Package 'csaw'

April 11, 2018

Version 1.12.0

Date 2017-10-13

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 (>= 3.4), GenomicRanges, SummarizedExperiment (>= 1.2.0), BiocParallel

Imports Rsamtools, edgeR, limma, GenomicFeatures, AnnotationDbi, methods, S4Vectors, IRanges, GenomeInfoDb, BiocGenerics, Rhtslib, stats, Rcpp

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

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

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

LinkingTo Rhtslib, zlibbioc, Rcpp

SystemRequirements C++11

R topics documented:

---

checkBimodality   *Check bimodality of regions*

Description

Compute the maximum bimodality score across all base pairs in each region.

Usage

```
checkBimodality(bam.files, regions, width=100, param=readParam(),
    prior.count=2, invert=FALSE)
```

Arguments

bam.files	a character vector containing paths to sorted and indexed BAM files
regions	a GRanges object specifying the regions over which bimodality is to be calculated
width	an integer scalar or list indicating the span with which to compute bimodality
param	a readParam object containing read extraction parameters
prior.count	a numeric scalar specifying the prior count to compute bimodality scores
invert	a logical scalar indicating whether bimodality score should be inverted

Details

Consider a base position x. This function counts the number of forward- and reverse-strand reads within the interval [x-width+1, x]. It then calculates the forward:reverse ratio after adding prior.count to both counts. This is repeated for the interval [x, x+width-1], and the reverse:forward ratio is then computed. The smaller of these two ratios is used as the bimodality score.

Sites with high bimodality scores will be enriched for forward- and reverse-strand enrichment on the left and right of the site, respectively. Given a genomic region, this function will treat each base position as a site. The largest bimodality score across all positions will be reported for each region. The idea is to assist with the identification of transcription factor binding sites, which exhibit strong strand bimodality. The function will be less useful for broad targets like histone marks.

If multiple `bam.files` are specified, they are effectively pooled so that counting uses all reads in all files. A separate value of `width` can be specified for each library, to account for differences in fragmentation – see the `ext` argument for `windowCounts` for more details. In practice, this is usually unnecessary. Setting `width` to the average fragment length yields satisfactory results in most cases.

If `invert` is set, the bimodality score will be flipped around, i.e., it will be maximized when reverse-strand coverage dominates on the left, and forward-strand coverage dominates on the right. This is designed for use in CAGE analyses where this inverted bimodality is symptomatic of enhancer RNAs.

Value

A numeric vector containing the maximum bimodality score across all bases in each region.

Author(s)

Aaron Lun

Examples

```
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(100, 580, 500, 1500)))

checkBimodality(bamFiles, incoming)
checkBimodality(bamFiles, incoming, width=200)
checkBimodality(bamFiles, incoming, param=readParam(minq=20, dedup=TRUE))
checkBimodality(bamFiles, incoming, prior.count=5)

# Works on PE data; scores are computed from paired reads.
bamFile <- system.file("exdata", "pet.bam", package="csaw")
checkBimodality(bamFile, incoming[1:3], param=readParam(pe="both"))
checkBimodality(bamFile, incoming[1:3], param=readParam(pe="both", max.frag=100))
```

---

clusterFDR *Compute the cluster-level FDR*

Description

Compute the FDR across clusters based on the test-level FDR threshold

Usage

```
clusterFDR(ids, threshold, weight=NULL)
controlClusterFDR(target, adjp, FUN, ..., weight=NULL, grid.param=NULL)
```

Arguments

`ids`	an integer vector of cluster IDs for each significant test below `threshold`
`threshold`	a numeric scalar, specifying the FDR threshold used to define the significant tests
`target`	a numeric scalar specifying the desired cluster-level FDR threshold
`adjp`	a numeric vector of window-level adjusted p-values
`FUN`	a clustering function that takes a logical vector indicating which windows are significant, and returns an integer vector of cluster IDs (see below)
`...`	additional arguments to be passed to `FUN`
`weight`	a numeric vector of frequency weights, for internal use
`grid.param`	a named list of grid search parameters, see Details

Value

For `clusterFDR`, a numeric scalar is returned as the cluster-level FDR.

For `controlClusterFDR`, a list is returned containing two numeric scalars – `threshold`, the window-level FDR threshold to control the cluster-level FDR near `target`; and `FDR`, the estimate of the cluster-level FDR corresponding to `threshold`.

Definition of the cluster-level FDR

The `clusterFDR` function computes an informal estimate of the cluster-level FDR, where each cluster is formed by aggregating only significant tests. In the context of ChIP-seq, each significant test refers to a DB window that is detected at a FDR below `threshold`. The idea is to obtain an error rate while reporting the precise coordinates of a DB subinterval in a complex region.

The cluster-level FDR is defined as the proportion of reported clusters that have no true positives. Simply using `threshold` is not appropriate, as the cluster- and window-level FDRs are not equivalent. This function also differs from the standard pipeline that is based on `combineTests`. Specifically, region definition in `combineTests` must be independent of DB so that precise coordinates of the DB subinterval cannot be reported. This is overcome here, by clustering directly on DB windows and applying post-hoc control of the cluster-level FDR.

Note that the calculation of the cluster-level FDR here is not statistically rigorous. In particular, the observed number of false positive tests is estimated based on `threshold` and the total number of significant tests. This is not guaranteed to be an upper bound, especially with few or correlated tests. Thus, users should use the standard `combineTests`-based pipeline wherever possible. Clustering on significant windows should only be performed where the precise coordinates of the DB subinterval are important for interpretation.

Searching for the best threshold

`controlClusterFDR` will identify the window-level FDR threshold required to control the cluster-level FDR at `target`. The former is not a simple function of the latter (neither continuous nor guaranteed to be monotonic), so a grid search is used. Clusters of significant windows are identified at each window-level threshold, and the corresponding cluster-level FDR is computed with `clusterFDR`.

