Package ‘csaw’

October 9, 2015

Version 1.2.1

Date 2015/05/15

Title ChIP-seq analysis with windows

Author Aaron Lun <alun@wehi.edu.au>, Gordon Smyth <smyth@wehi.edu.au>

Maintainer Aaron Lun <alun@wehi.edu.au>

Depends R (>= 3.2.0), GenomicRanges

Imports Rsamtools, edgeR, limma, GenomicFeatures, AnnotationDbi, methods, GenomicAlignments, S4Vectors, IRanges, GenomeInfoDb

Suggests org.Mm.eg.db, TxDb.Mmusculus.UCSC.mm10.knownGene

biocViews MultipleComparison, ChIPSeq, Normalization, Sequencing, Coverage, Genetics, Annotation

Description Detection of differentially bound regions in ChIP-seq data with sliding windows, with methods for normalization and proper FDR control.

License GPL-3

NeedsCompilation yes

R topics documented:

combineTests 2
consolidateSizes 4
correlateReads 6
csawUsersGuide 8
detailRanges 9
dumpPE 11
extractReads 13
filterWindows 14
findMaxima 16
getBestTest 18
getPESizes 19
getWidths 21
makeExtVector 22

mergeWindows 23
normalizeCounts 25
overlapStats 27
Parameter list methods 29
profileSites 30
readParam 32
regionCounts 35
scaledAverage 36
SEmethods 38
strandedCounts 39
upweightSummit 40
windowCounts 42
wwhm 44

Index **46**

`combineTests` *Combine statistics across multiple tests*

Description

Combines p-values across clustered tests using Simes' method to control the cluster FDR.

Usage

```
combineTests(ids, tab, weight=NULL, pval.col=NULL, fc.col=NULL)
```

Arguments

`ids`	an integer vector containing the cluster ID for each test
`tab`	a dataframe of results with `PValue`, `logCPM` and at least one `logFC` field for each test
`weight`	a numeric vector of weights for each window, defaults to 1 for each test
`pval.col`	an integer scalar specifying the column of `tab` containing the p-values, or a character string containing the name of that column
`fc.col`	an integer vector specifying the columns of `tab` containing the log-fold changes, or a character vector containing the names of those columns

Details

This function uses Simes' procedure to compute the combined p-value for each cluster of tests with the same value of `ids`. Each combined p-value represents evidence against the global null hypothesis, i.e., all individual nulls are true in each cluster. This may be more relevant than examining each test individually when multiple tests in a cluster represent parts of the same underlying event, e.g., genomic regions consisting of clusters of windows. The BH method is also applied to control the FDR across all clusters.

The importance of each test within a cluster can be adjusted by supplying different relative `weight` values. This may be useful for downweighting low-confidence tests, e.g., those in repeat regions. In Simes' procedure, weights are interpreted as relative frequencies of the tests in each cluster. Note that these weights have no effect between clusters and will not be used to adjust the computed FDR.

By default, the relevant fields in `tab` are identified by matching the column names to their expected values. Multiple fields in `tab` containing the `logFC` substring are allowed, e.g., to accommodate ANOVA-like contrasts. If the column names are different from what is expected, specification of the correct columns can be performed using `pval.col` and `fc.col`. This will overwrite any internal selection of the appropriate fields.

This function will report the number of windows with log-fold changes above 0.5 and below -0.5, to give some indication of whether binding increases or decreases in the cluster. If a cluster contains non-negligible numbers of `up` and `down` windows, this indicates that there may be a complex DB event within that cluster. Similarly, complex DB may be present if the total number of windows is larger than the number of windows in either category (i.e., change is not consistent across the cluster). Note that the threshold of 0.5 is arbitrary and has no impact on the significance calculations.

A simple clustering approach for windows is provided in `mergeWindows`. However, anything can be used so long as it does not compromise type I error control, e.g., promoters, gene bodies, independently called peaks.

Value

A dataframe with one row per cluster and the numeric fields `PValue`, the combined p-value; and `FDR`, the q-value corresponding to the combined p-value. An integer field `nWindows` specifies the total number of windows in each cluster. There are also two integer fields `*.up` and `*.down` for each log-FC column in `tab`, containing the number of windows with log-FCs above 0.5 or below -0.5, respectively. The name of each row represents the ID of the corresponding cluster.

Author(s)

Aaron Lun

References

Simes RJ (1986). An improved Bonferroni procedure for multiple tests of significance. *Biometrika* 73, 751-754.

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

Benjamini Y and Hochberg Y (1997). Multiple hypotheses testing with weights. *Scand. J. Stat.* 24, 407-418.

Lun ATL and Smyth GK (2014). De novo detection of differentially bound regions for ChIP-seq data using peaks and windows: controlling error rates correctly. *Nucleic Acids Res.* 42, e95

See Also

`mergeWindows`

Examples

```
ids <- round(runif(100, 1, 10))
tab <- data.frame(logFC=rnorm(100), logCPM=rnorm(100), PValue=rbeta(100, 1, 2))
combined <- combineTests(ids, tab)
head(combined)

# With window weighting.
w <- round(runif(100, 1, 5))
combined <- combineTests(ids, tab, weight=w)
head(combined)

# With multiple log-FCs.
tab$logFC.whee <- rnorm(100, 5)
combined <- combineTests(ids, tab)
head(combined)

# Manual specification of column IDs.
combined <- combineTests(ids, tab, fc.col=c(1,4), pval.col=3)
head(combined)

combined <- combineTests(ids, tab, fc.col="logFC.whee", pval.col="PValue")
head(combined)
```

`consolidateSizes` *Consolidate window sizes*

Description

Consolidate DB results from multiple window sizes.

Usage

```
consolidateSizes(data.list, result.list, equiweight=TRUE,  merge.args=list(tol=1000),
    combine.args=list(), region=NULL, overlap.args=list())
```

Arguments

`data.list`	a list of `SummarizedExperiment` objects, produced by `windowCounts`
`result.list`	a list of data frames containing the DB test results for each entry of `data.list`
`equiweight`	a logical scalar indicating whether equal weighting from each window size should be enforced
`merge.args`	a list of parameters to pass to `mergeWindows` when `FUN=NULL`
`combine.args`	a list of parameters to pass to `combineTests`
`region`	a `GRanges` object specifying regions of interest for overlapping with windows
`overlap.args`	a list of parameters to pass to `findOverlaps`

Details

This function consolidates DB results from multiple window sizes, to provide comprehensive detection of DB at a range of spatial resolutions. `SummarizedExperiment` objects can be generated by running `windowCounts` at a range of window sizes. Windows of all sizes are clustered together through `mergeWindows`, and the p-values from all windows in each cluster are combined using `combineTests`.

Some effort is required to equalize the contribution of each window size to the combined p-value of each cluster. This is done by setting `equiweight=TRUE`, where the weight of each window is inversely proportional to the number of windows of that size. Otherwise, the combined p-value would be determined by numerous small windows in each cluster.

If `region` is specified, each entry of `region` is defined as a cluster. Windows in each cluster are identified using `findOverlaps`, and consolidation is performed across multiple window sizes like before. Note that the returned `id` will be a list of `Hits` objects rather than integer vectors, as one window (subject) may overlap more than one region (query).

Value

A list is returned, containing:

`id`	a list of integer vectors, where each vector corresponds to an object in `data.list`; the entries of the vector specify the cluster to which each row of that object is assigned
`region`	a `GRanges` object containing the genomic coordinates of the clusters of merged windows (or other regions, if `region` is specified)
`table`	a data frame containing the combined DB results for each `region`

Author(s)

Aaron Lun

See Also

`windowCounts`, `mergeWindows`, `findOverlaps`, `combineTests`

Examples

```
bamFiles <- system.file("exdata", c("rep1.bam", "rep2.bam"), package="csaw")
low <- windowCounts(bamFiles, width=1, filter=1)
med <- windowCounts(bamFiles, width=100, filter=1)
high <- windowCounts(bamFiles, width=500, filter=1)

# Making up some DB results.
dbl <- data.frame(logFC=rnorm(nrow(low)), PValue=runif(nrow(low)), logCPM=0)
dbm <- data.frame(logFC=rnorm(nrow(med)), PValue=runif(nrow(med)), logCPM=0)
dbh <- data.frame(logFC=rnorm(nrow(high)), PValue=runif(nrow(high)), logCPM=0)

# Consolidating.
cons <- consolidateSizes(list(low, med, high), list(dbl, dbm, dbh),
merge.args=list(tol=100, max.width=300))
```

```
cons$region
cons$id
cons$table

# Without weights.
cons <- consolidateSizes(list(low, med, high), list(dbl, dbm, dbh),
merge.args=list(tol=100, max.width=300), equiweight=FALSE)
cons$table

# Trying with a custom region.
of.interest <- GRanges(c('chrA', 'chrA', 'chrB', 'chrC'),
    IRanges(c(1, 500, 100, 1000), c(200, 1000, 700, 1500)))
cons <- consolidateSizes(list(low, med, high), list(dbl, dbm, dbh),
    region=of.interest)
cons$table
cons$id

# Trying with limited numbers of overlaps; empty regions are ignored.
cons <- consolidateSizes(list(low[1,], med[1,], high[1,]),
    list(dbl[1,], dbm[1,], dbh[1,]), region=of.interest)
cons$region
cons$table
```

correlateReads *Compute correlation coefficients between reads*

Description

Computes the auto- or cross-correlation coefficients between read positions across a set of delay intervals.

Usage

```
correlateReads(bam.files, max.dist=1000, cross=TRUE, param=readParam())
```

Arguments

bam.files	a character vector containing paths to sorted and indexed BAM files
max.dist	integer scalar specifying the maximum delay distance over which correlation coefficients will be calculated
cross	a logical scalar specifying whether cross-correlations should be computed
param	a readParam object containing read extraction parameters, or a list of such objects (one for each BAM file)

Details

If `cross=TRUE`, reads are separated into those mapping on the forward and reverse strands. Positions on the forward strand are shifted forward by a delay interval. The chromosome-wide correlation coefficient between the shifted forward positions and the original reverse positions are computed. This is repeated for all delay intervals less than `maxDist`. A weighted mean for the cross-correlation is taken across all chromosomes, with weighting based on the number of reads.

Cross-correlation plots can be used to check the quality of immunoprecipitation for ChIP-Seq experiments involving transcription factors or punctate histone marks. Strong immunoprecipitation should result in a peak at a delay corresponding to the fragment length. A spike may also be observed at the delay corresponding to the read length. This is probably an artefact of the mapping process where unique mapping occurs to the same sequence on each strand.

