late B lymphopoiesis. *EMBO J.*, 31(14):3130–3146, 2012.

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

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.

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.

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

G. K. Smyth. Linear models and empirical bayes methods for assessing differential expression in microarray experiments. *Stat. Appl. Genet. Mol. Biol.*, 3:Article3, 2004.

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.

J. A. Zhang, A. Mortazavi, B. A. Williams, B. J. Wold, and E. V. Rothenberg. Dynamic transformations of genome-wide epigenetic marking and transcriptional control establish T cell identity. *Cell*, 149(2):467–482, Apr 2012.

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.