Package ‘CAGEfightR’

April 27, 2025

Type Package

Title Analysis of Cap Analysis of Gene Expression (CAGE) data using Bioconductor

Version 1.28.0

Author Malte Thodberg

Maintainer Malte Thodberg <maltethodberg@gmail.com>

Description CAGE is a widely used high throughput assay for measuring transcription start site (TSS) activity. CAGEfightR is an R/Bioconductor package for performing a wide range of common data analysis tasks for CAGE and 5'-end data in general.
Core functionality includes: import of CAGE TSSs (CTSSs), tag (or unidirectional) clustering for TSS identification, bidirectional clustering for enhancer identification, annotation with transcript and gene models, correlation of TSS and enhancer expression, calculation of TSS shapes, quantification of CAGE expression as expression matrices and genome brower visualization.

URL https://github.com/MalteThodberg/CAGEfightR

BugReports https://github.com/MalteThodberg/CAGEfightR/issues

Depends R (>= 3.5), GenomicRanges (>= 1.30.1), rtracklayer (>= 1.38.2), SummarizedExperiment (>= 1.8.1)

Imports pryr(>= 0.1.3), assertthat(>= 0.2.0), methods(>= 3.6.3), Matrix(>= 1.2-12), BiocGenerics(>= 0.24.0), S4Vectors(>= 0.16.0), IRanges(>= 2.12.0), GenomeInfoDb(>= 1.14.0), GenomicFeatures(>= 1.29.11), GenomicAlignments(>= 1.22.1), BiocParallel(>= 1.12.0), GenomicFiles(>= 1.14.0), Gviz(>= 1.22.2), InteractionSet(>= 1.9.4), GenomicInteractions(>= 1.15.1)

License GPL-3 + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 7.1.2

Suggests knitr, rmarkdown, BiocStyle, org.Mm.eg.db, TxDb.Mmusculus.UCSC.mm9.knownGene

VignetteBuilder knitr

biocViews Software, Transcription, Coverage, GeneExpression, GeneRegulation, PeakDetection, DataImport, DataRepresentation, Transcriptomics, Sequencing, Annotation, GenomeBrowsers, Normalization, Preprocessing, Visualization

git_url https://git.bioconductor.org/packages/CAGEfightR

git_branch RELEASE_3_21

git_last_commit f30b594

git_last_commit_date 2025-04-15

Repository Bioconductor 3.21

Date/Publication 2025-04-27

Contents

`assignGeneID` *Annotate ranges with gene ID.*

Description

Annotate a set of ranges in a GRanges object with gene IDs (i.e. Entrez Gene Identifiers) based on their genic context. Features overlapping multiple genes are resolved by distance to the nearest TSS. Genes are obtained from a TxDb object, or can manually supplied as a GRanges.

Usage

```
assignGeneID(object, geneModels, ...)

## S4 method for signature 'GenomicRanges,GenomicRanges'
assignGeneID(
  object,
  geneModels,
  outputColumn = "geneID",
  swap = NULL,
```

```
  upstream = 1000,
  downstream = 100
)

## S4 method for signature 'RangedSummarizedExperiment,GenomicRanges'
assignGeneID(object, geneModels, ...)

## S4 method for signature 'GenomicRanges,TxDb'
assignGeneID(
  object,
  geneModels,
  outputColumn = "geneID",
  swap = NULL,
  upstream = 1000,
  downstream = 100
)

## S4 method for signature 'RangedSummarizedExperiment,TxDb'
assignGeneID(object, geneModels, ...)
```

Arguments

`object`	GRanges or RangedSummarizedExperiment: Ranges to be annotated.
`geneModels`	TxDb or GRanges: Gene models via a TxDb, or manually specified as a GRangesList.
`...`	additional arguments passed to methods.
`outputColumn`	character: Name of column to hold geneID values.
`swap`	character or NULL: If not NULL, use another set of ranges contained in object to calculate overlaps, for example peaks in the thick column.
`upstream`	integer: Distance to extend annotated promoter upstream.
`downstream`	integer: Distance to extend annotated promoter downstream.

Value

object with geneID added as a column in rowData (or mcols).

See Also

Other Annotation functions: `assignMissingID()`, `assignTxID()`, `assignTxType()`

Examples

```
data(exampleUnidirectional)

# Obtain gene models from a TxDb-object:
library(TxDb.Mmusculus.UCSC.mm9.knownGene)
txdb <- TxDb.Mmusculus.UCSC.mm9.knownGene
```

```
# Assign geneIDs
assignGeneID(exampleUnidirectional,
             geneModels=txdb,
             outputColumn='geneID')

# Assign geneIDs using only TC peaks:
assignGeneID(exampleUnidirectional,
             geneModels=txdb,
             outputColumn='geneID',
             swap='thick')
```

`assignMissingID` *Annotate ranges with missing IDs.*

Description

This function can relabel ranges with missing IDs (i.e. returned by assignTxID and assignGeneID), in case they need to be retained for further analysis.

Usage

```
assignMissingID(object, ...)

## S4 method for signature 'character'
assignMissingID(object, prefix = "Novel")

## S4 method for signature 'GenomicRanges'
assignMissingID(object, outputColumn = "geneID", prefix = "Novel")

## S4 method for signature 'RangedSummarizedExperiment'
assignMissingID(object, outputColumn = "geneID", prefix = "Novel")
```

Arguments

`object`	character, GRanges or RangedSummarizedExperiment: IDs to have NAs replaces with new IDs.
`...`	additional arguments passed to methods.
`prefix`	character: New name to assign to ranges with missing IDs, in the style prefix1, prefix2, etc.
`outputColumn`	character: Name of column to hold txID values.