The construction of cross-correlation plots is usually uninformative for full paired-end data. This is because the presence of valid pairs will inevitably result in a strong peak at the fragment length. Nonetheless, immunoprecipitation efficiency can be diagnosed by treating paired-end data as single end data. This is done by using only the first or second read based on the value of `pe` used in `readParam`. Setting `pe="both"` will result in failure.

If multiple BAM files are specified in `bam.files`, the reads from all libraries are pooled prior to calculation of the correlation coefficients. This is convenient for determining the average correlation profile across an entire dataset. Separate calculations for each file will require multiple calls to `correlateReads`.

If `cross=FALSE`, auto-correlation coefficients are computed without use of strand information. This is designed to guide estimation of the average width of enrichment for diffuse histone marks. For example, the width can be defined as the delay distance at which the autocorrelations become negligble. However, this tends to be ineffective in practice as diffuse marks tend to have very weak correlations to begin with.

By default, marked duplicate reads are removed from each BAM file prior to calculation of coefficients. This is strongly recommended, even if the rest of the analysis will be performed with duplicates retained. Otherwise, the read length spike will dominate the plot. The fragment length peak will no longer be easily visible.

Value

A numeric vector of length `max.dist+1` containing the correlation coefficients for each delay interval from 0 to `max.dist`.

Author(s)

Aaron Lun

References

Kharchenko PV, Tolstorukov MY and Park, PJ (2008). Design and analysis of ChIP-seq experiments for DNA-binding proteins. *Nat. Biotechnol.* 26, 1351-1359.

See Also

`ccf`

Examples

```
n <- 20
bamFile <- system.file("exdata", "rep1.bam", package="csaw")
par(mfrow=c(2,2))

x <- correlateReads(bamFile, max.dist=n)
plot(0:n, x, xlab="delay (bp)", ylab="ccf")

x <- correlateReads(bamFile, max.dist=n, param=readParam(dedup=TRUE))
plot(0:n, x, xlab="delay (bp)", ylab="ccf")

x <- correlateReads(bamFile, max.dist=n, cross=FALSE)
plot(0:n, x, xlab="delay (bp)", ylab="acf")
```

---

csawUsersGuide *View csaw user's guide*

Description

Finds the location of the user's guide and opens it for viewing.

Usage

```
csawUsersGuide(view=TRUE)
```

Arguments

view	logical scalar specifying whether the document should be opened

Details

The csaw package is designed for de novo detection of differentially bound regions from ChIP-seq data. It provides methods for window-based counting, normalization, filtering and statistical analyses via edgeR. The user guide for this package can be obtained by running this function.

For non-Windows operating systems, the PDF viewer is taken from `Sys.getenv("R_PDFVIEWER")`. This can be changed to `x` by using `Sys.putenv(R_PDFVIEWER=x)`. For Windows, the default viewer will be selected to open the file.

Note that the user's guide is not a true vignette as it is not generated using `Sweave` when the package is built. This is due to the time-consuming nature of the code when run on realistic case studies.

Value

A character string giving the file location. If `view=TRUE`, the system's default PDF document reader is started and the user's guide is opened.

Author(s)

Aaron Lun

See Also

system

Examples

```
# To get the location:
csawUsersGuide(view=FALSE)
# To open in pdf viewer:
## Not run: csawUsersGuide()
```

`detailRanges` *Add annotation to ranges*

Description

Add detailed exon-based annotation to specified genomic regions.

Usage

```
detailRanges(incoming, txdb, orgdb, dist=5000, promoter=c(3000, 1000),
    max.intron=1e6, key.field="ENTREZID", name.field="SYMBOL")
```

Arguments

`incoming`	a `GRanges` object containing the ranges to be annotated
`txdb`	a `TranscriptDb` object for the genome of interest
`orgdb`	a genome wide annotation object for the genome of interest
`dist`	an integer scalar specifying the flanking distance to annotate
`promoter`	an integer vector of length 2, where first and second values define the promoter as some distance upstream and downstream from the TSS, respectively
`max.intron`	an integer scalar indicating the maximum distance between exons
`key.field`	a character scalar specifying the keytype for name extraction
`name.field`	a character scalar or vector specifying the column(s) to use as the gene name

Details

This function adds exon-based annotations to a given set of genomic regions, in the form of compact character strings specifying the features overlapping and flanking each region. The aim is to determine the genic context of empirically identified regions. This allows some basic biological interpretation of binding/marking in those regions. All neighbouring genes within a specified range are reported, rather than just the closest gene to the region.

For annotated features overlapping a region, the character string in the `overlap` output vector will be of the form `GENE|EXONS|STRAND`. `GENE` is the gene symbol by default, but reverts to `<XXX>` if no symbol is defined for a gene with the Entrez ID `XXX`. The `EXONS` indicate the exon or range of exons that are overlapped. The `STRAND` is, obviously, the strand on which the gene is coded. For annotated regions flanking the region within a distance of `dist`, the character string in the `left` or `right` output vectors will have an additional `DIST` value. This represents the gap between the edge of the region and the closest exon for that gene. In all cases, any strandedness in `incoming` is ignored.

Exons are numbered in order of increasing start or end position for genes on the forward or reverse strands, respectively. Promoters are defined as the region of length `promoter` upstream of the gene TSS, itself defined as the start of the first exon (for genes on the forward strand) or the end of the last exon (otherwise). All promoters are marked as exon 0 for simplicity. Exon ranges in `EXON` are reported from as a comma-separated list where stretches of consecutive exons are summarized into a range. If the region overlaps an intron, it is labelled with `I` in `EXON`. No intronic overlaps are reported if there is an exonic overlap.

Note that promoter and intronic annotations are only reported for the `overlap` vector to reduce redundancy in the output. For example, it makes little sense to report that the region is both flanking and overlapping an intron. Similarly, the value of `DIST` is more relevant when it is reported to the nearest exon rather than to an intron (in which case, the distance would be zero if the intron overlaps the region). In cases where the distance is reported to the first exon, it can be used to refine the choice of `promoter`.

The `max.intron` value is necessary to deal with genes that have ambiguous locations on the genome. If a gene has exons on different chromosomes, its location is uncertain and the gene is partitioned into two sets of exons for separate processing. However, this is less obvious when the ambiguous locations belong to the same chromosome. The `max.intron` value protects against excessively large genes that may occur from considering those locations as a single transcriptional unit. Exons are partitioned into two (or more) internal groupings for further processing.

If `incoming` is missing, then the annotation will be provided directly to the user in the form of a `GRanges` object. This may be more useful when further work on the annotation is required. Exon numbers are provided in the metadata with promoters and gene bodies labelled as 0 and -1, respectively. Overlaps to introns can be identified by finding those regions that overlap with gene bodies but not with any of the corresponding exons.

The default settings for `key.field` and `name.field` will work for human and mouse genomes, but may not work for other organisms. The `key.field` should refer to the key type in the `OrgDb` object, and also correspond to the `GENEID` of the `TxDb` object. For example, in S. cerevisiae, `key.field` is set to `"ORF"` while `name.field` is set to `"GENENAME"`. If multiple entries are supplied in `name.field`, the value of `GENE` is defined as a semicolon-separated list of each of those entries.

Value

If `incoming` is not provided, a `GRanges` object will be returned containing ranges for the exons, promoters and gene bodies. Gene keys (e.g., Entrez IDs) are stored as row names. Gene symbols, exon numbers and internal groupings (for exons of genes with multiple genomic locations) are also stored as metadata.

If `incoming` is a `GRanges` object, a list will be returned with `overlap`, `left` and `right` elements. Each element is a character vector of length equal to the number of ranges in `incoming`. Each non-empty string records the gene symbol, the overlapped exons and the strand. For `left` and `right`, the gap between the range and the annotated feature is also included.

Author(s)

Aaron Lun

Examples

```
require(org.Mm.eg.db)
require(TxDb.Mmusculus.UCSC.mm10.knownGene)

current <- readRDS(system.file("exdata", "exrange.rds", package="csaw"))
output <- detailRanges(current, orgdb=org.Mm.eg.db,
    txdb=TxDb.Mmusculus.UCSC.mm10.knownGene)
head(output$overlap)
head(output$right)
head(output$left)

detailRanges(txdb=TxDb.Mmusculus.UCSC.mm10.knownGene, orgdb=org.Mm.eg.db)

## Not run:
output <- detailRanges(current, txdb=TxDb.Mmusculus.UCSC.mm10.knownGene,
    orgdb=org.Mm.eg.db, name.field=c("ENTREZID"))
head(output$overlap)

output <- detailRanges(current, txdb=TxDb.Mmusculus.UCSC.mm10.knownGene,
    orgdb=org.Mm.eg.db, name.field=c("SYMBOL", "ENTREZID"))
head(output$overlap)

## End(Not run)
```

---

dumpPE *Dump paired-end data to file*

Description

Extract proper pairs from a BAM file, and dump fragment intervals into another BAM file.

Usage

```
dumpPE(bam.file, prefix, param=readParam(pe="both"), overwrite=FALSE)
```

Arguments

`bam.file`	a character string containing the path to a paired-end BAM file
`prefix`	a character string containing the prefix to an output BAM file
`param`	a `readParam` object
`overwrite`	a logical scalar indicating whether to overwrite the existing file starting with `prefix`

Details

This function extracts proper pairs from `bam.file` according to the settings in `param`. It then generates another output BAM file that stores fragment information for each proper pair. Each alignment entry represents the forward-stranded read of a valid fragment. Fragment data can be extracted as the read position and insert size.

The idea is to generate a pre-filtered BAM file by specifying the appropriate settings in `param`. The output BAM file can then be efficiently analyzed with, e.g., `windowCounts` with `fast.pe=TRUE`. This avoids the need to load and match read names in every function. Note that all alignment-specific information is lost, e.g., MAPQ scores, duplicate information, CIGAR strings.

Value

A sorted and indexed BAM file is produced at the specified location. A character string containing the full name of the output BAM file is silently returned.