At each iteration, the grid point with the closest cluster-level FDR to `target` is chosen and the grid is recentered around that point. The size of the grid is also scaled down to provide greater resolution. Users can tune the settings of the grid search by specifying elements in `grid.param` as:

`length`: an integer scalar specifying the length of the grid, defaults to 21

range: a numeric scalar indicating the range of the grid in logit units, defaults to 20

iter: an integer scalar indicating the number of iterations of the grid search

scale: a numeric scalar specifying how the range of the grid should be downscaled at each iteration

The FUN argument should be a function that accepts a logical vector specifying significance, and returns an integer vector of cluster IDs. If, for example, it accepts an input vector ix, then the output should contain cluster IDs corresponding to the entries of which(ix). This is because cluster IDs are only defined for significant tests, given that only those tests are used for clustering.

An additional requirement is that the returned window-level FDR threshold should be less than target. Thus, each window should be significantly DB on its own merits before it is placed into a cluster. This protects against scenarios where very large thresholds yield low cluster-level FDRs, due to the formation of a few large clusters.

Note about weights

In both functions, the weight argument is assumed to contain frequency weights of significant tests/windows. For example, a weight of 2 for a test would be equivalent to repeating that test (i.e., repeating the same window so it shows up twice in your analysis). These weights should be the same as those used during weighted FDR control to compute adjusted p-values. In general, you should not set this argument unless you know what you're doing.

Author(s)

Aaron Lun

See Also

mergeWindows, combineTests, clusterWindows

Examples

```
# Setting up the windows and p-values.
set.seed(100)
windows <- GRanges("chrA", IRanges(1:1000, 1:1000))
test.p <- runif(1000)
test.p[c(1:10, 100:110, 220:240)] <- 0 # 3 significant subintervals.

# Defining significant windows.
threshold <- 0.05
is.sig <- p.adjust(test.p, method="BH") <= threshold

# Assuming that we only cluster significant windows.
merged <- mergeWindows(windows[is.sig], tol=0)
clusterFDR(merged$id, threshold)

# Setting up another example with more subintervals.
test.p <- runif(1000)
test.p[rep(1:2, 50) + rep(0:49, each=2) * 20] <- 0
adj.p <- p.adjust(test.p, method="BH")
clusterFUN <- function(x) { mergeWindows(windows[x], tol=0)$id }
controlClusterFDR(0.05, adj.p, clusterFUN)
```

clusterWindows	*Cluster DB windows into clusters*

Description

Clusters significant windows into clusters while controlling the cluster-level FDR.

Usage

```
clusterWindows(regions, tab, target, pval.col=NULL,
    fc.col=NA, tol, ..., weight=NULL, grid.param=NULL)
```

Arguments

regions	a GRanges or RangedSummarizedExperiment object containing window coordinates
tab	a dataframe of results with a PValue field for each window
target	a numeric scalar indicating the desired cluster-level FDR
pval.col	a string or integer scalar specifying the column of tab with the p-values
fc.col	a string or integer scalar specifying the column of tab with the log-fold changes
tol, ...	arguments to be passed to mergeWindows
weight, grid.param	
	arguments to be passed to controlClusterFDR

Details

Windows are identified as DB based on the adjusted p-values in tab. Only these DB windows are then used directly for clustering via mergeWindows. This identifies DB regions consisting solely of DB windows. If tol is not specified, it is set to 100 bp by default and a warning is raised. If fc.col is used to specify the column of log-fold changes, clusters are formed according to the sign of the log-fold change in mergeWindows.

DB-based clustering is obviously not blind to the DB status, so standard methods for FDR control are not valid. Instead, post-hoc control of the cluster-level FDR is applied by using controlClusterFDR. This aims to control the cluster-level FDR at target (which is set to 0.05 if not specified). The aim is to provide some interpretable results when DB-blind clustering is not appropriate, e.g., for diffuse marks involving long stretches of the genome. Reporting each marked stretch in its entirety would be cumbersome, so this method allows the DB subintervals to be identified directly.

Value

A named list similar to that reported by mergeWindows with an ID vector in id and region coordinates of each cluster in region. Non-significant windows are marked with NA values in ids. An additional element FDR is also included, representing the estimate of the cluster-level FDR for the returned regions.

Author(s)

Aaron Lun

See Also

[mergeWindows](#), [controlClusterFDR](#)

Examples

```
set.seed(10)
x <- round(runif(100, 100, 1000))
gr <- GRanges("chrA", IRanges(x, x+5))
tab <- data.frame(PValue=rbeta(length(x), 1, 50), logFC=rnorm(length(x)))

clusterWindows(gr, tab, target=0.05, tol=10)
clusterWindows(gr, tab, target=0.05, tol=10, fc.col="logFC")
```

---

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 or factor containing the cluster ID for each test
tab	a dataframe of results with `PValue` 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 or string specifying the column of `tab` containing the p-values
fc.col	an integer or character vector specifying the columns of `tab` containing the log-fold changes

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. The p-value column is expected to be named as PValue. 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.

A simple clustering approach for windows is provided in [mergeWindows](). However, anything can be used so long as it is independent of the p-values and does not compromise type I error control, e.g., promoters, gene bodies, independently called peaks. Any tests with NA values for ids will be ignored.

Value

A dataframe with one row per cluster and various fields:

- An integer field nWindows, specifying the total number of windows in each cluster.
- 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.
- A numeric field containing the combined p-value. If pval.col=NULL, this column is named PValue, otherwise its name is set to colnames(tab[,pval.col]).
- A numeric field FDR, containing the q-value corresponding to the combined p-value.
- A character field direction (if fc.col is of length 1), specifying the dominant direction of change for windows in each cluster.