Value

object with NAs replaced in outputColumn

See Also

Other Annotation functions: `assignGeneID()`, `assignTxID()`, `assignTxType()`

Examples

```
data(exampleUnidirectional)

# Obtain gene models from a TxDb-object:
library(TxDb.Mmusculus.UCSC.mm9.knownGene)
txdb <- TxDb.Mmusculus.UCSC.mm9.knownGene

# Assign geneIDs using only TC peaks:
exampleUnidirectional <- assignGeneID(exampleUnidirectional,
                                      geneModels=txdb,
                                      outputColumn='geneID',
                                      swap='thick')

# Replace NAs with 'Novel'
assignMissingID(exampleUnidirectional)

# Replace NAs with 'NovelTSS'
assignMissingID(exampleUnidirectional, prefix = 'NovelTSS')
```

`assignTxID` *Annotate ranges with transcript ID.*

Description

Annotate a set of ranges in a GRanges object with transcript IDs based on their genic context. All overlapping transcripts are returned. Transcripts are obtained from a TxDb object, or can manually supplied as a GRanges.

Usage

```
assignTxID(object, txModels, ...)

## S4 method for signature 'GenomicRanges,GenomicRanges'
assignTxID(object, txModels, outputColumn = "txID", swap = NULL)

## S4 method for signature 'RangedSummarizedExperiment,GenomicRanges'
assignTxID(object, txModels, ...)

## S4 method for signature 'GenomicRanges,TxDb'
assignTxID(
  object,
  txModels,
  outputColumn = "txID",
  swap = NULL,
  upstream = 1000,
  downstream = 0
)
```

```
## S4 method for signature 'RangedSummarizedExperiment,TxDb'
assignTxID(object, txModels, ...)
```

Arguments

`object`	GRanges or RangedSummarizedExperiment: Ranges to be annotated.
`txModels`	TxDb or GRanges: Transcript models via a TxDb, or manually specified as a GRanges.
`...`	additional arguments passed to methods.
`outputColumn`	character: Name of column to hold txID values.
`swap`	character or NULL: If not NULL, use another set of ranges contained in object to calculate overlaps, for example peaks in the thick column.
`upstream`	integer: Distance to extend annotated promoter upstream.
`downstream`	integer: Distance to extend annotated promoter downstream.

Value

object with txID added as a column in rowData (or mcols)

See Also

Other Annotation functions: `assignGeneID()`, `assignMissingID()`, `assignTxType()`

Examples

```
data(exampleUnidirectional)

# Obtain transcript models from a TxDb-object:
library(TxDb.Mmusculus.UCSC.mm9.knownGene)
txdb <- TxDb.Mmusculus.UCSC.mm9.knownGene

# Assign txIDs
assignTxID(exampleUnidirectional,
           txModels=txdb,
           outputColumn='geneID')

# Assign txIDs using only TC peaks:
assignTxID(exampleUnidirectional,
             txModels=txdb,
             outputColumn='geneID',
             swap='thick')
```

`assignTxType` *Annotate ranges with transcript type.*

Description

Annotate a set of ranges in a GRanges object with transcript type (txType) based on their genic context. Transcripts are obtained from a TxDb object, but can alternatively be specified manually as a GRangesList.

Usage

```
assignTxType(object, txModels, ...)

## S4 method for signature 'GenomicRanges,GenomicRangesList'
assignTxType(
  object,
  txModels,
  outputColumn = "txType",
  swap = NULL,
  noOverlap = "intergenic"
)

## S4 method for signature 'RangedSummarizedExperiment,GenomicRangesList'
assignTxType(object, txModels, ...)

## S4 method for signature 'GenomicRanges,TxDb'
assignTxType(
  object,
  txModels,
  outputColumn = "txType",
  swap = NULL,
  tssUpstream = 100,
  tssDownstream = 100,
  proximalUpstream = 1000,
  detailedAntisense = FALSE
)

## S4 method for signature 'RangedSummarizedExperiment,TxDb'
assignTxType(object, txModels, ...)
```

Arguments

`object`	GRanges or RangedSummarizedExperiment: Ranges to be annotated.
`txModels`	TxDb or GRangesList: Transcript models via a TxDb, or manually specified as a GRangesList.
`...`	additional arguments passed to methods.

outputColumn	character: Name of column to hold txType values.
swap	character or NULL: If not NULL, use another set of ranges contained in object to calculate overlaps, for example peaks in the thick column.
noOverlap	character: In case categories are manually supplied with as a GRangesList, what to call regions with no overlap.
tssUpstream	integer: Distance to extend annotated promoter upstream.
tssDownstream	integer: Distance to extend annotated promoter downstream.
proximalUpstream	integer: Maximum distance upstream of promoter to be considered proximal.
detailedAntisense	logical: Wether to mirror all txType categories in the antisense direction (TRUE) or lump them all together (FALSE).

Value

object with txType added as factor column in rowData (or mcols)

See Also

Other Annotation functions: `assignGeneID()`, `assignMissingID()`, `assignTxID()`

Examples

```
## Not run:
data(exampleUnidirectional)

# Obtain transcript models from a TxDb-object:
library(TxDb.Mmusculus.UCSC.mm9.knownGene)
txdb <- TxDb.Mmusculus.UCSC.mm9.knownGene

# Assign txIDs
assignTxType(exampleUnidirectional,
             txModels=txdb)

# Assign txIDs using only TC peaks:
exampleUnidirectional <- assignTxType(exampleUnidirectional,
                                      txModels=txdb,
                                      swap='thick')

# The 'promoter' and 'proximal' category boundaries can be changed:
assignTxType(exampleUnidirectional,
             txModels=txdb,
             swap='thick',
             tssUpstream=50,
             tssDownstream=50,
             proximalUpstream=100)

# Annotation using complete antisense categories:
exampleUnidirectional <- assignTxType(exampleUnidirectional,
                                    txModels=txdb,
```

```
                                        outputColumn='txTypeExtended',
                                        swap='thick',
                                        detailedAntisense=TRUE)

# The output is always a factor added as a column:
summary(rowRanges(exampleUnidirectional)$txType)
summary(rowRanges(exampleUnidirectional)$txTypeExtended)

# To avoid using a TxDb-object, a GRangesList can be supplied:
custom_hierarchy <- GRangesList(promoters=granges(promoters(txdb)),
                                exons=granges(exons(txdb)))
assignTxType(exampleUnidirectional,
             txModels=custom_hierarchy,
             outputColumn='customType',
             swap='thick',
             noOverlap = 'intergenic')

## End(Not run)
```

---

balanceBC *Balance statistic: Bhattacharyya coefficient (BC)*

Description

Calculates the Bhattacharyya coefficient from observed plus/minus upstream/downstream signals to a perfect bidirectional site, where plus-downstream = 50

Usage

```
balanceBC(PD, MD, PU, MU)
```

Arguments

PD	Plus-Downstream signal
MD	Minus-Downstream signal
PU	Plus-Upstream signal
MU	Plus-Upstream signal

Value

Balance score of the same class as inputs.