See Also

`readParam`, `windowCounts`

Examples

```
# Loading PE data.
bamFile <- system.file("exdata", "pet.bam", package="csaw")

xparam <- readParam(pe="both")
out <- windowCounts(bamFile, param=xparam, filter=1)

outBam <- dumpPE(bamFile, "whee", param=xparam)
out2 <- windowCounts(outBam, param=reform(xparam, fast.pe=TRUE), filter=1)

stopifnot(identical(assay(out), assay(out2)))
stopifnot(identical(out$totals, out2$totals))
unlink("whee.bam")

# Comparing it to a more complex scenario.
xparam <- readParam(pe="both", rescue.ext=200)
out <- windowCounts(bamFile, param=xparam, filter=1)

outBam <- dumpPE(bamFile, "whee", param=xparam)
out2 <- windowCounts(outBam, param=reform(xparam, fast.pe=TRUE), filter=1)

stopifnot(identical(assay(out), assay(out2)))
stopifnot(identical(out$totals, out2$totals))

# Looking at extracted reads.
last.reg <- rowRanges(out)[6]
reg1 <- extractReads(last.reg, bamFile, param=xparam)
reg2 <- extractReads(last.reg, outBam, param=reform(xparam, fast.pe=TRUE))

stopifnot(identical(sort(reg1), sort(reg2)))
unlink("whee.bam")
```

`extractReads` *Extract reads from a BAM file*

Description

Extract reads from a BAM file with the specified parameter settings.

Usage

```
extractReads(cur.region, bam.file, ext=NA, param=readParam())
```

Arguments

`cur.region`	a `GRanges` object of length 1 describing the region of interest
`bam.file`	a character string containing the path to a sorted and indexed BAM file
`ext`	an integer scalar or vector, containing the fragment lengths for directional read extension
`param`	a `readParam` object specifying how reads should be extracted

Details

This function extracts the reads from a BAM file overlapping a given genomic interval. The interpretation of the values in `param` is the same as that throughout the package. The aim is to supply the raw data for visualization, in a manner that maintains consistency with the rest of the analysis.

Note that this does not account for any read extension that might have been performed during read counting. In such cases, users are advised to expand `cur.region` by the extension length on each side. Counted reads can then be extracted by identifying their extended counterparts that overlap with the original `cur.region`.

Any strandedness of `cur.region` is ignored. If strand-specific extraction is desired, this can be done by setting `param$forward` via `reform`. Alternatively, the returned `GRanges` can be filtered to retain only the desired strand.

If `ext` is not `NA`, directional read extension will be performed for single-end data. See `windowCounts` for more details.

Value

A `GRanges` object is returned. If `pe="both"` in `param`, intervals are unstranded and correspond to fragments. Otherwise, strand-specific intervals that represent reads are returned.

Author(s)

Aaron Lun

See Also

`readParam`, `windowCounts`

Examples

```
bamFile <- system.file("exdata", "rep1.bam", package="csaw")
extractReads(GRanges("chrA", IRanges(100, 500)), bamFile)
extractReads(GRanges("chrA", IRanges(100, 500)), bamFile, param=readParam(dedup=TRUE))

bamFile <- system.file("exdata", "pet.bam", package="csaw")
extractReads(GRanges("chrB", IRanges(100, 500)), bamFile)
extractReads(GRanges("chrB", IRanges(100, 500)), bamFile, param=readParam(pe="both"))
extractReads(GRanges("chrB", IRanges(100, 500)), bamFile, param=readParam(pe="first"))

# Dealing with the extension length.
bamFile <- system.file("exdata", "rep1.bam", package="csaw")
ext <- 100
my.reg <- GRanges("chrA", IRanges(10, 200))
my.reg2 <- resize(my.reg, fix="center", width=width(my.reg)+2*ext)
collected <- extractReads(my.reg2, bamFile)

expanded <- resize(collected, width=ext)
strand(expanded) <- "*"
relevant <- overlapsAny(expanded, my.reg)
collected[relevant,]
```

`filterWindows` *Filtering methods for SummarizedExperiment objects*

Description

Convenience function to compute filter statistics for windows, based on proportions or using enrichment over background.

Usage

```
filterWindows(data, background, type="global", prior.count=2)
```

Arguments

`data`	a `SummarizedExperiment` object containing window- or bin-level counts
`background`	another `SummarizedExperiment` object, containing counts for background regions when `type!="proportion"`
`type`	a character string specifying the type of filtering to perform; can be any of `c("global", "local", "control", "proportion")`
`prior.count`	a numeric scalar, specifying the prior count to use in `aveLogCPM`

Details

Proportion-based filtering supposes that a certain percentage of the genome is genuinely bound. If `type="proportion"`, the filter statistic is defined as the ratio of the rank to the total number of windows. Rank is in ascending order, i.e., higher abundance windows have higher ratios. Windows are retained that have rank ratios above a threshold, e.g., 0.99 if 1% of the genome is assumed to be bound.

All other values of `type` will perform background-based filtering, where abundances of the windows are compared to those of putative background regions. The filter statistic are generally defined as the difference between window and background abundances, i.e., the log-fold increase in the counts. Windows can be filtered to retain those with large filter statistics, to select those that are more likely to contain genuine binding sites. The differences between the methods center around how the background abundances are obtained for each window.

If `type="global"`, the median average abundance across the genome is used as a global estimate of the background abundance. This should be used when `background` contains unfiltered counts for large (2 - 10 kbp) genomic bins, from which the background abundance can be computed. The filter statistic for each window is defined as the difference between the window abundance and the global background. If `background` is not supplied, the background abundance is directly computed from entries in `data`.

If `type="local"`, the counts of each row in `data` are subtracted from those of the corresponding row in `background`. The average abundance of the remaining counts is computed and used as the background abundance. The filter statistic is defined by subtracting the background abundance from the corresponding window abundance for each row. This is designed to be used when `background` contains counts for expanded windows, to determine the local background estimate.

If `type="control"`, the background abundance is defined as the average abundance of each row in `background`. The filter statistic is defined as the difference between the average abundance of each row in `data` and that of the corresponding row in `background`. This is designed to be used when `background` contains read counts for each window in the control sample(s). Unlike `type="local"`, there is no subtraction of the counts in `background` prior to computing the average abundance.

Value

A list is returned with `abundances`, the average abundance of each entry in `data`; `filter`, the filter statistic for the given `type`; and, for `type!="proportion"`, `back.abundances`, the average abundance of each entry in `background`.

Additional details

Proportion and global background filtering are dependent on the total number of windows/bins in the genome. However, empty windows or bins are automatically discarded in `windowCounts` (exacerbated if `filter` is set above unity). This will result in underestimation of the rank or over-estimation of the global background. To avoid this, the total number of windows or bins is inferred from the spacing.

For background-based methods, the abundances of large bins or regions in `background` must be rescaled for comparison to those of smaller windows - see `getWidths` and `scaledAverage` for more details. In particular, the effective width of the window is often larger than `width`, due to the counting of fragments rather than reads. The fragment length is extracted from `data$ext` and background$ext, though users will need to set data$rlen or background$rlen for unextended reads (i.e., ext=NA).

The prior.count protects against inflated log-fold increases when the background counts are near zero. A low prior is sufficient if background has large counts, which is usually the case for wide regions. Otherwise, prior.count should be increased to a larger value like 5. This may be necessary in type="control", where background contains counts for small windows in the control sample.

See Also

windowCounts, aveLogCPM, getWidths, scaledAverage

Examples

```
bamFiles <- system.file("exdata", c("rep1.bam", "rep2.bam"), package="csaw")
data <- windowCounts(bamFiles, filter=1)

# Proportion-based (keeping top 1%)
stats <- filterWindows(data, type="proportion")
head(stats$filter)
keep <- stats$filter > 0.99
new.data <- data[keep,]

# Global background-based (keeping fold-change above 3).
background <- windowCounts(bamFiles, bin=TRUE, width=300)
stats <- filterWindows(data, background, type="global")
head(stats$filter)
keep <- stats$filter > log2(3)

# Local background-based.
locality <- regionCounts(bamFiles, resize(rowRanges(data), fix="center", 300))
stats <- filterWindows(data, locality, type="local")
head(stats$filter)
keep <- stats$filter > log2(3)

# Control-based (pretend "rep.2" is a control library).
stats <- filterWindows(data[,1], data[,2], type="control", prior.count=5)
head(stats$filter)
keep <- stats$filter > log2(3)
```

---

findMaxima *Find local maxima*

Description

Find the local maxima for a given set of genomic regions.

Usage

```
findMaxima(regions, range, metric, ignore.strand=TRUE)
```

Arguments

`regions`	a `GRanges` object
`range`	an integer scalar specifying the range of surrounding regions to consider as local
`metric`	a numeric vector of values for which the local maxima is found
`ignore.strand`	a logical scalar indicating whether to consider the strandedness of `regions`

Details

For each region in `regions`, this function will examine all regions within `range` on either side. It will then determine if the current region has the maximum value of `metric` across this range. A typical metric to maximize might be the sum of counts or the average abundance across all libraries.

Preferably, `regions` should contain regularly sized and spaced windows or bins, e.g., from `windowCounts`. The sensibility of using this function for arbitrary regions is left to the user. In particular, the algorithm will not support nested regions and will fail correspondingly if any are detected.

If `ignore.strand=FALSE`, the entries in `regions` are split into their separate strands. The function is run separately on the entries for each strand, and the results are collated into a single output. This may be useful for strand-specific applications.

Value

A logical vector indicating whether each region in `regions` is a local maxima.

Author(s)

Aaron Lun

See Also

`windowCounts`, `aveLogCPM`

Examples

```
bamFiles <- system.file("exdata", c("rep1.bam", "rep2.bam"), package="csaw")
data <- windowCounts(bamFiles, filter=1)
regions <- rowRanges(data)
metric <- edgeR::aveLogCPM(asDGEList(data))
findMaxima(regions, range=10, metric=metric)
findMaxima(regions, range=50, metric=metric)
findMaxima(regions, range=100, metric=metric)

findMaxima(regions, range=10, metric=runif(length(regions)))
findMaxima(regions, range=50, metric=runif(length(regions)))
findMaxima(regions, range=100, metric=runif(length(regions)))
```

getBestTest *Get the best test in a cluster*

Description

Find the test with the strongest evidence for rejection of the null in each cluster.

Usage

```
getBestTest(ids, tab, by.pval=TRUE, weight=NULL, pval.col=NULL, cpm.col=NULL)
```

Arguments

`ids`	an integer vector containing the cluster ID for each test
`tab`	a table of results with a `PValue` field for each test
`by.pval`	a logical scalar, indicating whether selection should be performed on corrected p-values
`weight`	a numeric vector of weights for each window, defaults to 1 for each test
`pval.col`	an integer scalar specifying the column of `tab` containing the p-values, or a character string containing the name of that column
`cpm.col`	an integer scalar specifying the column of `tab` containing the log-CPM values, or a character string containing the name of that column

Details

Clusters are identified as those tests with the same value of `ids`. If `by.pval=TRUE`, this function identifies the test with the lowest p-value as that with the strongest evidence against the null in each cluster. The p-value of the chosen test is adjusted using the Bonferroni correction, based on the total number of tests in the parent cluster. This is necessary to obtain strong control of the family-wise error rate such that the best test can be taken from each cluster for further consideration.