Each row is named according to the ID of the corresponding cluster.

Determining the direction of DB

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-negligble 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.

When only one log-fold change column is specified, combineTests will determine which DB direction contributes to the combined p-value. This is done by considering whether the combined p-value would increase if all tests in one direction were assigned p-values of unity. If there is an increase, then tests changing in that direction must contribute to the calculations in Simes' method. In this manner, clusters are labelled based on whether they are driven by tests with positive ("up") or negative log-fold changes ("down") or both ("mixed").

The label for each cluster is stored as the direction field in the returned data frame. However, keep in mind that the label only describes the direction of change among the most significant tests in the cluster. Clusters with complex DB may still be labelled as changing in only one direction, if the tests changing in one direction have much lower p-values than the tests changing in the other direction (even if both sets of p-values are significant).

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)
```

---

consolidateClusters   *Consolidate DB clusters*

Description

Consolidate DB results from multiple analyses with cluster-level FDR control.

Usage

```
consolidateClusters(data.list, result.list, equiweight=TRUE, ...)
```

Arguments

data.list	a list of RangedSummarizedExperiment and/or GRanges objects
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 analysis should be enforced
...	arguments to be passed to [clusterWindows](#)

Details

This function consolidates DB results from multiple analyses, typically involving different window sizes. The aim is to provide comprehensive detection of DB at a range of spatial resolutions. Significant windows from each analysis are identified and used for clustering with clusterWindows. This represents the post-hoc counterpart to consolidateSizes.

Some effort is required to equalize the contribution of the results from each analysis. This is done by setting equiweight=TRUE, where the weight of each window is inversely proportional to the number of windows from that analysis. These weights are used as frequency weights for window-level FDR control (to identify DB windows prior to clustering). Otherwise, the final results would be dominated by large number of small windows.

Users can cluster by the sign of log-fold changes, to obtain clusters of DB windows of the same sign. However, note that nested windows with opposite signs may give unintuitive results - see mergeWindows for details.

Value

A named list is returned containing:

id	a list of integer vectors indicating the cluster ID for each window in data.list
region	a GRanges object containing the coordinates for each cluster
FDR	a numeric scalar containing the cluster-level FDR estimate

Author(s)

Aaron Lun

See Also

clusterWindows, consolidateSizes

Examples

```
# Making up some GRanges.
low <- GRanges("chrA", IRanges(runif(100, 1, 1000), width=5))
med <- GRanges("chrA", IRanges(runif(40, 1, 1000), width=10))
high <- GRanges("chrA", IRanges(runif(10, 1, 1000), width=20))

# Making up some DB results.
dbl <- data.frame(logFC=rnorm(length(low)), PValue=rbeta(length(low), 1, 20))
dbm <- data.frame(logFC=rnorm(length(med)), PValue=rbeta(length(med), 1, 20))
dbh <- data.frame(logFC=rnorm(length(high)), PValue=rbeta(length(high), 1, 20))
result.list <- list(dbl, dbm, dbh)

# Consolidating.
cons <- consolidateClusters(list(low, med, high), result.list, tol=20)
cons$region
cons$id
cons$FDR

# Without weights.
cons <- consolidateClusters(list(low, med, high), result.list, tol=20, equiweight=FALSE)
cons$FDR

# Using the signs.
```

```
cons <- consolidateClusters(list(low, med, high), result.list, tol=20, fc.col="logFC")
```

---

consolidateSizes *Consolidate window sizes*

Description

Consolidate DB results from multiple window sizes.

Usage

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

Arguments

data.list	a list of RangedSummarizedExperiment and/or GRanges objects
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 named list of parameters to pass to mergeWindows
combine.args	a named list of parameters to pass to combineTests
region	a GRanges object specifying regions of interest for overlapping with windows
overlap.args	a named 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. RangedSummarizedExperiment 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. If merge.args$tol is not specified, it is automatically set to 100 bp and a warning is raised.

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 named 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

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 `max.dist`. 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.

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, such that the fragment length peak will no longer be easily visible.

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.

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`.

Paired-end data is also supported, whereby correlations are computed using only those reads in proper pairs. This may be less meaningful as the presence of proper pairs will inevitably result in a strong peak at the fragment length. Instead, IP efficiency can be diagnosed by treating paired-end data as single-end, e.g., with `pe="first"` in `readParam`.

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](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")

# Also works on paired-end data.
bamFile <- system.file("exdata", "pet.bam", package="csaw")
x <- correlateReads(bamFile, param=readParam(pe="both"))
head(x)
```

---

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",
    ignore.strand=TRUE)
```

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

ignore.strand   a logical scalar indicating whether strandedness in incoming should be ignored

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 neighboring genes within a specified range are reported, rather than just the closest gene to the region. If a region in `incoming` is stranded and `ignore.strand=FALSE`, annotated features will only be reported if they lie on the same strand as that region.

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.

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.

Explanation of fields

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.

Exons are numbered in order of increasing start or end position for genes on the forward or reverse strands, respectively. Exon ranges in EXON are reported from as a comma-separated list where stretches of consecutive exons are summarized into a range. Promoters are defined around any annotated TSS in `txdb`, and are marked as exon 0. Genes can have multiple TSS, but an overlap to multiple promoters will only be reported once. If the region overlaps an intron, it is labelled with I in EXON. Intronic overlaps are not 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 cases where the distance is reported to the first exon, it can be used to refine the choice of `promoter`.

Other options

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.

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.

Author(s)

Aaron Lun

Examples

```
library(org.Mm.eg.db)
library(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)
```

---

empiricalFDR *Control the empirical FDR*

Description

Control the empirical FDR across clusters for comparisons to negative controls, based on tests that are significant in the wrong direction.

Usage