Examples

```
# Unbalanced
balanceBC(2, 3, 1, 0)

# Balanced
balanceBC(3, 3, 0, 0)
```

`balanceD` *Balance statistic: Andersson's D.*

Description

Calculates the D-statistics from Andersson et al the observed plus/minus downstream signals. The D statistics is rescaled from -1:1 to 0:1 so it can be used for slice-reduce identification of bidirectional sites.

Usage

```
balanceD(PD, MD, PU, MU)
```

Arguments

PD	Plus-Downstream signal
MD	Minus-Downstream signal
PU	Plus-Upstream signal
MU	Plus-Upstream signal

Value

Balance score of the same class as inputs.

Examples

```
# Unbalanced
balanceD(2, 3, 1, 0)

# Balanced
balanceD(3, 3, 0, 0)
```

`bwCommonGenome` *Find a common genome for a series of BigWig files.*

Description

Finds a common genome for a series of BigWig-files, either using only levels present in all files (intersect) or in any file (union).

Usage

```
bwCommonGenome(plusStrand, minusStrand, method = "intersect")
```

Arguments

plusStrand	BigWigFileList: BigWig files with plus-strand CTSS data.
minusStrand	BigWigFileList: BigWig files with minus-strand CTSS data.
method	character: Either 'intersect' or 'union'.

Value

Sorted Seqinfo-object.

See Also

Other BigWig functions: bwGenomeCompatibility(), bwValid()

Examples

```
if (.Platform$OS.type != "windows") {
# Use the BigWig-files included with the package:
data('exampleDesign')
bw_plus <- system.file('extdata', exampleDesign$BigWigPlus,
                       package = 'CAGEfightR')
bw_minus <- system.file('extdata', exampleDesign$BigWigMinus,
                        package = 'CAGEfightR')

# Create two named BigWigFileList-objects:
bw_plus <- BigWigFileList(bw_plus)
bw_minus <- BigWigFileList(bw_minus)
names(bw_plus) <- exampleDesign$Name
names(bw_minus) <- exampleDesign$Name

# Find the smallest common genome (intersect) across the BigWigList-objects:
bwCommonGenome(plusStrand=bw_plus, minusStrand=bw_minus, method='intersect')

# Find the most inclusive genome (union) across the BigWigList-objects:
bwCommonGenome(plusStrand=bw_plus, minusStrand=bw_minus, method='union')
}
```

---

bwGenomeCompatibility *Check if BigWig-files are compatible with a given genome.*

Description

Given a genome, checks whether a series of BigWig-files are compatible by checking if all common seqlevels have equal seqlengths.

Usage

```
bwGenomeCompatibility(plusStrand, minusStrand, genome)
```

Arguments

plusStrand	BigWigFileList: BigWig files with plus-strand CTSS data.
minusStrand	BigWigFileList: BigWig files with minus-strand CTSS data.
genome	Seqinfo: Genome information.

Value

TRUE, raises an error if the supplied genome is incompabtible.

See Also

Other BigWig functions: `bwCommonGenome()`, `bwValid()`

Examples

```
if (.Platform$OS.type != "windows") {
# Use the BigWig-files included with the package:
data('exampleDesign')
bw_plus <- system.file('extdata', exampleDesign$BigWigPlus,
                       package = 'CAGEfightR')
bw_minus <- system.file('extdata', exampleDesign$BigWigMinus,
                        package = 'CAGEfightR')

# Create two named BigWigFileList-objects:
bw_plus <- BigWigFileList(bw_plus)
bw_minus <- BigWigFileList(bw_minus)
names(bw_plus) <- exampleDesign$Name
names(bw_minus) <- exampleDesign$Name

# Make a smaller genome:
si <- seqinfo(bw_plus[[1]])
si <- si['chr18']

# Check if it is still compatible:
bwGenomeCompatibility(plusStrand=bw_plus, minusStrand=bw_minus, genome=si)
}
```

---

bwValid *Check if BigWig-files are valid.*

Description

Checks if a BigWigFile or BigWigFileList is composed of readable files with the proper .bw extension.

Usage

```
bwValid(object)

## S4 method for signature 'BigWigFile'
bwValid(object)

## S4 method for signature 'BigWigFileList'
bwValid(object)
```

Arguments

`object`	BigWigFile or BigWigFileList

Value

TRUE, if any tests fails an error is raised.