The importance of each window in each cluster can be adjusted by supplying different relative `weight` values. Each weight is interpreted as a different threshold for each test in the cluster. Larger weights correspond to lower thresholds, i.e., less evidence is needed to reject the null for tests deemed to be more important. This may be useful for upweighting particular tests, e.g., windows containing a motif for the TF of interest.

Note the difference between this function and `combineTests`. The latter presents evidence for any rejections within a cluster. This function specifies the exact location of the rejection in the cluster, which may be more useful in some cases but at the cost of conservativeness. In both cases, clustering procedures such as `mergeWindows` can be used to identify the cluster.

If `by.pval=FALSE`, the best test is defined as that with the highest log-CPM value. This should be independent of the p-value so no adjustment is necessary. Weights are not applied here. This mode may be useful when abundance is correlated to rejection under the alternative hypothesis, e.g., picking high-abundance regions that are more likely to contain peaks.

By default, the relevant fields in `tab` are identified by matching the column names to their expected values. If the column names are different from what is expected, specification of the correct column can be performed using `pval.col` and `cpm.col`.

Value

A dataframe with one row per cluster and the numeric fields `best`, the index for the best test in the cluster; `PValue`, the (possibly adjusted) p-value for that test; and `FDR`, the q-value corresponding to the adjusted p-value. Note that the p-value column may be named differently if `pval.col` is specified. Other fields in `tab` corresponding to the best test inthe cluster are also returned. Cluster IDs are stored as the row names.

Author(s)

Aaron Lun

References

Genovese CR, Roeder K and Wasserman L (2006). False discovery control with p-value weighting. *Biometrika* 93, 509-524.

See Also

`combineTests`, `mergeWindows`

Examples

```
ids <- round(runif(100, 1, 10))
tab <- data.frame(logFC=rnorm(100), logCPM=rnorm(100), PValue=rbeta(100, 1, 2))
best <- getBestTest(ids, tab)
head(best)

best <- getBestTest(ids, tab, cpm.col="logCPM", pval.col="PValue")
head(best)

# With window weighting.
w <- round(runif(100, 1, 5))
best <- getBestTest(ids, tab, weight=w)
head(best)

# By logCPM.
best <- getBestTest(ids, tab, by.pval=FALSE)
head(best)

best <- getBestTest(ids, tab, by.pval=FALSE, cpm.col=2, pval.col=3)
head(best)
```

`getPESizes` *Compute fragment lengths for paired-end tags*

Description

Compute the length of the sequenced fragment for each read pair in paired-end tag (PE) data.

Usage

```
getPESizes(bam.file, param=readParam(pe="both"))
```

Arguments

`bam.file`	a character string containing the file path to a sorted and indexed BAM file
`param`	a `readParam` object containing read extraction parameters

Details

This function assembles a number of paired-end diagnostics. For starters, a read is only mapped if it is not removed by `dedup`, `minq`, `restrict` or `discard` in `readParam`. Otherwise, the alignment is not considered to be reliable. Any read pair with exactly one unmapped read is discarded, and the number of read pairs lost in this manner is recorded. Obviously, read pairs with both reads unmapped will be ignored completely.

Of the mapped pairs, the valid (i.e., proper) read pairs are identified. These refer to intrachromosomal read pairs where the reads with the lower and higher genomic coordinates map to the forward and reverse strand, respectively. The distance between the positions of the mapped 5' ends of the two reads must also be equal to or greater than the read lengths. Any intrachromosomal read pair that fails these criteria will be considered as improperly oriented. If the reads are on different chromosomes, the read pair will be recorded as being interchromosomal.

Each valid read pair corresponds to a DNA fragment where both ends are sequenced. The size of the fragment can be determined by calculating the distance between the 5' ends of the mapped reads. The distribution of sizes is useful for assessing the quality of the library preparation, along with all of the recorded diagnostics.

Value

A list containing:

`sizes`	an integer vector of fragment lengths for all valid read pairs in the library
`diagnostics`	an integer vector containing the total number of mapped reads, number of mapped singleton reads, pairs with exactly one unmapped read, number of improperly orientated read pairs and interchromosomal pairs

Author(s)

Aaron Lun

See Also

`readParam`

Examples

```
bamFile <- system.file("exdata", "pet.bam", package="csaw")
out <- getPESizes(bamFile, param=readParam(pe="both"))
out <- getPESizes(bamFile, param=readParam(pe="both", restrict="chrA"))
out <- getPESizes(bamFile, param=readParam(pe="both", discard=GRanges("chrA", IRanges(1, 50))))
```

getWidths *Get region widths*

Description

Get the widths of the read counting interval for each region.

Usage

```
getWidths(data)
```

Arguments

`data`	a `SummarizedExperiment` object, produced by `windowCounts` or `regionCounts`

Details

Widths of all regions are increased by the average fragment length during the calculations. This is because each count represents the number of (imputed) fragments overlapping each region. Thus, a 1 bp window has an effective width that includes the average length of each fragment.

The fragment length is taken from `exptData(data)$final.ext`, if it is not `NA`. Otherwise, it is taken from `data$ext`. For paired-end data, the average fragment length should be the median of the values obtained with `getPESizes`. If the fragment lengths are different between libraries, the average is used to computed the effective width of the window.

If `final.ext` is `NA` and any of `ext` are `NA`, the function will throw an error. Users should set the read length in `data$rlen` to avoid this, as `NA` values of `ext` correspond to the use of unextended reads.

Value

An integer vector containing the effective width, in base pairs, of each region.

Author(s)

Aaron Lun

See Also

`windowCounts`, `regionCounts`

Examples

```
bamFiles <- system.file("exdata", c("rep1.bam", "rep2.bam"), package="csaw")
data <- windowCounts(bamFiles, filter=1)
getWidths(data)

data <- windowCounts(bamFiles, ext=c(50, 100), filter=1)
getWidths(data)
```

```
data <- windowCounts(bamFiles, ext=makeExtVector(c(50, 100)), filter=1)
getWidths(data)

# Avoid error by defining 'rlen'.
data <- windowCounts(bamFiles, ext=c(NA, 100), filter=1)
try(getWidths(data))
data$rlen <- 200
getWidths(data)

# Paired-end data takes the fragment length from 'ext'.
bamFile <- system.file("exdata", "pet.bam", package="csaw")
data <- windowCounts(bamFile, param=readParam(pe="both"), ext=200, filter=1)
getWidths(data)
```

`makeExtVector` *Make an extVector object*

Description

Construct an extVector to hold read extension parameters.

Usage

```
makeExtVector(ext, final.ext=NULL)
```

Arguments

`ext`	an integer scalar or vector containing average fragment lengths for read extension in one or more libraries
`final.ext`	an integer scalar, specifying the length to which all fragments are scaled

Details

This function returns an `extVector` object, which is just an augmented integer vector with the `final.ext` attribute. The aim of using a specialized class is to preserve attributes during subsetting. Thus, functions like `windowCounts` or `regionCounts` can be run with different combinations of libraries, and `final.ext` will be maintained in each call.

The values in `ext` will ultimately be used for directional extension, to infer the fragment corresponding to each read. Different fragment lengths will be used in different libraries if a vector is supplied for `ext`. If `final.ext` is `NULL`, it is defined by `makeExtVector` as the mean of all `ext`.

Lengths of all fragments will be scaled to the same `final.ext` value in each library. If `final.ext` is `NA`, no scaling is performed. See `windowCounts` for more details.

Value

An `extVector` object containing the values of `ext`, with the value of `final.ext` stored in the attributes.

Author(s)

Aaron Lun

See Also

`windowCounts`, `regionCounts`

Examples

```
out <- makeExtVector(1:5*10, NA)
out
out[1:2]

out <- makeExtVector(1:5*10, NULL)
out
out[1:2] # Doesn't recalculate, which is correct.
```

`mergeWindows` *Merge windows into clusters*

Description

Uses a simple single-linkage approach to merge adjacent or overlapping windows into clusters.

Usage

```
mergeWindows(regions, tol, sign=NULL, max.width=NULL, ignore.strand=TRUE)
```

Arguments

`regions`	a `GRanges` object
`tol`	a numeric scalar specifying the maximum distance between adjacent windows
`sign`	a logical vector specifying whether each window has a positive log-FC
`max.width`	a numeric scalar specifying the maximum size of merged intervals
`ignore.strand`	a logical scalar indicating whether to consider the strandedness of `regions`

Details

Windows are merged if the gap between the end of one window and the start of the next is no greater than `tol`. Adjacent windows can then be chained together to build a cluster of windows across the linear genome. A value of zero for `tol` means that the windows must be contiguous whereas negative values specify minimum overlaps.

If `sign!=NULL`, windows are only merged if they have the same sign of the log-FC and are not separated by intervening windows with opposite log-FC values. This can be useful to ensure consistent changes when summarizing adjacent DB regions. However, it is not recommended for routine clustering in differential analyses as the resulting clusters will not be independent of the p-value.

Specification of `max.width` prevents the formation of excessively large clusters when many adjacent regions are present. Any cluster that is wider than `max.width` is split into multiple subclusters of (roughly) equal size. Specifically, the cluster interval is partitioned into the smallest number of equally-sized subintervals where each subinterval is smaller than `max.width`. Windows are then assigned to each subinterval based on the location of the window midpoints. Suggested values range from 2000 to 10000 bp, but no limits are placed on the maximum size if it is `NULL`.

The tolerance should reflect the minimum distance at which two regions of enrichment are considered separate. If two windows are more than `tol` apart, they *will* be placed into separate clusters. In contrast, the `max.width` value reflects the maximum distance at which two windows can be considered part of the same region.

Arbitrary regions can also be used in this function. However, caution is required if any fully nested regions are present. Clustering with `sign!=NULL` will throw an error as splitting by sign becomes undefined. Splitting with `max.width!=NULL` will not fail, but cluster sizes may not be reduced if very large regions are present.

If `ignore.strand=FALSE`, the entries in `regions` are split into their separate strands. The function is run separately on the entries for each strand, and the results collated. The `region` returned in the output will be stranded to reflect the strand of the contributing input regions. This may be useful for strand-specific applications.