```
empiricalFDR(ids, tab, weight=NULL, pval.col=NULL, fc.col=NULL, neg.down=TRUE)
```

Arguments

`ids`	an integer vector containing the cluster ID for each test
`tab`	a dataframe of results with `PValue` 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 or string specifying the column of `tab` containing the p-values
`fc.col`	an integer scalar or string specifying the column of `tab` containing the log-fold changes
`neg.down`	a logical scalar indicating if negative log-fold changes correspond to the "wrong" direction

Details

Some experiments involve comparisons to negative controls where there should be no signal/binding. In such case, genuine differences should only occur in one direction, i.e., up in the non-control samples. Thus, the number of significant tests that change in the wrong direction can be used as an estimate of the number of false positives.

This function converts two-sided p-values in `tab[,pval.col]` to one-sided counterparts in the wrong direction. It combines the one-sided p-values for each cluster using [combineTests](). The number of significant clusters at some p-value threshold represents the estimated number of false positive clusters.

The same approach is applied for one-sided p-values in the right direction, where the number of detected clusters at the threshold represents the total number of discoveries. Dividing the number of false positives by the nmber of discoveries yields the empirical FDR at each p-value threshold. Monotonicity is enforced (i.e., the empirical FDR only decreases with decreasing p-value) as is the fact that the empirical FDR must be below unity.

The p-values specified in `pval.col` are assumed to be originally computed from some two-sided test, where the direction of change is independent of the magnitude of the p-value under the null hypothesis. This rules out p-values computed from, e.g., ANODEV where multiple contrasts are tested at once.

Control of the empirical FDR is best used for very noisy data sets where the BH method is not adequate. While the BH method protects against statistical false positives, the empirical FDR also protects against experimental false positives, e.g., due to non-specific binding. The downside is that the empirical FDR calculation relies on the availability of a good estimate of the number of false positives. The BH method in [combineTests]() is more statistically rigorous and should be preferred for routine analyses.

Value

A data frame containing one row per cluster, with various fields:

- A numeric field containing the one-sided p-value for each cluster in the right direction. This field is named `PValue` if `pval.col=NULL`, otherwise its name is set to `colnames(tab[,pval.col]`.
- A numeric field `FDR`, containing the empirical FDR corresponding to the p-value threshold equal to the value in `PValue`.

All other fields are the same as those returned by [combineTests](). The exception is the `direction` field, which is not returned as it is not informative for one-sided tests.

Author(s)

Aaron Lun

References

Zhang Y, Liu T, Meyer CA et al. (2008). Model-based Analysis of ChIP-Seq (MACS). *Genome Biol.* 9, R137.

See Also

[combineTests](#)

Examples

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

---

extractReads	*Extract reads from a BAM file*

Description

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

Usage

```
extractReads(bam.file, region, ext=NA, param=readParam(), as.reads=FALSE)
```

Arguments

bam.file	a character string containing the path to a sorted and indexed BAM file
region	a GRanges object of length 1 describing the region of interest
ext	an integer scalar or list specifying the fragment length for directional read extension
param	a readParam object specifying how reads should be extracted
as.reads	a logical scalar indicating whether reads should be returned instead of fragments for paired-end data

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.

If pe!="both" in param, stranded intervals corresponding to the reads will be reported. If ext is not NA, directional read extension will also be performed – see [windowCounts](#) for more details. If pe="both", intervals are unstranded and correspond to fragments from proper pairs.

If as.reads=TRUE and pe="both", the reads in each proper pair are returned directly as a GRangesList of length 2. The two internal GRanges are of the same length and contain the forward and reverse reads for each proper pair in parallel. In other words, the nth elements of the first and second GRanges represent the nth proper pair.

Any strandedness of 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.

Value

If pe!="both" or as.reads=FALSE, a GRanges object is returned containing the read (for single-end data) or fragment intervals (for paired-end data). If pe="both" and as.reads=TRUE, a GRanges-List is returned containing the paired reads – see Details.

Author(s)

Aaron Lun

See Also

[readParam](#), [windowCounts](#)

Examples

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

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

# Extracting as reads.
extractReads(bamFile, GRanges("chrB", IRanges(100, 500)),
    param=readParam(pe="both"), as.reads=TRUE)

# Dealing with the extension length.
bamFile <- system.file("exdata", "rep1.bam", package="csaw")
my.reg <- GRanges("chrA", IRanges(10, 200))
extractReads(bamFile, my.reg)
extractReads(bamFile, my.reg, ext=100)
```

---

filterWindows   *Filtering methods for RangedSummarizedExperiment objects*

Description

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

Usage

```
filterWindows(data, background, type="global", assay.data=1,
    assay.back=1, prior.count=2, scale.info=NULL)

scaleControlFilter(data, background)
```

Arguments

data	A RangedSummarizedExperiment object containing window-level counts for `filterWindows`, and bin-level counts for `scaleControlFilter`.
background	A RangedSummarizedExperiment object. For `filterWindows`, this should contain counts for background regions when `type` is not `"proportion"`. For `scaleControlFilter`, this should contain bin-level counts for negative control samples.
type	a character string specifying the type of filtering to perform; can be any of `c("global", "local", "control", "proportion")`
assay.data	a string or integer scalar specifying the assay containing window/bin counts in `data`
assay.back	a string or integer scalar specifying the assay containing window/bin counts in `background`
prior.count	a numeric scalar, specifying the prior count to use in [aveLogCPM](#)
scale.info	A list containing the output of `scaleControlFilter`, i.e., a normalization factor and library sizes for ChIP and control samples.

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

For `filterWindows`, a named list is returned containing:

- abundances, a numeric vector containing the average abundance of each row in data.