See Also

Other BigWig functions: `bwCommonGenome()`, `bwGenomeCompatibility()`

Examples

```
# Use the BigWig-files included with the package:
data('exampleDesign')
bw_plus <- system.file('extdata', exampleDesign$BigWigPlus,
                       package = 'CAGEfightR')

# Create a named BigWigFileList-object with names
bw_plus <- BigWigFileList(bw_plus)
names(bw_plus) <- exampleDesign$Name

# Check a single BigWigFile:
bwValid(bw_plus[[1]])

# Check the entire BigWigFileList:
bwValid(bw_plus)
```

`calcBidirectionality` *Calculate sample-wise bidirectionally of clusters.*

Description

For each cluster, calculate how many individual samples shows transcription in both directions. This is refered to as the 'bidirectionality'. Clusters must be unstranded (*) and have a midpoint stored in the thick column

Usage

```
calcBidirectionality(object, ...)

## S4 method for signature 'GRanges'
calcBidirectionality(
  object,
  samples,
  inputAssay = "counts",
  outputColumn = "bidirectionality"
)

## S4 method for signature 'GPos'
calcBidirectionality(object, ...)

## S4 method for signature 'RangedSummarizedExperiment'
calcBidirectionality(object, ...)
```

Arguments

`object`	GenomicRanges or RangedSummarizedExperiment: Unstranded clusters with midpoints stored in the 'thick' column.
`...`	additional arguments passed to methods.
`samples`	RangedSummarizedExperiment: Sample-wise CTSSs stored as an assay.
`inputAssay`	character: Name of assay in samples holding input CTSS values.
`outputColumn`	character: Name of column in object to hold bidirectionality values.

Value

object returned with bidirectionality scores added in rowData (or mcols).

See Also

Other Calculation functions: `calcComposition()`, `calcPooled()`, `calcShape()`, `calcSupport()`, `calcTPM()`, `calcTotalTags()`, `subsetByBidirectionality()`, `subsetByComposition()`, `subsetBySupport()`

Examples

```
data(exampleCTSSs)
data(exampleBidirectional)

calcBidirectionality(exampleBidirectional, samples=exampleCTSSs)
```

calcComposition *Calculate composition of CAGE data.*

Description

For every feature, count in how many samples it is expressed above a certain fraction (e.g. 10 percent) within a grouping, usually genes. This count is refered to as the 'composition' value.

Usage

```
calcComposition(
  object,
  inputAssay = "counts",
  outputColumn = "composition",
  unexpressed = 0.1,
  genes = "geneID"
)
```

Arguments

`object`	RangedSummarizedExperiment: CAGE data quantified at CTSS, cluster or gene-level.
`inputAssay`	character: Name of assay holding input expression values.
`outputColumn`	character: Name of column in rowRanges to hold composition values.
`unexpressed`	numeric: Composition will be calculated based on features larger than this cut-off.
`genes`	character: Name of column in rowData holding genes (NAs are not currently allowed.)

Value

object with composition added as a column in rowData.

See Also

Other Calculation functions: `calcBidirectionality()`, `calcPooled()`, `calcShape()`, `calcSupport()`, `calcTPM()`, `calcTotalTags()`, `subsetByBidirectionality()`, `subsetByComposition()`, `subsetBySupport()`

Examples

```
data(exampleUnidirectional)

# Annotate clusters with geneIDs:
library(TxDb.Mmusculus.UCSC.mm9.knownGene)
txdb <- TxDb.Mmusculus.UCSC.mm9.knownGene
exampleUnidirectional <- assignGeneID(exampleUnidirectional,
                                      geneModels=txdb,
```

```
                                          outputColumn='geneID',
                                          swap='thick')

# Calculate composition values:
exampleUnidirectional <- subset(exampleUnidirectional, !is.na(geneID))
calcComposition(exampleUnidirectional)

# Use a lower threshold
calcComposition(exampleUnidirectional,
                unexpressed=0.05,
                outputColumn='lenientComposition')
```

---

calcPooled *Calculate pooled expression across all samples.*

Description

Sum expression of features across all samples to obtain a 'pooled' signal.

Usage

```
calcPooled(object, inputAssay = "TPM", outputColumn = "score")
```

Arguments

`object`	RangedSummarizedExperiment: CAGE data quantified at CTSS, cluster or gene-level.
`inputAssay`	character: Name of assay holding input expression values.
`outputColumn`	character: Name of column in rowRanges to hold pooled expression.

Value

object with pooled expression added as a column in rowRanges.

See Also

Other Calculation functions: `calcBidirectionality()`, `calcComposition()`, `calcShape()`, `calcSupport()`, `calcTPM()`, `calcTotalTags()`, `subsetByBidirectionality()`, `subsetByComposition()`, `subsetBySupport()`

Examples

```
data(exampleCTSSs)

# Calculate TPM using supplied total number of tags:
exampleCTSSs <- calcTPM(exampleCTSSs, totalTags='totalTags')

# Sum TPM values over samples:
calcPooled(exampleCTSSs)
```

calcShape *Calculate Tag Cluster shapes*

Description

Apply a shape-function to the pooled CTSS signal of every Tag Cluster (TC).

Usage

```
calcShape(object, pooled, ...)

## S4 method for signature 'GRanges,GRanges'
calcShape(object, pooled, outputColumn = "IQR", shapeFunction = shapeIQR, ...)

## S4 method for signature 'RangedSummarizedExperiment,GRanges'
calcShape(object, pooled, ...)

## S4 method for signature 'GRanges,RangedSummarizedExperiment'
calcShape(object, pooled, ...)

## S4 method for signature 'GRanges,GPos'
calcShape(object, pooled, ...)

## S4 method for signature
## 'RangedSummarizedExperiment,RangedSummarizedExperiment'
calcShape(object, pooled, ...)
```

Arguments

`object`	GenomicRanges or RangedSummarizedExperiment: TCs.
`pooled`	GenomicRanges or RangedSummarizedExperiment: Pooled CTSS as the score column.
`...`	additional arguments passed to shapeFunction.
`outputColumn`	character: Name of column to hold shape statistics.
`shapeFunction`	function: Function to apply to each TC (See details).

Value

object with calculated shape statistics added as a column in rowData (or mcols).