Note that, in the output, the cluster ID reported in `id` corresponds to the index of the cluster coordinates in the input `region`.

Value

A list containing `id`, an integer vector containing the cluster ID for each window; and `region`, a `GRanges` object containing the start/stop coordinates for each cluster of windows.

Author(s)

Aaron Lun

See Also

`combineTests`, `windowCounts`

Examples

```
x <- round(runif(10, 100, 1000))
gr <- GRanges(rep("chrA", 10), IRanges(x, x+40))
mergeWindows(gr, 1)
mergeWindows(gr, 10)
mergeWindows(gr, 100)
mergeWindows(gr, 100, sign=rep(c(TRUE, FALSE), 5))
```

`normalizeCounts` *Normalize counts between libraries*

Description

Calculate normalization factors or offsets using count data from multiple libraries.

Usage

```
normalizeCounts(counts, lib.sizes, type=c("scaling", "loess"), weighted=FALSE, ...)
```

Arguments

`counts`	a matrix of integer counts with one column per library
`lib.sizes`	a numeric vector specifying the total number of reads per library
`type`	a character string indicating what type of normalization is to be performed
`weighted`	a logical scalar indicating whether precision weights should be used for TMM normalization
`...`	other arguments to be passed to `calcNormFactors` for `type="scaling"`, or `loessFit` for `type="loess"`

Details

If `type="scaling"`, this function provides a convenience wrapper for the `calcNormFactors` function in the `edgeR` package. Specifically, it uses the trimmed mean of M-values (TMM) method to perform normalization. Precision weighting is turned off by default so as to avoid upweighting high-abundance regions. These are more likely to be bound and thus more likely to be differentially bound. Assigning excessive weight to such regions will defeat the purpose of trimming when normalizing the coverage of background regions.

If `type="loess"`, this function performs non-linear normalization similar to the fast loess algorithm in `normalizeCyclicLoess`. For each sample, a lowess curve is fitted to the log-counts against the log-average count. The fitted value for each bin pair is used as the generalized linear model offset for that sample. The use of the average count provides more stability than the average log-count when low counts are present for differentially bound regions.

If `lib.sizes` is not specified, the column sums of `counts` are used instead and a warning is issued. The same `lib.sizes` should be used throughout the analysis if multiple `normalizeCounts` calls are involved. This ensures that the normalization factors or offsets are comparable between calls.

Value

For `type="scaling"`, a numeric vector containing the relative normalization factors for each library.

For `type="loess"`, a numeric matrix of the same dimensions as `counts`, containing the log-based offsets for use in GLM fitting.

Author(s)

Aaron Lun

References

Robinson MD, Oshlack A (2010). A scaling normalization method for differential expression analysis of RNA-seq data. *Genome Biology* 11, R25.

Ballman KV, Grill DE, Oberg AL, Therneau TM (2004). Faster cyclic loess: normalizing RNA arrays via linear models. *Bioinformatics* 20, 2778-86.

See Also

calcNormFactors, loessFit, normalizeCyclicLoess

Examples

```
# A trivial example
counts <- matrix(rnbinom(400, mu=10, size=20), ncol=4)
normalizeCounts(counts)
normalizeCounts(counts, lib.sizes=rep(400, 4))

# Adding undersampling
n <- 1000L
mu1 <- rep(10, n)
mu2 <- mu1
mu2[1:100] <- 100
mu2 <- mu2/sum(mu2)*sum(mu1)
counts <- cbind(rnbinom(n, mu=mu1, size=20), rnbinom(n, mu=mu2, size=20))
actual.lib.size <- rep(sum(mu1), 2)
normalizeCounts(counts, lib.sizes=actual.lib.size)
normalizeCounts(counts, logratioTrim=0.4, lib.sizes=actual.lib.size)
normalizeCounts(counts, sumTrim=0.3, lib.size=actual.lib.size)

# With and without weighting, for high-abundance spike-ins.
n <- 100000
blah <- matrix(rnbinom(2*n, mu=10, size=20), ncol=2)
tospike <- 10000
blah[1:tospike,1] <- rnbinom(tospike, mu=1000, size=20)
blah[1:tospike,2] <- rnbinom(tospike, mu=2000, size=20)
full.lib.size <- colSums(blah)

normalizeCounts(blah, weighted=TRUE, lib.sizes=full.lib.size)
normalizeCounts(blah, lib.sizes=full.lib.size)
true.value <- colSums(blah[(tospike+1):n,])/colSums(blah)
true.value <- true.value/exp(mean(log(true.value)))
true.value

# Using loess-based normalization, instead.
offsets <- normalizeCounts(counts, type="loess", lib.size=full.lib.size)
head(offsets)
offsets <- normalizeCounts(counts, type="loess", span=0.4, lib.size=full.lib.size)
offsets <- normalizeCounts(counts, type="loess", iterations=1, lib.size=full.lib.size)
```

`overlapStats` *Compute overlap statistics*

Description

Compute assorted statistics for overlaps between windows and regions in a Hits object.

Usage

```
combineOverlaps(olap, tab, o.weight=NULL, i.weight=NULL, ...)
getBestOverlaps(olap, tab, o.weight=NULL, i.weight=NULL, ...)
summitOverlaps(olap, region.best, o.summit=NULL, i.summit=NULL)
```

Arguments

`olap`	a `Hits` object produced by `findOverlaps`, containing overlaps between regions (query) and windows (subject)
`tab`	a dataframe of DE results for each window
`o.weight`	a numeric vector specifying weights for each overlapped window
`i.weight`	a numeric vector specifying weights for each individual window
`...`	other arguments to be passed to the wrapped functions
`region.best`	an integer vector specifying the window index that is the summit for each region
`o.summit`	a logical vector specifying the overlapped windows that are summits, or a corresponding integer vector of indices for such windows
`i.summit`	a ogical vector specifying whether an individual window is a summit, or a corresponding integer vector of indices

Details

These functions provide convenient wrappers around `combineTests`, `getBestTest` and `upweightSummit`. They accept `Hits` objects produced by running `findOverlaps` between windows and some pre-specified regions. Each set of windows overlapping a region is defined as a cluster to compute the various statistics.

A wrapper is necessary as a window may overlap multiple regions. If so, the multiple instances of that window are defined as distinct "overlapped" windows, where each overlapped window is assigned to a different region. Each overlapped window is represented by a row of `olap`. In contrast, the "individual" window just refers to the window itself, regardless of what it overlaps. This is represented by each row of the `SummarizedExperiment` object and the `tab` derived from it.

The distinction between these two definitions is required to describe the weight arguments. The `o.weight` argument refers to the weights for each region-window relationship. This allows for different weights to be assigned to the same window in different regions. The `i.weight` argument is the weight of the window itself, and is the same regardless of the region. If both are specified, `o.weight` takes precedence.

For summitOverlaps, the region.best argument is designed to accept the best field in the output of getBestOverlaps (run with by.pval=FALSE). This contains the index for the individual window that is the summit within each region. In contrast, the i.summit argument indicates whether an individual window is a summit, e.g., from findMaxima. The o.summit argument does the same for overlapped windows, though this has no obvious input within the csaw pipeline.

Value

For combineOverlaps and getBestOverlaps, a dataframe is returned from their respective wrapped functions. Each row of the dataframe corresponds to a region, where regions without overlapped windows are assigned NA values.

For summitOverlaps, a numeric vector of weights is produced. This can be used as o.weight in the other two functions.

Author(s)

Aaron Lun

See Also

combineTests, getBestTest, upweightSummit

Examples

```
bamFiles <- system.file("exdata", c("rep1.bam", "rep2.bam"), package="csaw")
data <- windowCounts(bamFiles, width=1, filter=1)
of.interest <- GRanges(c('chrA', 'chrA', 'chrB', 'chrC'),
    IRanges(c(1, 500, 100, 1000), c(200, 1000, 700, 1500)))

# Making some mock results.
N <- nrow(data)
mock <- data.frame(logFC=rnorm(N), PValue=runif(N), logCPM=rnorm(N))

olap <- findOverlaps(of.interest, rowRanges(data))
combineOverlaps(olap, mock)
getBestOverlaps(olap, mock)

# See what happens when you don't get many overlaps.
getBestOverlaps(olap[1,], mock)
combineOverlaps(olap[2,], mock)

# Weighting example, with window-specific weights.
window.weights <- runif(N)
comb <- combineOverlaps(olap, mock, i.weight=window.weights)

# Weighting example, with relation-specific weights.
best.by.ave <- getBestOverlaps(olap, mock, by.pval=FALSE)
w <- summitOverlaps(olap, region.best=best.by.ave$best)
head(w)
stopifnot(length(w)==length(olap))
combineOverlaps(olap, mock, o.weight=w)
```

```
# Running summitOverlaps for window-specific summits
# (output is still relation-specific weights, though).
is.summit <- findMaxima(rowRanges(data), range=100, metric=mock$logCPM)
w <- summitOverlaps(olap, i.summit=is.summit)
head(w)
```

Parameter list methods

Get or modify readParam lists

Description

Extract and modify lists of library-specific readParam objects

Usage

```
reformList(paramlist, ...)
checkList(paramlist)
```

Arguments

`paramlist`	a list of `readParam` objects
`...`	other arguments, to be passed to `reform`

Details

The `reformList` function sets the parameters specified in `...` across all elements of `paramlist`. This standardizes all of the internal `readParam` objects, e.g., for the `restrict` parameter. If `paramlist` is a single `readParam` object, then `reformList` is equivalent to `reform`.

The `checkList` function checks which parameter settings are identical between elements of `paramlist`. This is useful when some parameters must be variable, while others must be the same.

Value

For `reformList`, a list of library-specific `readParam` objects is returned.

For `checkList`, a character vector is returned containing all non-identical parameters across the list.

Author(s)

Aaron Lun

See Also

`windowCounts`, `regionCounts`, `reform`

Examples

```
bamFiles <- system.file("exdata", c("rep1.bam", "rep2.bam"), package="csaw")
data <- windowCounts(bamFiles, width=100, filter=1)
data$param

data <- windowCounts(bamFiles, width=100, filter=1, param=list(readParam(), readParam(minq=100)))
data$param
data[,1]$param
data[,2]$param

plist <- data$param
reformList(plist, restrict="chrA")
reformList(plist, minq=10)

checkList(plist)
checkList(reformList(plist, minq=10))
```

`profileSites` *Profile binding sites*

Description

Get the coverage profile around potential binding sites.