- filter, a numeric vector containing the filter statistic for the given type for each row.

- back.abundances, a numeric vector containing the average abundance of each entry in background.
  Only reported for type!="proportion".

For scaleControlFilter, a named list is returned containing:

- scale, a numeric scalar containing the scaling factor for multiplying the control counts.

- data.totals, a numeric vector containing the library sizes for data.

- back.totals, anumeric vector containing the library sizes for 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.

Normalization for composition bias

When type=="control", ChIP samples will be compared to control samples to compute the filter statistic. Composition biases are likely to be present, where increased binding at some loci reduces coverage of other loci in the ChIP samples. This incorrectly results in smaller filter statistics for the latter loci, as the fold-change over the input is reduced. To correct for this, a normalization factor between ChIP and control samples can be computed with scaleControlFilter.

Users should supply two RangedSummarizedExperiment objects, each containing the counts for large (~10 kbp) bins in the ChIP and control samples. The difference in the average abundance between the two objects is computed for each bin. The median of the differences across all bins is used as a normalization factor to correct the filter statistics for each window. The idea is that most bins represent background regions, such that a systematic difference in abundance between ChIP and control should represent the composition bias.

scaleControlFilter will also store the library sizes for each object in its output. This is used to check that the data and background supplied to filterWindows have the same library sizes. Otherwise, the normalization factor computed with bin-level counts cannot be correctly applied to the window-level counts.

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, with binning for normalization (pretend rep2.bam is a control).
binned <- windowCounts(bamFiles, width=10000, bin=TRUE)
chip.bin <- binned[,1]
con.bin <- binned[,2]
scinfo <- scaleControlFilter(chip.bin, con.bin)

chip.data <- data[,1]
con.data <- data[,2]
stats <- filterWindows(chip.data, con.data, type="control",
    prior.count=5, scale.info=scinfo)

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 or factor 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 or string specifying the column of `tab` containing the p-values
`cpm.col`	an integer scalar or string specifying the column of `tab` containing the log-CPM values

Details

Clusters are identified as those tests with the same value of `ids` (any NA values are ignored). 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

Wasserman, L, and Roeder, K (2006). Weighted hypothesis testing. *arXiv preprint math/0604172*.

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 computes a number of diagnostics for paired-end data in a supplied BAM file. The aims is to provide an indication of the quality of library preparation and sequencing.

Firstly, a read is only considered to be 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, as will any unpaired reads in the BAM file. Secondary and supplementary alignments are ignored completely and do not contribute to the total - see readParam for details.

Of the mapped pairs, the valid (i.e., proper) read pairs are identified. This involves several criteria:

- Read pairs must be intrachromosomal. If the reads are on different chromosomes, the read pair will be recorded as being interchromosomal.
- The two reads in the pair must lie on opposite strands. Otherwise, the read pair will be considered as being improperly orientated.
- The 5' end of the forward read must not map to a higher genomic coordinate than the 5' end of the reverse read. Otherwise, the read pair will be considered as being improperly orientated.

Note that the 3' end of one read is allowed to overrun the 5' end of the other. This avoids being too stringent in the presence of sequencing errors, untrimmed adaptors, etc. at the start or ends of reads.

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. Note that any `max.frag` specification in `param` will be ignored; sizes for all valid pairs will be returned.

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 reads, the 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 RangedSummarizedExperiment 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 metadata(data)$final.ext, if it is not NA. Otherwise, it is taken from data$ext. If the fragment lengths are different between libraries, the average is used to compute the effective width of the window. For paired-end data, data$ext should be an average of the inferred fragment sizes, e.g., obtained with getPESizes.

If final.ext is NA and any of ext are NA, the function will extract the read lengths in data$rlen. This is because NA values of ext correspond to the use of unextended reads in windowCounts and regionCounts. The likely read lengths are automatically computed in each function but can also be set manually.

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)

# Average is used when multiple fragment lengths are present.
data <- windowCounts(bamFiles, ext=list(c(50, 100), NA), filter=1)
getWidths(data)

# Using the automatically computed 'rlen'.
data <- windowCounts(bamFiles, ext=NA, filter=1)
getWidths(data)
data$rlen <- 200 # Manually defining it, if required.
getWidths(data)

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

maximizeCcf   *Find the delay at the maximal CCF*

Description

Estimate the average fragment length by maximizing the cross-correlations.

Usage

```
maximizeCcf(profile, ignore=100)
```

Arguments

profile   a numeric vector containing a coverage profile, as produced by [correlateReads](#)
ignore   an integer scalar specifying the distances to ignore

Details

This function identifies the delay distance at which the cross-correlations are maximized. This distance can then be used as an estimate of the average fragment length, for use in directional extension during read counting.

In some datasets, identification of the maxima is confounded by a phantom peak at the read length. This can be overcome by ignoring the first ignore delay distances, such that the distance corresponding to the true peak is used.

Obviously, this only works in TF experiments with moderate to strong enrichment, where a strong peak in the CCF profile is present. The function may not perform sensibly in the presence of noisy profiles containing multiple local maxima.

Value

The average fragment length is returned as an integer scalar.

Author(s)

Aaron Lun

References

Landt SG, Marinov GK, Kundaje A, et al. (2012). ChIP-seq guidelines and practices of the EN-CODE and modENCODE consortia. *Genome Res.* 22, 1813-31.