See Also

Other Calculation functions: `calcBidirectionality()`, `calcComposition()`, `calcPooled()`, `calcSupport()`, `calcTPM()`, `calcTotalTags()`, `subsetByBidirectionality()`, `subsetByComposition()`, `subsetBySupport()`

Other Shape functions: `shapeEntropy()`, `shapeIQR()`, `shapeMean()`

Examples

```
data(exampleCTSSs)
data(exampleUnidirectional)

# Calculate pooled CTSSs using pre-calculated number of total tags:
exampleCTSSs <- calcTPM(exampleCTSSs, totalTags='totalTags')
exampleCTSSs <- calcPooled(exampleCTSSs)

# Calculate shape statistics
calcShape(exampleUnidirectional, pooled=exampleCTSSs,
    outputColumn='entropy', shapeFunction=shapeEntropy)
calcShape(exampleUnidirectional, pooled=exampleCTSSs, outputColumn='IQR',
    shapeFunction=shapeIQR, lower=0.2, upper=0.8)

# See the vignette for how to implement custom shape functions!
```

`calcSupport` *Calculate support of CAGE data.*

Description

Calculate the number of samples expression a feature above a certain level. This number is refered to as the 'support'.

Usage

```
calcSupport(
  object,
  inputAssay = "counts",
  outputColumn = "support",
  unexpressed = 0
)
```

Arguments

`object`	RangedSummarizedExperiment: CAGE data quantified at CTSS, cluster or gene-level.
`inputAssay`	character: Name of assay holding input expression values.
`outputColumn`	character: Name of column in rowRanges to hold support values.
`unexpressed`	numeric: Support will be calculated based on features larger than this cutoff.

Value

object with support added as a column in rowRanges.

See Also

Other Calculation functions: calcBidirectionality(), calcComposition(), calcPooled(), calcShape(), calcTPM(), calcTotalTags(), subsetByBidirectionality(), subsetByComposition(), subsetBySupport()

Examples

```
data(exampleUnidirectional)

# Count samples with at at least a single tags
exampleUnidirectional <- calcSupport(exampleUnidirectional,
                                     inputAssay='counts',
                                     unexpressed=0)

# Count number of samples with more than 1 TPM and save as a new column.
exampleUnidirectional <- calcTPM(exampleUnidirectional,
                                 totalTags = 'totalTags')
exampleUnidirectional <- calcSupport(exampleUnidirectional,
                                     inputAssay='TPM',
                                     unexpressed=1,
                                     outputColumn='TPMsupport')
```

---

calcTotalTags *Calculate the total number of CAGE tags across samples.*

Description

For each CAGE library, calculate the total number of tags.

Usage

```
calcTotalTags(object, inputAssay = "counts", outputColumn = "totalTags")
```

Arguments

object	RangedSummarizedExperiment: CAGE data quantified at CTSS, cluster or gene-level.
inputAssay	character: Name of assay holding input expression values.
outputColumn	character: Name of column in colData to hold number of total tags.

Value

object with total tags per library added as a column in colData.

See Also

Other Calculation functions: calcBidirectionality(), calcComposition(), calcPooled(), calcShape(), calcSupport(), calcTPM(), subsetByBidirectionality(), subsetByComposition(), subsetBySupport()

Examples

```
data(exampleUnidirectional)
calcTotalTags(exampleUnidirectional)
```

---

calcTPM *Calculate CAGE Tags-Per-Million (TPM)*

Description

Normalize CAGE-tag counts into TPM values.

Usage

```
calcTPM(
  object,
  inputAssay = "counts",
  outputAssay = "TPM",
  totalTags = NULL,
  outputColumn = "totalTags"
)
```

Arguments

`object`	RangedSummarizedExperiment: CAGE data quantified at CTSS, cluster or gene-level.
`inputAssay`	character: Name of assay holding input expression values.
`outputAssay`	character: Name of assay to hold TPM values.
`totalTags`	character or NULL: Column in colData holding the total number of tags for each samples. If NULL, this will be calculated using calcTotalTags.
`outputColumn`	character: Name of column in colData to hold number of total tags, only used if totalTags is NULL.

Value

object with TPM-values added as a new assay. If totalTags is NULL, total tags added as a column in colData.

See Also

Other Calculation functions: `calcBidirectionality()`, `calcComposition()`, `calcPooled()`, `calcShape()`, `calcSupport()`, `calcTotalTags()`, `subsetByBidirectionality()`, `subsetByComposition()`, `subsetBySupport()`

Examples

```
data(exampleUnidirectional)

# Calculate TPM:
calcTPM(exampleUnidirectional)

# Use pre-calculated total number of tags:
calcTPM(exampleUnidirectional,
        outputAssay='TPMsupplied',
        totalTags='totalTags')
```

---

checkCTSSs *Helper for checking files containing CTSSs*

Description

Checks whether a file (or GRanges/GPos) contains data formatted in the same manner as CAGE Transcription Start Sites (CTSSs): Each basepair of the genome is associated with a single integer count.

Usage

```
checkCTSSs(object)

## S4 method for signature 'ANY'
checkCTSSs(object)

## S4 method for signature 'GRanges'
checkCTSSs(object)

## S4 method for signature 'character'
checkCTSSs(object)

## S4 method for signature 'GPos'
checkCTSSs(object)

## S4 method for signature 'BigWigFile'
checkCTSSs(object)
```

Arguments

`object`	BigWigFile, character, GRanges or GPos: Path to the file storing CTSSs, or an already improted GRanges/GPos.

Value

TRUE if CTSSs are correctly formatted, otherwise a (hopefully) informative error is thrown.

Note

In the case that a character is supplied pointing to a file, checkCTSSs will not check any extensions, but simply try to read it using rtracklayer::import. This means that checkCTSSs can technically analyze BED-files, although CAGEfightR can only import CTSSs from BigWig or bedGraph files.

Examples

```
if (.Platform$OS.type != "windows") {
# Load example data
data('exampleDesign')
bw_plus <- system.file('extdata',
                       exampleDesign$BigWigPlus,
                       package = 'CAGEfightR')
bw_plus <- BigWigFileList(bw_plus)

# Check raw file
checkCTSSs(bw_plus[[1]])

# Import first, then check
gr  <- import(bw_plus[[1]])
checkCTSSs(gr)
}
```

---

checkPeaked   *Helper for checking cluster with peaks*

Description

Checks whether a supplied set of cluster have valid peaks: Whether the thick column contains IRanges all contained within the main ranges.