Usage

```
profileSites(bam.files, regions, range=5000, ext=100, weight=1,
    param=readParam(), use.strand=TRUE, match.strand=FALSE)
```

Arguments

`bam.files`	a character vector containing paths to one or more BAM files
`regions`	a `GRanges` object over which profiles are to be aggregated
`range`	an integer scalar specifying the range over which the profile will be collected
`ext`	an integer scalar specifying the average fragment length for single-end data
`weight`	a numeric vector indicating the relative weight to be assigned to the profile for each region
`param`	a `readParam` object containing read extraction parameters, or a list of such objects (one for each BAM file)
`use.strand`	a logical scalar indicating whether to consider the strandedness of `regions`
`match.strand`	a logical scalar indicating whether to match the strandedness of the regions to the reads

Details

This function aggregates the coverage profile around the specified regions. The shape of this profile can guide an intelligent choice of the window size in `windowCounts`, or to determine if region expansion is necessary in `regionCounts`. For the former, restricting the `regions` to locally maximal windows with `findMaxima` is recommended prior to use of `profileSites`. The function can be also used to examine average coverage around known features of interest, like genes.

The profile records the number of fragments overlapping each base within `range` of the start of all `regions`. Single-end reads are directionally extended to `ext` to impute the fragment (see `windowCounts` for more details). For paired-end reads, the interval between each pair is used as the fragment. If multiple `bam.files` are specified, reads are pooled across files for counting into each profile.

Direct aggregation will favour high-abundance regions as these have higher counts. If this is undesirable, high-abundance regions can be downweighted using the `weight` argument. For example, this can be set to the inverse of the sum of counts across all libraries for each region in `regions`. This will ensure that each region contributes equally to the final profile.

Value

A numeric vector of average coverages for each position within `range`, where the average is taken over all `regions`. If `weight` is set as described above for each region, then the vector will represent average relative coverages, i.e., relative to the number of fragments counted in the region itself.

Comments on strand specificity

If `use.strand=TRUE`, the function is recalled on the reverse-stranded regions. The profile for these regions is computed such that the left side of the profile corresponds to the upstream flank on the reverse strand (i.e., the profile is flipped). The center of the profile corresponds to the 5' end of the region on the reverse strand. This may be useful for features where strandedness is important, e.g., TSS's. Otherwise, if `use.strand=FALSE`, no special treatment is given to reverse-stranded features.

By default, the strandedness of the region has no effect on read extraction. If `match.strand=TRUE`, the profile for reverse-strand regions is made with reverse-strand reads only (this profile is also flipped, as described for `use.strand=TRUE`). Similarly, only forward-strand reads are used for forward- or unstranded regions. Note that `param$forward` must be set to `NULL` for this to work.

Author(s)

Aaron Lun

See Also

`findMaxima`, `windowCounts`, `wwhm`

Examples

```
bamFile <- system.file("exdata", "rep1.bam", package="csaw")
data <- windowCounts(bamFile, filter=1)
rwsms <- rowSums(assay(data))
maxed <- findMaxima(rowRanges(data), range=100, metric=rwsms)
```

```
x <- profileSites(bamFile, rowRanges(data)[maxed], range=200)
plot(as.integer(names(x)), x)

x <- profileSites(bamFile, rowRanges(data)[maxed], range=500)
plot(as.integer(names(x)), x)

x <- profileSites(bamFile, rowRanges(data)[maxed], range=500, weight=1/rwsms)
plot(as.integer(names(x)), x)

# Introducing some strandedness.
regs <- rowRanges(data)[maxed]
strand(regs) <- sample(c("-", "+", "*"), sum(maxed), replace=TRUE)
x <- profileSites(bamFile, regs, range=500)
plot(as.integer(names(x)), x)
x2 <- profileSites(bamFile, regs, range=500, use.strand=FALSE)
points(as.integer(names(x2)), x2, col="red")
x3 <- profileSites(bamFile, regs, range=500, match.strand=TRUE,
    param=readParam(forward=NULL))
points(as.integer(names(x3)), x3, col="blue")
```

`readParam` *readParam class and methods*

Description

Class to specify read loading parameters

Details

Each readParam object stores a number of parameters, each pertaining to the extraction of reads from a BAM file. Slots are defined as:

- `pe`: a character string indicating whether paired-end data is present; set to `"none"`, `"both"`, `"first"` or `"second"`
- `max.frag`: an integer scalar, specifying the maximum fragment length corresponding to a read pair
- `rescue.ext`: an integer scalar indicating the extension length for rescued reads from invalid pairs
- `fast.pe`: a logical scalar specifying whether fast fragment extraction should be performed for paired-end data
- `dedup`: a logical scalar indicating whether marked duplicate reads should be ignored
- `minq`: an integer scalar, specifying the minimum mapping quality score for an aligned read
- `forward`: a logical scalar indicating whether only forward reads should be extracted
- `restrict`: a character vector containing the names of allowable chromosomes from which reads will be extracted
- `discard`: a `GRanges` object containing intervals in which any alignments will be discarded

Removing low-quality or irrelevant reads

Marked duplicate reads will be removed with `dedup=TRUE`. This may be necessary when many rounds of PCR have been performed during library preparation. However, it is not recommended for routine counting as it will interfere with the downstream statistical methods. Note that the duplicate field must be set beforehand in the BAM file for this argument to have any effect.

Reads can also be filtered by their mapping quality scores if `minq` is specified at a non-`NA` value. This is generally recommended to remove low-confidence alignments. The exact threshold for `minq` will depend on the range of scores provided by the aligner. If `minq=NA`, no filtering on the score will be performed.

If `restrict` is supplied, reads will only be extracted for the specified chromosomes. This is useful to restrict the analysis to interesting chromosomes, e.g., no contigs/scaffolds or mitochondria. Conversely, if `discard` is set, a read will be removed if the corresponding alignment is wholly contained within the supplied ranges. This is useful for removing reads in repeat regions.

Parameter settings for paired-end data

For `pe="both"`, reads are extracted with the previously described filters, i.e., `discard`, `minq`, `dedup`. Extracted reads are then grouped into proper pairs. Proper pairs are those where the two reads are close together and in an inward-facing orientation. The fragment interval is defined as that bounded by the 5' ends of the two reads in a proper pair. Fragment sizes above `max.frag` are removed; use `getPESizes` to pick an appropriate value.

By default, only reads in proper pairs are used to construct a fragment interval. If `rescue.ext!=NA`, the function will also attempt to recover reads from improper pairs. For each improper pair, the read with the higher MAPQ score will be directionally extended with `rescue.ext`. Users should set `rescue.ext` as the median fragment length from `getPESizes`. The extended read will then be used as the fragment interval. Similarly, any reads without a mapped mate will be extended. For reasons of convenience, both reads will be rescued and extended for interchromosomal read pairs.

Each run through the position-sorted BAM file tends to be time-consuming, as read names must be extracted and matched. This can be avoided by setting `fast.pe=TRUE`. In this case, only cursory checks are done with regards to proper pairing. Each fragment is defined from a forward-stranded read with a reverse-stranded mate and a positive insert size no larger than `max.frag`. Values of `minq` and `discard` are ignored, as the specifics of the mate alignment are not accessed in this fast mode.

Finally, paired-end data can also be treated as single-end data by only using one read from each pair with `pe="first"` or `"second"`. This is useful for poor-quality data where the paired-end procedure has obviously failed, e.g., with many interchromosomal read pairs or pairs with large fragment lengths. Treating the data as single-end may allow the analysis to be salvaged.

In all cases, users should ensure that each BAM file containing paired-end data is properly synchronized prior to count loading.

Parameter settings for single-end data

If `pe="none"`, reads are assumed to be single-end. Read extraction from BAM files is performed with the same quality filters described above. If `forward=NA`, reads are extracted from all strands. Otherwise, reads are only extracted from the forward or reverse strands for `TRUE` or `FALSE`, respectively. This may be useful for applications requiring strand-specific counting. A special case is `forward=NULL` - see `strandedCounts` for more details.

Note that directional extension is often used in analyses of single-end data. However, it must be stressed that the `rescue.ext` parameter here does *not* determine the length of the extension. Instead, the average fragment length must be specified in each of the relevant downstream functions. This is because directional extension of single-end data pertains to analysis, not read extraction.

Constructor

`readParam(pe="none", max.frag=500, rescue.ext=NA, dedup=FALSE, minq=NA, forward=NA, restrict=NULL, d`
: Creates a readParam object. Each argument is placed in the corresponding slot, with coercion into the appropriate type.

Subsetting

In the code snippes below, `x` is a readParam object.

`x$name`: Returns the value in slot `name`.

Other methods

In the code snippes below, `x` is a readParam object.

`show(x)`: Describes the parameter settings in plain English.

`reform(x, ...)`: Creates a new readParam object, based on the existing `x`. Any named arguments in `...` are used to modify the values of the slots in the new object, with type coercion as necessary.

Author(s)

Aaron Lun

See Also

`windowCounts`, `regionCounts`, `correlateReads`, `getPESizes`

Examples

```
blah <- readParam()
blah <- readParam(discard=GRanges("chrA", IRanges(1, 10)))
blah <- readParam(restrict='chr2')
blah$pe
blah$dedup

# Use 'reform' if only some arguments need to be changed.
blah
reform(blah, dedup=TRUE)
reform(blah, rescue.ext=200)
reform(blah, pe="both", max.frag=212.0)
reform(blah, pe="both", fast.pe=TRUE)
```

`regionCounts` *Count reads overlapping each region*

Description

Count the number of extended reads overlapping pre-specified regions

Usage

```
regionCounts(bam.files, regions, ext=100, param=readParam())
```

Arguments

`bam.files`	a character vector containing paths to sorted and indexed BAM files
`regions`	a `GRanges` object containing the regions over which reads are to be counted
`ext`	an integer scalar or vector, describing the average length of the sequenced fragment in each library
`param`	a `readParam` object containing read extraction parameters, or a list of such objects (one for each BAM file)

Details

This function simply provides a wrapper around `countOverlaps` for read counting into specified regions. It is provided so as to allow for counting with awareness of the other parameters, e.g., `ext`, `pe`. This allows users to coordinate region-based counts with those from `windowCounts`. Checking that the output `totals` are the same between the two calls is strongly recommended.

Note that the strandedness of `regions` will not be considered when computing overlaps. The strandedness of the output `rowRanges` will depend on the strand(s) from which reads were counted. This is determined by the `forward` slot in the `param` object.

See `windowCounts` and `makeExtVector` for more details on read extension.