See Also

[correlateReads](#)

Examples

```
x <- dnorm(-200:200/100) # Mocking up a profile.
maximizeCcf(x)

x2 <- x + dnorm(-50:250/10) # Adding a phantom peak
maximizeCcf(x2)
maximizeCcf(x2, ignore=0)
```

---

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 or RangedSummarizedExperiment object containing window coordinates
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 in regions 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 lead to a warning as splitting by sign becomes undefined. This is because any genomic region involving the parent window must contain the nested window, such that the cluster will always contain opposite log-fold changes. 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 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))
```

---

mixedClusters   *Tests for mixed DB clusters*

Description

Intersects two one-sided tests to determine if a cluster contains DB events in both directions.

Usage

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

Arguments

ids	an integer vector or factor containing the cluster ID for each test
tab	a dataframe of results with PValue 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 or string specifying the column of tab containing the p-values
fc.col	an integer scalar or string specifying the columns of tab containing the log-fold changes

Details

This function converts two-sided p-values to one-sided counterparts for each direction of log-fold change. For each direction, the corresponding one-sided p-values are combined to yield a combined p-value for each cluster. Each cluster is associated with two combined p-values (one in each direction), which are intersected using the Berger's intersection-union test (IUT). This tests against the null hypothesis that either direction is not significant. In short, a low p-value is only possible if there is DB in both directions. This formally identifies mixed clusters corresponding to complex DB events.

Value

A data frame containing one row per cluster, with various fields:

- A numeric field containing the IUT p-value for each cluster. This field is named PValue if pval.col=NULL, otherwise its name is set to colnames(tab[,pval.col].

- A numeric field FDR, containing the adjusted p-value for each cluster.

All other fields are the same as those returned by [combineTests](). The exception is the direction field, which is not returned as the test is done explicitly for mixed clusters.

Author(s)

Aaron Lun

References

Berger RL and Hsu JC (1996). Bioequivalence trials, intersection-union tests and equivalence confidence sets. *Statist. Sci.* 11, 283-319.

See Also

[combineTests]()

Examples

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

---

normOffsets   *Normalize counts between libraries*

Description

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

Usage

```
## S4 method for signature 'matrix'
normOffsets(object, lib.sizes=NULL, type=c("scaling", "loess"),
    weighted=FALSE, ...)

## S4 method for signature 'SummarizedExperiment'
normOffsets(object, assay=1, type="scaling", ..., se.out=NULL)

## S4 method for signature 'SummarizedExperiment'
normalize(object, ...)
```

Arguments

object	A matrix of numeric counts with one column per library and one row per genomic feature (e.g., window).
lib.sizes	A numeric vector specifying the total number of reads per library. This is extracted from object$totals for SummarizedExperiment inputs, or computed as colSums(object) for matrices.
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".
assay	An integer scalar or string specifying the assay values to use for normalization.
se.out	A logical scalar indicating whether or not a SummarizedExperiment object should be returned. Alternatively, a SummarizedExperiment object in which normalization factors are to be stored.

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 genomic window is used as an offset in a generalized linear model for that feature and 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 a SummarizedExperiment object is supplied, the values to be used for normalization will be extracted according to the specified assay field. Furthermore, se.out=TRUE means that the SummarizedExperiment method will return a modified version of object containing normalization information. Normalization factors are stored in the "norm.factors" field in the mcols, while the offset matrix is stored in the "offset" field in the assays. se.out=NULL will soon be deprecated and defaults to FALSE.

normalize,SummarizedExperiment-method has the same behaviour as normOffsets with se.out=TRUE. However, this will be deprecated to avoid clashes with other normalize methods for SummarizedExperiment objects.

Value

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

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

If `se.out=TRUE`, a SummarizedExperiment is returned that contains the computed normalization factors/offsets but is otherwise identical to `object`.

Additional details for SummarizedExperiment inputs

If `se.out` is a SummarizedExperiment object and `type="scaling"`, the function will calculate the normalization factors from `object` but return them in a modified version of `se.out`. This is useful when `se.out` contains counts for windows, but the normalization factors are computed using larger bins in `object`.

Note that the normalization factors can only be interpreted with respect to the library sizes used to calculate them. As such, the function will throw an error if the library sizes in `se.out$totals` are not identical to `object$totals`. Consistent library sizes can be achieved by using the same [readParam](#) object in [windowCounts](#) and related functions.

The use of a SummarizedExperiment object in `se.out` with `type="loess"` is not yet supported.

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)
normOffsets(counts)
normOffsets(counts, lib.sizes=rep(400, 4))

# Using loess-based normalization, instead.
offsets <- normOffsets(counts, type="loess")
head(offsets)
offsets <- normOffsets(counts, type="loess", span=0.4)
offsets <- normOffsets(counts, type="loess", iterations=1)

# Same for SummariedExperiment objects.
bamFiles <- system.file("exdata", c("rep1.bam", "rep2.bam"), package="csaw")
data <- windowCounts(bamFiles, width=100, filter=1)

normOffsets(data, se.out=FALSE)
```

```
normOffsets(data, se.out=TRUE)

another.data <- windowCounts(bamFiles, width=10)
normOffsets(data, se.out=another.data)

normOffsets(data, type="loess", se.out=TRUE)
head(normOffsets(data, type="loess", se.out=FALSE))
```

---

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, ...)

empiricalOverlaps(olap, tab, o.weight=NULL, i.weight=NULL, ...)

mixedOverlaps(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](#), [empiricalFDR](#), [mixedClusters](#) and [upweightSummit](#) for handling overlaps between windows and arbitrary pre-specified regions. They accept Hits objects produced by running [findOverlaps](#) between regions (as query) and windows (as subject). Each set of windows overlapping a region is defined as a cluster to compute 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 RangedSummarizedExperiment 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, getBestOverlaps, empiricalOverlaps and mixedOverlaps, 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](), [empiricalFDR](), [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)
empiricalOverlaps(olap, mock)

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

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

```
comb <- getBestOverlaps(olap, mock, i.weight=window.weights)
comb <- empiricalOverlaps(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)
```

profileSites	*Profile binding sites*

Description

Get the coverage profile around potential binding sites.