Usage

```
checkPeaked(object)
```

Arguments

`object`	GRanges or GPos: Clusters with peaks to be checked.

Value

TRUE if object is correct format, otherwise an error is thrown

See Also

Other Checking functions: `checkPooled()`

Examples

```
data(exampleUnidirectional)
checkPeaked(rowRanges(exampleUnidirectional))
```

`checkPooled` *Helper for checking pooled signal*

Description

Checks whether a supplied pooled signal is valid: Single bp disjoint with signal in the score column with supplied genome information.

Usage

```
checkPooled(object)
```

Arguments

`object`	GRanges or GPos: Pooled signal to be checked

Value

TRUE if object is correct format, otherwise an error is thrown

See Also

Other Checking functions: `checkPeaked()`

Examples

```
data(exampleCTSSs)
checkPooled(rowRanges(exampleCTSSs))
```

`clusterBidirectionally` *Bidirectional clustering of pooled CTSSs.*

Description

Finds sites with (balanced and divergent) bidirectional transcription using sliding windows of summed coverage: The Bhattacharyya coefficient (BC) is used to quantify depature from a perfectly balanced site, and a slice-reduce is used to identify sites.

Usage

```
clusterBidirectionally(object, ...)

## S4 method for signature 'GRanges'
clusterBidirectionally(
  object,
  window = 201,
  balanceThreshold = 0.95,
  balanceFun = balanceBC
)

## S4 method for signature 'GPos'
clusterBidirectionally(object, ...)

## S4 method for signature 'RangedSummarizedExperiment'
clusterBidirectionally(object, ...)
```

Arguments

`object`	GenomicRanges or RangedSummarizedExperiment: Pooled CTSSs stored in the score column.
`...`	additional arguments passed to methods.
`window`	integer: Width of sliding window used for calculating window sums.
`balanceThreshold`	numeric: Minimum value of the BC to use for slice-reduce, a value of 1 corresponds to perfectly balanced sites.
`balanceFun`	function: Advanced users may supply their own function for calculating the balance score instead of the the default balanceBC. See details for instructions.

Value

GRanges with bidirectional sites: Minimum width is 1 + 2*window, TPM sum (on both strands) in the score column, maximal bidirectional site in the thick column and maximum balance in the balance column.

See Also

Other Clustering functions: `clusterUnidirectionally()`, `trimToPeak()`, `trimToPercentiles()`, `tuneTagClustering()`

Examples

```
## Not run:
data(exampleCTSSs)

# Calculate pooledTPM, using supplied number of total tags
exampleCTSSs <- calcTPM(exampleCTSSs,
                        inputAssay='counts',
```

```
                        outputAssay='TPM',
                        totalTags='totalTags')
exampleCTSSs <- calcPooled(exampleCTSSs, inputAssay='TPM')

# Cluster using defaults: balance-treshold of 199 and window of 199 bp:
clusterBidirectionally(exampleCTSSs)

# Use custom thresholds:
clusterBidirectionally(exampleCTSSs, balanceThreshold=0.99, window=101)

## End(Not run)
```

---

clusterUnidirectionally *Unidirectional Clustering (Tag Clustering) of pooled CTSSs.*

Description

Finds unidirectional Tag Clusters (TCs) with a pooled TPM above a certain threshold using a slice-reduce approach. Addtionally calculates the sum and peak position of the TCs.

Usage

```
clusterUnidirectionally(object, ...)

## S4 method for signature 'GRanges'
clusterUnidirectionally(object, pooledCutoff = 0, mergeDist = 20L)

## S4 method for signature 'RangedSummarizedExperiment'
clusterUnidirectionally(object, ...)

## S4 method for signature 'GPos'
clusterUnidirectionally(object, ...)
```

Arguments

`object`	GRanges or RangedSummarizedExperiment: Basepair-wise pooled CTSS.
`...`	additional arguments passed to methods.
`pooledCutoff`	numeric: Minimum pooled value to be considered as TC.
`mergeDist`	integer: Merge TCs within this distance.

Value

GRanges with TPM sum as the score column, and TC peak as the thick column.

See Also

Other Clustering functions: `clusterBidirectionally()`, `trimToPeak()`, `trimToPercentiles()`, `tuneTagClustering()`

Examples

```
data(exampleCTSSs)

# Calculate pooledTPM, using supplied number of total tags
exampleCTSSs <- calcTPM(exampleCTSSs,
                        inputAssay='counts',
                        outputAssay='TPM',
                        totalTags='totalTags')
exampleCTSSs <- calcPooled(exampleCTSSs, inputAssay='TPM')

# Cluster using defaults: slice-threshold of 0 and reduce-distance of 20
clusterUnidirectionally(exampleCTSSs)

# Use custom thresholds:
clusterUnidirectionally(exampleCTSSs, pooledCutoff=1, mergeDist=25)
```

`combineClusters` *Combine two CAGE experiments.*

Description

This function can safely combine two CAGE experiments, for example TCs and enhancers, for later analysis, by making sure no ranges in the final object are overlapping.

Usage

```
combineClusters(object1, object2, ...)

## S4 method for signature
## 'RangedSummarizedExperiment,RangedSummarizedExperiment'
combineClusters(object1, object2, removeIfOverlapping = "none")
```

Arguments

`object1`	RangedSummarizedExperiment: First experiment to be combined.
`object2`	RangedSummarizedExperiment: First experiment to be combined.
`...`	arguments passed to methods.
`removeIfOverlapping`	character: Whether to keep overlapping ranges ('none') or discard from either the first ('object1') or second ('object2') experiment.

Value

RangedSummarizedExperiment with merged and sorted ranges (colData and metadata are carried over unchanged).