Value

A `SummarizedExperiment` object is returned containing one integer matrix. Each entry of the matrix contains the count for each library (column) at each region (row). The coordinates of each region are stored as the `rowRanges`. The total number of reads, read extension length and `param` used in each library are in the `colData`.

Author(s)

Aaron Lun

See Also

`countOverlaps`, `windowCounts`, `readParam`, `makeExtVector`

Examples

```
# A low filter is only used here as the examples have very few reads.
bamFiles <- system.file("exdata", c("rep1.bam", "rep2.bam"), package="csaw")
incoming <- GRanges(c('chrA', 'chrA', 'chrB', 'chrC'),
    IRanges(c(1, 500, 100, 1000), c(200, 1000, 700, 1500)))
regionCounts(bamFiles, regions=incoming)
regionCounts(bamFiles, regions=incoming, param=readParam(restrict="chrB"))

# Loading PE data.
bamFile <- system.file("exdata", "pet.bam", package="csaw")
regionCounts(bamFile, regions=incoming, param=readParam(pe="both"))
regionCounts(bamFile, regions=incoming, param=readParam(max.frag=100,
pe="first", restrict="chrA"))
regionCounts(bamFile, regions=incoming, param=readParam(max.frag=100,
pe="both", restrict="chrA", rescue.ext=100))
```

`scaledAverage` *Scaled average abundance*

Description

Compute the scaled average abundance for each feature.

Usage

```
scaledAverage(y, scale=1, prior.count=NULL, ...)
```

Arguments

`y`	a `DGEList` object
`scale`	a numeric vector indicating the magnitude with which each abundance is to be downscaled
`prior.count`	a numeric scalar specifying the prior count to add
`...`	other arguments, to be passed to `aveLogCPM`

Details

This function computes the average abundance of each feature in `y`, and downscales it according to `scale`. For example, if `scale=2`, the average count is halved, i.e., the returned abundances are decreased by 1 (as they are log2-transformed values). The aim is to set `scale` based on the relative width of regions, to allow abundances to be compared between regions of different size. Widths can be obtained using the `getWidths` function.

Some subtlety is necessary regarding the treatment of the `prior.count`. Specifically, the prior used in `aveLogCPM` is automatically increased when `scale` is larger. This ensures that the effective prior is the same after the abundance is scaled down. Otherwise, the use of the same prior would incorrectly result in a smaller abundance for larger regions.

Note that the adjustment for width assumes that reads are uniformly distributed throughout each region. This is reasonable for most background regions, but may not be for enriched regions. When the distribution is highly heterogeneous, the downscaled abundance of a large region will not be an accurate representation of the abundance of the smaller regions nested within.

For consistency, the `prior.count` is set to the default value of `aveLogCPM.DGEList`, if it is not otherwise specified. If a non-default value is used, make sure that it is the same for all calls to `scaledAverage`. This ensures that comparisons between the returned values are valid.

Value

A numeric vector of scaled abundances, with one entry for each row of `y`.

Author(s)

Aaron Lun

See Also

`getWidths`, `aveLogCPM`

Examples

```
bamFiles <- system.file("exdata", c("rep1.bam", "rep2.bam"), package="csaw")
size1 <- 50
data1 <- windowCounts(bamFiles, width=size1, filter=1)
size2 <- 500
data2 <- windowCounts(bamFiles, width=size2, filter=1)

# Adjusting by `scale`, based on median sizes.
head(scaledAverage(asDGEList(data1)))
relative <- median(getWidths(data2))/median(getWidths(data1))
head(scaledAverage(asDGEList(data2), scale=relative))

# Need to make sure the same prior is used, if non-default.
pc <- 5
head(scaledAverage(asDGEList(data1), prior.count=pc))
head(scaledAverage(asDGEList(data2), scale=relative, prior.count=pc))

# Different way to compute sizes, for 1-to-1 relations.
data3 <- regionCounts(bamFiles, regions=resize(rowRanges(data1),
    fix="center", width=size2))
head(scaledAverage(asDGEList(data1)))
relative.2 <- getWidths(data1)/getWidths(data2)
head(scaledAverage(asDGEList(data3), scale=relative.2))
```

SEmethods *Statistical wrappers for SummarizedExperiment objects*

Description

Convenience wrappers for statistical routines operating on SummarizedExperiment objects

Usage

```
normalize(object, ...)
asDGEList(object, ...)
```

Arguments

`object`	a `SummarizedExperiment` object, like that produced by `windowCounts`
`...`	other arguments to be passed to the function being wrapped

Details

Counts are extracted using the matrix corresponding to the first assay in the `SummarizedExperiment` object. The total library size is taken from the `totals` entry in the column data; warnings will be generated if this entry is not present. In the `normalize` method, the extracted counts and library sizes are supplied to `normalizeCounts`, along wth arguments in `...`. Similarly, the `asDGEList` method wraps the `DGEList` constructor.

Value

For `normalize`, either a numeric matrix or vector is returned; see `normalizeCounts`.

For `asDGEList`, a `DGEList` object is returned.

Author(s)

Aaron Lun

See Also

`normalizeCounts`, `DGEList`, `windowCounts`

Examples

```
bamFiles <- system.file("exdata", c("rep1.bam", "rep2.bam"), package="csaw")
data <- windowCounts(bamFiles, width=100, filter=1)
normalize(data)
head(normalize(data, type="loess"))

asDGEList(data)
asDGEList(data, norm.factors=c(1.11, 2.23), group=c("a", "b"))
```

strandedCounts *Get strand-specific counts*

Description

Obtain strand-specific counts for each genomic window or region.

Usage

```
strandedCounts(bam.files, param=readParam(), regions=NULL, ...)
```

Arguments

`bam.files`	a character vector containing paths to sorted and indexed BAM files
`param`	a `readParam` object containing read extraction parameters, or a list of such objects (one for each BAM file), where the `forward` slot must be set to `NULL`
`regions`	a `GRanges` object specifying the regions over which reads are to be counted
`...`	other arguments to be passed to `windowCounts` or `regionCounts`

Details

Some applications require strand-specific counts for each genomic region. This function calls `windowCounts` after setting `param$forward` to `TRUE` and `FALSE`. Any existing value of `param$forward` is ignored. If `regions` is specified, `regionCounts` is used instead of `windowCounts`.

The function then concatenates the two `SummarizedExperiment` objects (one from each strand). The total numbers of reads are added together to form the new `totals`. However, the total numbers of reads for each strand are also stored for future reference. Count loading parameters are also stored in the `exptData`.

Each row in the concatenated object corresponds to a stranded genomic region, where the strand of the region indicates the strand of the reads that were counted in that row. Note that there may not be two rows for each genomic region. This is because any empty rows, or those with counts below `filter`, will be removed within each call to `windowCounts`.

Value

A `SummarizedExperiment` object containing strand-specific counts for genomic regions.

Warnings

Users should be aware that many of the downstream range-processing functions are not strand-aware by default, e.g., `mergeWindows`. Any strandedness of the ranges will be ignored in these functions. If strand-specific processing is desired, users must manually set `ignore.strand=FALSE`.

The input `param$forward` should be set to `NULL`, as a safety measure. This is because the returned object is a composite of two separate calls to the relevant counting function. If the same `param` object is supplied to other functions, an error will be thrown if `param$forward` is `NULL`. This serves to remind users that such functions should instead be called twice, i.e., separately for each strand after setting `param$forward` to `TRUE` or `FALSE`.

Author(s)

Aaron Lun

See Also

`windowCounts`, `regionCounts`

Examples

```
bamFiles <- system.file("exdata", c("rep1.bam", "rep2.bam"), package="csaw")
xparam <- readParam(forward=NULL)
out <- strandedCounts(bamFiles, filter=1, param=xparam)
out
strandedCounts(bamFiles, filter=1, width=100, param=xparam)
strandedCounts(bamFiles, filter=1, param=reform(xparam, minq=20))

incoming <- GRanges(c('chrA', 'chrA', 'chrB', 'chrC'),
    IRanges(c(1, 500, 100, 1000), c(200, 1000, 700, 1500)))
strandedCounts(bamFiles, regions=incoming, param=xparam)
strandedCounts(bamFiles, regions=incoming, param=reform(xparam, dedup=TRUE))

# Throws an error, as the same reads are not involved.
try(windowCounts(bamFiles, filter=1, width=100, param=xparam))

# Running mergeWindows with strand-specificity.
mergeWindows(rowRanges(out), tol=100)
mergeWindows(rowRanges(out), tol=100, ignore.strand=FALSE)

rwsms <- rowSums(assay(out))
summary(findMaxima(rowRanges(out), range=100, metric=rwsms))
summary(findMaxima(rowRanges(out), range=100, metric=rwsms, ignore.strand=FALSE))
```

`upweightSummit` *Upweight summits*

Description

Upweight the highest-abudance window(s) in a cluster.

Usage

```
upweightSummit(ids, summits)
```

Arguments

`ids`	an integer vector of cluster IDs
`summits`	a logical vector indicating whether each window is a summit, or an integer vector containing the indices of summit windows

Details

This function computes weights for each window in a cluster, where the highest-abundance windows are upweighted. These weights are intended for use in `combineTests`, such that the summits of a cluster have a greater influence on the combined p-value. This is more graduated than simply using the summits alone, as potential DB between summits can still be detected. Summits can be obtained through `findMaxima` or by running `getBestTest` with `by.pval=FALSE`.

The exact value of the weight is arbitrary. Greater weight represents a stronger belief that DB occurs at the most abundant window. Here, the weighting scheme is designed such that the maximum Simes correction is not more than twice that without weighting. It will also be no more than twice that from applying Simes' method on the summits alone. This (restrained) conservativeness is an acceptable cost for considering DB events elsewhere in the cluster, while still focusing on the most abundant site.

Value

A numeric vector of weights, where the highest-abundance window in each cluster is assigned a greater weight.

Author(s)

Aaron Lun

References

Benjamini Y and Hochberg Y (1997). Multiple hypotheses testing with weights. *Scand. J. Stat.* 24, 407-418.

See Also

`combineTests`, `findMaxima`, `getBestTest`

Examples

```
nwin <- 20
set.seed(20)
ids <- sample(5, nwin, replace=TRUE)
summits <- sample(5, nwin, replace=TRUE)==1L
weights <- upweightSummit(ids, summits)

# Checking that the summit is upweighted in each cluster.
split(data.frame(summits, weights), ids)
```

`windowCounts` *Count reads overlapping each window*

Description

Count the number of extended reads overlapping a sliding window at spaced positions across the genome.