Usage

```
profileSites(bam.files, regions, param=readParam(), range=5000, ext=100,
    average=TRUE, normalize="none", strand=c("ignore", "use", "match"))
```

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

param   a readParam object containing read extraction parameters

range   an integer scalar specifying the range over which the profile will be collected

ext   an integer scalar or list specifying the average fragment length for single-end data

average   a logical scalar specifying whether the profiles should be averaged across regions

normalize   a string specifying how normalization of each region's profile should be performed prior to averaging

strand   a string indicating how stranded regions should be handled

Details

This function computes the average coverage profile around the specified regions. Specifically, the profile is constructed by counting the number of fragments overlapping each base in the interval flanking each entry of regions. The interval for each entry is centred at its start location (base zero) and spans the flanking range on either side.

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.

By default, an average of the coverage profiles across all regions is returned. Normalization of each region's profile is performed on by setting normalize to:

none: No normalization is performed, i.e., counts per base are directly averaged across all regions. Thus, the shape of the average profile is largely determined by high-abundance regions.

total: The profile for each region is divided by the sum of coverages across all bases in the interval. This effectively normalizes for the total number of reads in each region.

max: The profile for each region is divided by its maximum value. This ensures that the maximum height of each region is the same.

If average=FALSE, a separate profile will be returned for each region instead. This may be useful, e.g., for constructing heatmaps of enrichment across many regions.

The profile can be used to examine average coverage around known features of interest, like genes or transcription start sites. Its shape can guide the choice of the window size in windowCounts, or to determine if larger regions should be used in regionCounts. For the former, restricting the regions to locally maximal windows with findMaxima is recommended to capture the profile of binding events.

Value

If average=TRUE, a numeric vector of average coverages for each base position within range is returned, where the average is taken over all regions. The vector is named according to the relative position of each base to the start of the region. The interpretation of the coverages will depend on the value of normalize.

If average=FALSE, an integer matrix of coverage values is returned. Each row of the matrix corresponds to an entry in regions, while each column corresponds to a base position with range. Column names are set to the relative position of each base to the start of each region.

Comments on strand specificity

By default, the strandedness of the regions are ignored with strand="ignore". If strand="use", the behaviour of this function will differ between forward- and reverse-stranded entries in regions.

- Forward-stranded or unstranded regions are processed as previously described above. Base zero corresponds to the start of the region, negative distances correspond to the 5' flanking region, and positive distances correspond to the 3' flanking region.

- Reverse-stranded regions are flipped, i.e., base zero corresponds to the *end* of the region. Negative distances correspond to the 5' flanking region on the reverse strand, while positive distances correspond to the 3' flanking region on this strand.

This ensures that the center of the profile always corresponds to the 5' end of the region, with upstream regions on the left and downstream regions on the right. This may be useful for features where strandedness is important, e.g., TSS's.

By default, the strandedness of the region has no effect on the choice of reads that are used. If strand="match", the profile for reverse-strand regions is constructed from reverse-strand reads only. 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. Flipping of reverse-stranded profiles is also performed in this setting, as described for strand="use".

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)

# Running profileSites .
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)

# 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, strand="use")
points(as.integer(names(x2)), x2, col="red")
x3 <- profileSites(bamFile, regs, range=500, strand="match",
    param=readParam(forward=NULL))
points(as.integer(names(x3)), x3, col="blue")

# Returning separate profiles.
y <- profileSites(bamFile, rowRanges(data)[maxed], range=500, average=FALSE)
dim(y)
```

---

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

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

BPPARAM: a BiocParallelParam object specifying if and how parallelization is performed

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.

Note that secondary or supplementary alignments are ignored in all functions. The former usually refer to alternative mapping locations for the same read, while the latter refer to chimeric reads. Neither are of much use in a typical ChIP-seq analysis and will be discarded if they are present in the BAM file.

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 (on the same chromosome, obviously) 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.

The fragment size is defined as the length of the interval bounded by the 5' ends of two inward-facing reads. Those pairs with fragment sizes above max.frag are removed, as they are more likely to be the result of mapping errors than genuinely large fragment sizes. Users should run getPESizes to pick an appropriate value for their data sets, though thresholds of around 500-1000 bp are usually fine.

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. This can be done using standard tools like FixMateInformation from the Picard suite (http://broadinstitute.github.io/picard).

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 is 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.

Any soft clipping in alignments are ignored during extraction (this is also true for paired-end data). Soft clips are presumed to be sequencing artifacts, e.g., when the adaptor or barcodes are not properly removed from the read sequence. They are not relevant to computing genomic coverage. Thus, in this package, any references to the length or 5'/3' ends of the read will actually refer to that of the *alignment*. This is often more appropriate, e.g., the 5' end of the alignment represents the end of the fragment after clipping of the artifacts.

Parallelization options

Parallelization can be turned on in several functions by setting BPPARAM appropriately. This will usually extract reads from multiple files simultaneously to speed up processing. Users are referred to ?BiocParallelParam for more details on how to set BPPARAM. By default, a SerialParam object is used, i.e., no parallelization is performed. This is because it provides little benefit for small files or on systems with I/O bottlenecks.

Constructor

readParam(pe="none", max.frag=500, dedup=FALSE, minq=NA, forward=NA, restrict=NULL, discard=GRan
    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, BiocParallelParam

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, pe="both", max.frag=212.0)
```

---

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 list describing the average length of the sequenced fragment in each library, see ?windowCounts
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. In other words, both forward and reverse reads will be counted into each region, regardless of the strandedness of that region. This can be altered by setting the forward slot in the param object to only count reads from one strand or the other. The strandedness of the output rowRanges will depend on the strand(s) from which reads were counted.

See windowCounts for more details on read extension.