Examples

```
data(exampleUnidirectional)
data(exampleBidirectional)

# Clusters must have identical colData to be combined:
exampleUnidirectional$totalTags <- NULL

# Combine, keeping potential overlaps
combineClusters(object1=exampleUnidirectional, object2=exampleBidirectional)

# If features overlap, keep only from object1
combineClusters(object1=exampleUnidirectional, object2=exampleBidirectional,
   removeIfOverlapping='object2')

# If features overlap, keep only from object2
combineClusters(object1=exampleUnidirectional, object2=exampleBidirectional,
   removeIfOverlapping='object1')
```

---

convertBAM2BigWig *Extract CTSSs from BAM-files (EXPERIMENTAL)*

Description

Function for converting mapped reads in BAM-files to CAGE Transcription Start Sites (CTSSs) in BigWig-files. Currently, this function will simply load a (single-end) BAM-file (respecting a supplied ScanBamParam), optionally remove short tags, and count the number of 5'-ends at each bp. Note, the BAM-file is loaded as a single object, so you must be able to keep at least one complete BAM-file in RAM.

Usage

```
convertBAM2BigWig(input, outputPlus, outputMinus, minLength = 1L, ...)
```

Arguments

`input`	character: Path to input BAM-file
`outputPlus`	character: Path to output BigWig-file holding CTSSs on the plus strand.
`outputMinus`	character: Path to output BigWig-file holding CTSSs on the minus strand.
`minLength`	integer: Minimum length of mapped reads.
`...`	Additional arguments passed to rtracklayer::import. This will often include a ScanBamParam

Value

Number of CTSSs/Tags returned invisibly.

Note

WARNING: This function is experimental, has not been thoroughly tested, and will most likely significantly change in upcoming CAGEfightR version. For comments/question please go to the CAGEfightR github page.

Examples

```
# TBA
```

`convertBED2BigWig` *Convert CTSSs stored in different file formats.*

Description

Collection of functions for converting CTSSs/CTSSs-like data stored in BigWig, bedGraph or BED file formats. BigWig and bedGraph files use a file for each strand, while BED-files stores both strands in a single file. As BigWig files stores info about the chromosome lenghts, conversion from bedGraph/BED to BigWig requires a genome. Note that CAGEfightR will only import BigWig or bedGraph files!

Usage

```
convertBED2BigWig(input, outputPlus, outputMinus, genome)

convertBED2BedGraph(input, outputPlus, outputMinus)

convertBedGraph2BigWig(input, output, genome)

convertBigWig2BedGraph(input, output)

convertBigWig2BED(inputPlus, inputMinus, output)

convertBedGraph2BED(inputPlus, inputMinus, output)
```

Arguments

`input`	charater: Path to input files holding CTSSs on both strands.
`outputPlus`	character: Path to output files holding CTSSs on plus strand.
`outputMinus`	character: Path to output files holding CTSSs on minus strand.
`genome`	Seqinfo or character: Genome info passed to rtracklayer::import (see note).
`output`	charater: Path to output files holding CTSSs on both strands.
`inputPlus`	character: Path to input files holding CTSSs on plus strand.
`inputMinus`	character: Path to input files holding CTSSs on minus strand.

Value

TRUE returned invisibly if conversion(s) was succesful, otherwise an error is raised.

Note

These functions will warn if input files do not have the correct extensions (.bw, .bedGraph, .bed), but otherwise simply pass input to rtracklayer::import. This makes them able to handle compressed files (like .gz). The same applies to the genome argument, which can also be the name of a UCSC genome.

Examples

```
## Not run:
# Find paths to BigWig files
data('exampleDesign')
bw_plus <- system.file('extdata', exampleDesign$BigWigPlus,
                       package = 'CAGEfightR')
bw_minus <- system.file('extdata', exampleDesign$BigWigMinus,
                        package = 'CAGEfightR')

# Designate paths to new files
n_samples <- length(bw_plus)
beds <- replicate(n=n_samples, tempfile(fileext=".bed"))
bg_plus <- replicate(n=n_samples, tempfile(fileext="_plus.bedGraph"))
bg_minus <- replicate(n=n_samples, tempfile(fileext="_minus.bedGraph"))
conv_plus <- replicate(n=n_samples, tempfile(fileext="_plus.bw"))
conv_minus <- replicate(n=n_samples, tempfile(fileext="_minus.bw"))

# Convert BigWig to BED
convertBigWig2BED(inputPlus=bw_plus,
                  inputMinus=bw_minus,
                  output=beds)

# Convert BED to bedGraph
convertBED2BedGraph(input=beds,
                    outputPlus=bg_plus,
                    outputMinus=bg_minus)

# Convert BED to bedGraph
mm9 <- SeqinfoForUCSCGenome("mm9")
convertBED2BigWig(input=beds,
                  outputPlus=conv_plus,
                  outputMinus=conv_minus,
                  genome=mm9)

# Check it's still the same data
x <- import(bw_plus[1])
y <- import(bg_plus[1])
z <- import(conv_plus[1])
all(x == y)
all(x == z)
sum(score(x)) ==  sum(score(y))
```

```
sum(score(x)) ==  sum(score(z))

## End(Not run)
```

---

convertGRanges2GPos *Convert GRanges with scores to GPos*

Description

Converts a GRanges to a GPos, correctly expanding the score column. This is useful is nearby CTSSs with the same count are grouped in the same range (see example).

Usage

```
convertGRanges2GPos(object)
```

Arguments

`object`	GRanges object with a score column

Value

GPos with score column

Examples

```
# Example GRanges
gr <- GRanges(Rle(c("chr2", "chr2", "chr3", "chr4")),
              IRanges(start=c(1, 10, 5, 3),
              end=c(5L, 10L, 5L, 4L)),
              strand="+",
              score=c(2, 1, 3, 11))

# Expand to proper GPos / CTSS format:
gp <- convertGRanges2GPos(gr)

# Double check that the total number of counts remains the same
stopifnot(sum(score(gr) * width(gr)) == sum(score(gp)))
```

exampleDesign *Example CAGE Data*

Description

Subset of the CAGE dataset from the paper 'Identification of Gene Transcription Start Sites and Enhancers Responding to Pulmonary Carbon Nanotube Exposure in Vivo'. CTSS data from subsets of chr18 and chr19 across 3 mouse (mm9) samples are included. Datasets can be loaded with the data function.