Usage

```
windowCounts(bam.files, spacing=50, width=spacing, ext=100, shift=0,
filter=10, bin=FALSE, param=readParam())
```

Arguments

`bam.files`	a character vector containing paths to sorted and indexed BAM files
`spacing`	an integer scalar specifying the distance between consecutive windows
`width`	an integer scalar specifying the width of the window
`ext`	an integer scalar or vector, containing the average length(s) of the sequenced fragments in each library
`shift`	an integer scalar specifying how much the start of each window should be shifted to the left
`filter`	an integer scalar for the minimum count sum across libraries for each window
`bin`	an integer scalar indicating whether binning should be performed
`param`	a `readParam` object containing read extraction parameters, or a list of such objects (one for each BAM file)

Value

A `SummarizedExperiment` object is returned containing one integer matrix. Each entry of the matrix contains the count for each library (column) at each window (row). The coordinates of each window are stored as the `rowRanges`. The total number of reads in each library are stored as `totals` in the `colData`, along with the read extension length and `param` used in each library. Other window counting parameters (e.g., `spacing`, `width`) are stored in the `exptData`.

Defining the sliding windows

A window is defined as a genomic interval of size equal to `width`. The value of `width` can be interpreted as the width of the contact area between the DNA and protein. In practical terms, it determines the spatial resolution of the analysis. Larger windows count reads over a larger region which results in larger counts. This results in greater detection power at the cost of resolution.

By default, the first window on a chromosome starts at base position 1. This can be shifted to the left by specifying an appropriate value for `shift`. New windows are found by sliding the current window to the right by the specified `spacing`. Increasing `spacing` will reduce the frequency at which counts are extracted from the genome. This results in some loss of resolution but it may be necessary when machine memory is limited.

If `bin` is set, settings are internally adjusted so that all reads are counted into non-overlapping adjacent bins of size `width`. Specifically, `spacing` is set to `width` and `filter` is capped at a maximum value of 1 (empty bins can be retained with `filter=0`). Only the 5' end of each read or the midpoint of each fragment (for paired-end data) is used in counting.

Read extraction and counting

Read extraction from the BAM files is governed by the `param` argument. Specifying a single `readParam` object will use the same extraction parameters for all files. Different parameters can also be used for each file by specifying a list of `readParam` objects. However, users should take care with library-specific parameters, lest spurious differences be introduced between libraries.

Fragments are inferred from reads by directional extension (single-end; see below) or by identifying proper pairs (paired-end; see `readParam` for more details). The number of fragments overlapping the window for each library is then counted for each window position. Windows will be removed if the count sum across all libraries is below `filter`. This reduces the memory footprint of the output by not returning empty or near-empty windows, which are usually uninteresting anyway.

The strandedness of the output `rowRanges` is set based on the strand(s) from which the reads are extracted and counted. This is determined by the value of the `forward` slot in the input `param` object.

Elaborating on directional extension

For single-end reads, directional extension is performed whereby each read is extended from its 3' end to the average fragment length, i.e., `ext`. This obtains a rough estimate of the interval of the fragment from which the read was derived. It is particularly useful for TF data, where extension specifically increases the coverage of peaks that exhibit strand bimodality. Substantially different fragment lengths between libraries can be accommodated by supplying a vector to `ext`, where each entry represents the extension length for the corresponding library. If any value of `ext` is set to `NA`, the read length is used as the fragment length in that library.

However, different fragment lengths will result in different peak widths between libraries. This may result in the detection of irrelevant differences corresponding to these differences in widths. To avoid this, fragment lengths in all libraries can be scaled to `final.ext`. For a bimodal peak, scaling effectively aligns the subpeaks on a given strand across all libraries to a common location. This removes the most obvious differences in widths.

The value of `final.ext` is sourced from the attributes of `ext` (see `makeExtVector` for more details). If this attribute is not present or is `NA`, no rescaling is performed.

Comments on ext for paired-end data

Directional extension is not performed for paired-end data, so the values in `ext` are not used directly. Hwoever, rescaling can still be performed to standardize fragment lengths across libraries, by resizing each fragment from its midpoint. This still uses `final.ext` in the attributes of the `ext` parameter.

On a similar note, some downstream functions will use the extension length in the output `colData` as the average fragment length. Thus, to maintain compatibility, users are recommended to set `ext` to a vector holding the median fragment length in each library. These values will not be used in `windowCounts`, but instead, in functions like `getWidths`.

Author(s)

Aaron Lun

See Also

`correlateReads`, `readParam`, `makeExtVector`

Examples

```
# A low filter is only used here as the examples have very few reads.
bamFiles <- system.file("exdata", c("rep1.bam", "rep2.bam"), package="csaw")
windowCounts(bamFiles, filter=1)
windowCounts(bamFiles, width=100, filter=1)
windowCounts(bamFiles, ext=c(50, 100), spacing=100, filter=1)
windowCounts(bamFiles, ext=makeExtVector(c(50, 100), 80), width=100)

# Loading PE data.
bamFile <- system.file("exdata", "pet.bam", package="csaw")
windowCounts(bamFile, param=readParam(pe="both"), filter=1)
windowCounts(bamFile, param=readParam(pe="first"), filter=1)
windowCounts(bamFile, param=readParam(max.frag=100, pe="both"), filter=1)
windowCounts(bamFile, param=readParam(max.frag=100, pe="both", restrict="chrA"), filter=1)

# Running rescues of PE data (use max.frag=-1 to coerce failure).
windowCounts(bamFile, param=readParam(max.frag=50, pe="both", rescue.ext=200), filter=1)
```

---

wwhm *Window width at half maximum*

Description

Get the width of the window from the half-maximum of the coverage profile.

Usage

```
wwhm(profile, regions, ext=100, param=readParam(), proportion=0.5, rlen=NULL)
```

Arguments

`profile`	a numeric vector containing a coverage profile, as produced by `profileSites`
`regions`	the GRanges object with which the profile was constructed
`ext`	an integer scalar specifying the average fragment length for single-end data
`param`	a readParam object containing read extraction parameters, or a list of such objects

`proportion`	a numeric scalar specifying the proportion of the maximum coverage at which to compute the window width
`rlen`	a numeric scalar or vector containing read lengths, if any `ext=NA`, i.e., fragments are unextended reads

Details

This function computes the ideal window size, based on the width of the peak in the coverage profile at the specified `proportion` of the maximum. Obviously, the values of `regions`, `ext` and `param` should be the same as those used in `profileSites`. The `regions` should contain windows of a constant size.

Some subtleties are involved in obtaining the window width. First, twice the average fragment length must be subtracted from the peak width, as the profile is constructed from (inferred) fragments. The size of the viewpoints in `regions` must also be subtracted, to account for the inflated peak width when spatial resolution is lost.

Value

An integer scalar is returned, specifying the ideal window width.

Author(s)

Aaron Lun

See Also

`profileSites`, `getWidths`

Examples

```
x <- dnorm(-200:200/100) # Mocking up a profile.
windows <- GRanges("chrA", IRanges(1, 50)) # Making up some windows.

wwhm(x, windows)
wwhm(x, windows, ext=50)
wwhm(x, windows, proportion=0.2)

# Need to set 'rlen' if ext=NA.
wwhm(x, windows, ext=NA, rlen=10)
```

Index

∗Topic **annotation**
- detailRanges, 9

∗Topic **clustering**
- consolidateSizes, 4
- mergeWindows, 23

∗Topic **counting**
- dumpPE, 11
- makeExtVector, 22
- Parameter list methods, 29
- readParam, 32
- regionCounts, 35
- strandedCounts, 39
- windowCounts, 42

∗Topic **diagnostics**
- correlateReads, 6
- getPESizes, 19
- profileSites, 30
- wwhm, 44

∗Topic **documentation**
- csawUsersGuide, 8

∗Topic **filtering**
- filterWindows, 14
- findMaxima, 16
- getWidths, 21
- scaledAverage, 36

∗Topic **normalization**
- normalizeCounts, 25
- SEmethods, 38

∗Topic **testing**
- combineTests, 2
- getBestTest, 18
- overlapStats, 27
- upweightSummit, 40

∗Topic **visualization**
- extractReads, 13

[.extVector *(makeExtVector)*, 22
$,readParam-method *(readParam)*, 32

asDGEList *(SEmethods)*, 38
asDGEList,SummarizedExperiment-method *(SEmethods)*, 38
aveLogCPM, *14*, *16*, *17*, *36*, *37*
aveLogCPM.DGEList, *37*

calcNormFactors, *25*, *26*
ccf, *7*
checkList *(Parameter list methods)*, 29
combineOverlaps *(overlapStats)*, 27
combineTests, 2, *4*, *5*, *18*, *19*, *24*, *27*, *28*, *41*
consolidateSizes, 4
correlateReads, 6, *7*, *34*, *44*
countOverlaps, *35*
csaw *(csawUsersGuide)*, 8
csawUsersGuide, 8

detailRanges, 9
DGEList, *38*
dumpPE, 11

extractReads, 13

filterWindows, 14
findMaxima, 16, *28*, *31*, *41*
findOverlaps, *4*, *5*, *27*

getBestOverlaps *(overlapStats)*, 27
getBestTest, 18, *27*, *28*, *41*
getPESizes, 19, *21*, *33*, *34*
getWidths, *15*, *16*, 21, *36*, *37*, *44*, *45*

Hits, *5*

loessFit, *25*, *26*

makeExtVector, 22, *35*, *43*, *44*
mergeWindows, *3–5*, *18*, *19*, 23, *39*

normalize *(SEmethods)*, 38
normalize,SummarizedExperiment-method *(SEmethods)*, 38

normalizeCounts, *25*, 25, *38*
normalizeCyclicLoess, *25*, *26*

overlapStats, 27

Parameter list methods, 29
profileSites, 30, *44*, *45*

readParam, *7*, *12*, *13*, *20*, 32, *35*, *43*, *44*
readParam-class (readParam), 32
reform, *13*, *29*
reform (readParam), 32
reform,readParam-method (readParam), 32
reformList (Parameter list methods), 29
regionCounts, *21–23*, *29*, *31*, *34*, 35, *39*, *40*

scaledAverage, *15*, *16*, 36
SEmethods, 38
show,readParam-method (readParam), 32
strandedCounts, *33*, 39
summitOverlaps (overlapStats), 27
Sweave, *8*
system, *9*

upweightSummit, *27*, *28*, 40

windowCounts, *4*, *5*, *12*, *13*, *15–17*, *21–24*, *29*, *31*, *34*, *35*, *38–40*, 42
wwhm, *31*, 44