Value

A RangedSummarizedExperiment 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 length and extension length used in each library are stored in the colData. Other parameters (e.g., param) are stored in the metadata.

Author(s)

Aaron Lun

See Also

countOverlaps, windowCounts, readParam

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"))
```

---

scaledAverage   *Scaled average abundance*

Description

Compute the scaled average abundance for each feature.

Usage

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

Arguments

y	a DGEList object
scale	a numeric scalar vector indicating the magnitude with which each abundance is to be downscaled
prior.count	a numeric scalar specifying the prior count to add
dispersion	a numeric scalar or vector specifying the dispersion for GLM fitting.

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.

This function mimics the behaviour of aveLogCPM but handles the prior.count with some subtlety. Specifically, it scales up the prior count by scale before adding it to the counts. 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, regardless of the read density.

An additional difference from aveLogCPM is that the prior count is *not* scaled up before being added to the library sizes/offsets. (See addPriorCount for more details.) This ensures that the modified offsets do not depend on scale, which allows abundances to be compared between regions of differing size. Otherwise, larger regions with greater scale would always have (slightly) larger modified offsets and lower abundances than appropriate.

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, addPriorCount

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   *SummarizedExperiment to DGEList*

Description

Converting a SummarizedExperiment object to a DGEList object for analysis with **edgeR**.

Usage

```
## S4 method for signature 'SummarizedExperiment'
asDGEList(object, lib.sizes, norm.factors, assay=1, ...)
```

Arguments

`object`	A SummarizedExperiment object or its derived classes, like that produced by [windowCounts](#).
`lib.sizes`	An (optional) integer vector of library sizes.
`norm.factors`	An (optional) numeric vector of normalization factors.
`assay`	A numeric scalar indicating which assay in `object` contains the count matrix.
`...`	Other arguments to be passed to [DGEList](#).

Details

Counts are extracted from specified `assay` matrix in the SummarizedExperiment object and used to construct a DGEList object via [DGEList](#). If not specified in `lib.sizes`, library sizes are taken from the `totals` field in the column data of `object`. Warnings will be generated if this field is not present.

If `norm.factors` is not specified, asDGEList will attempt to extract normalization factors from `object$norm.factors`. If this is not available, factors will be set to the default (all unity). If `assays(object)$offset` is present, this will be assigned to the `offset` field of the output DGEList object.

Value

A DGEList object is returned containing counts and normalization information.

Author(s)

Aaron Lun

See Also

[DGEList](#)

Examples

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

asDGEList(data)
asDGEList(data, lib.sizes=c(10, 100))
asDGEList(data, norm.factors=c(1.11, 2.23), group=c("a", "b"))

norm.data <- normOffsets(data, se.out=TRUE)
norm.data$norm.factors
asDGEList(norm.data)$samples$norm.factors
asDGEList(norm.data, norm.factors=c(1,2))$samples$norm.factors # gets priority
norm.data <- normOffsets(data, type="loess", se.out=TRUE)
head(asDGEList(norm.data)$offset)
```

---

strandedCounts   *Get strand-specific counts*

Description

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

Usage

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

Arguments

bam.files   a character vector containing paths to sorted and indexed BAM files

param   a readParam object containing read extraction parameters, 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 RangedSummarizedExperiment 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 metadata.

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 RangedSummarizedExperiment 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))

# Library sizes should be the same as that without strand-specificity.
colData(out)
out.ref <- windowCounts(bamFiles, param=reform(xparam, forward=NA))
stopifnot(identical(out.ref$totals, out$totals))

# Running assorted functions on strandedCounts output.
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 or factor 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. Any windows with NA values for ids or summits will have a weight of zero.

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 a list of two integer scalars/vectors, 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	a logical scalar indicating whether binning should be performed
param	a readParam object containing read extraction parameters

Value

A RangedSummarizedExperiment 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 (rlen) and extension lengths (ext) for each library. Other window counting parameters (e.g., spacing, width, param) are stored in the metadata.

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 defined 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. This specifies whether reads are to be read in single- or paired-end mode, whether to apply a threshold to the mapping quality, and so on – see ?readParam for details. 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.

Fragments are inferred from reads by directional extension in single-end data (see below) or by identifying proper pairs in paired-end data (see readParam and getPESizes 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.

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. No extension is performed if `ext` is set to `NA`, such that the read length is used as the fragment length in that library.

If libraries have different fragment lengths, this can be accommodated by supplying a list of 2 elements to `ext`. The first element (named `init.ext` here, for convenience) should be an integer vector specifying the extension length for each library. The second element (`final.ext`) should be an integer scalar specifying the final fragment length. All reads are directionally extended by `init.ext`, and the resulting fragment is resized to `final.ext` by shrinking or expanding from the fragment midpoint. 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.

If any element of `init.ext` is `NA`, no extension is performed for the corresponding library. If `final.ext` is set to `NA`, no rescaling is performed from the library-specific fragment lengths. Values of `init.ext` are stored as the `ext` field in the `colData` of the output object, while `final.ext` is stored in the `metadata`.

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 will use the second element of `ext` as `final.ext`, if `ext` is specified as a list of length 2.

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, the `ext` field in `colData` is set to the average of the inferred fragment lengths for valid read pairs. These values will not be used in `windowCounts`, but instead, in functions like `getWidths`.

Author(s)

Aaron Lun

See Also

correlateReads, readParam, getPESizes

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)

# Multiple extension lengths.
windowCounts(bamFiles, ext=list(c(50, 100), NA), filter=1)
```

```
windowCounts(bamFiles, ext=list(c(50, 100), 80), filter=1)

# 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)
```

---

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, 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
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 and ext should be the same as those used in [profileSites](#) (set ext to the median fragment length for paired-end data). 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 after aggregation across many windows.

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