Usage

```
exampleDesign

exampleCTSSs

exampleUnidirectional

exampleBidirectional

exampleGenes
```

Format

Example data from various stages of CAGEfightR:

exampleDesign DataFrame: Description of samples, including .bw filenames
exampleCTSS RangedSummarizedExperiment: CTSSs
exampleUnidirectional RangedSummarizedExperiment: Unidirectional or Tag Clusters
exampleBidirectionalCluster RangedSummarizedExperiment: Bidirectional clusters
exampleGenes RangedSummarizedExperiment: Genes

An object of class `RangedSummarizedExperiment` with 41256 rows and 3 columns.

An object of class `RangedSummarizedExperiment` with 21008 rows and 3 columns.

An object of class `RangedSummarizedExperiment` with 377 rows and 3 columns.

An object of class `RangedSummarizedExperiment` with 127 rows and 3 columns.

Source

http://pubs.acs.org/doi/abs/10.1021/acsnano.6b07533

Examples

```
data(exampleDesign)
data(exampleCTSSs)
data(exampleUnidirectional)
data(exampleBidirectional)
data(exampleGenes)
```

findLinks *Find nearby pairs of clusters and calculate pairwise correlations.*

Description

Finds all links or pairs of clusters within a certain distance of each other and then calculates the correlation between them. The links found can be restricted to only be between two classes, for example TSSs to enhancers.

Usage

```
findLinks(object, ...)

## S4 method for signature 'GRanges'
findLinks(object, maxDist = 10000L, directional = NULL)

## S4 method for signature 'RangedSummarizedExperiment'
findLinks(
  object,
  inputAssay,
  maxDist = 10000L,
  directional = NULL,
  corFun = stats::cor.test,
  vals = c("estimate", "p.value"),
  ...
)
```

Arguments

`object`	GRanges or RangedSummarizedExperiment: Clusters, possibly with expression for calculating correlations.
`...`	additional arguments passed to methods or ultimately corFun.
`maxDist`	integer: Maximum distance between links.
`directional`	character: Name of a column in object holding a grouping of the clusters. This must be a factor with two levels. The first level is used as the basis for calculating orientation (see below).
`inputAssay`	character: Name of assay holding expression values (if object is a RangedSummarizedExperiment)
`corFun`	function: Function for calculating pairwise correlations. See notes for supplying custom functions.
`vals`	character: Statistics extracted from the results produced by corFun. See notes for supplying custom functions.

Details

A custom function for calculation correlations can be supplied by the user. The output of this function must be a named list or vector of numeric values. The names of the vals to be extracted should be supplied to vals.

Value

A GInteractions holding the links, along with the distance between them and correlation estimate and p-value calculated from their expression. If a directional analysis was performed, the two anchors are always connecting members of the two classes and the orientation of the second anchor relative to the first is additionaly calculated (e.g. whether an enhancers is upstream or downstream of the TSS).

See Also

Other Spatial functions: `findStretches()`, `trackLinks()`

Examples

```
library(InteractionSet)

# Subset to highly expressed unidirectional clusters
TCs <- subset(exampleUnidirectional, score > 10)

# Find links within a certain distance
findLinks(TCs, inputAssay="counts", maxDist=10000L)

# To find TSS-to-enhancer type links, first merge the clusters:
colData(exampleBidirectional) <- colData(TCs)
rowRanges(TCs)$clusterType <- "TSS"
rowRanges(exampleBidirectional)$clusterType <- "Enhancer"
SE <- combineClusters(TCs, exampleBidirectional, removeIfOverlapping="object1")
rowRanges(SE)$clusterType <- factor(rowRanges(SE)$clusterType, levels=c("TSS", "Enhancer"))

# Calculate kendall correlations of TPM values:
SE <- calcTPM(SE, totalTags="totalTags")
findLinks(SE, inputAssay="TPM", maxDist=10000L, directional="clusterType", method="kendall")
```

`findStretches` *Find stretches of clusters*

Description

Finds stretches or groups of clusters along the genome, where each cluster is within a certain distance of the next. Once stretches have been identified, the average pairwise correlation between all clusters in the stretch is calculated. A typical use case is to look for stretches of enhancers, often refered to as "super enhancers".

Usage

```
findStretches(object, ...)

## S4 method for signature 'GRanges'
findStretches(object, mergeDist = 10000L, minSize = 3L)

## S4 method for signature 'RangedSummarizedExperiment'
findStretches(
  object,
  inputAssay,
  mergeDist = 10000L,
  minSize = 3L,
  corFun = cor,
  ...
)
```

Arguments

`object`	GRanges or RangedSummarizedExperiment: Clusters, possibly with expression for calculating correlations.
`...`	additional arguments passed to methods or ultimately corFun.
`mergeDist`	integer: Maximum distance between clusters to be merged into stretches.
`minSize`	integer: Minimum number of clusters in stretches.
`inputAssay`	character: Name of assay holding expression values (if object is a RangedSummarizedExperiment)
`corFun`	function: Function for calculating correlations. Should behave and produce output similar to cor().

Value

A GRanges containing stretches with number of clusters and average pairwise correlations calculated. The revmap can be used to retrieve the original clusters (see example below.)

See Also

Other Spatial functions: `findLinks()`, `trackLinks()`

Examples

```
# Calculate TPM values for bidirectional clusters
data(exampleBidirectional)
BCs <- calcTPM(exampleBidirectional)

# Find stretches
pearson_stretches <- findStretches(BCs, inputAssay="TPM")

# Use Kendall instead of pearson and require bigger stretches
kendall_stretches <- findStretches(BCs, inputAssay="TPM",
```

```
                                          minSize=5, method="kendall")

# Use the revmap to get stretches as a GRangesList
grl <- extractList(rowRanges(BCs), kendall_stretches$revmap)
names(grl) <- names(kendall_stretches)
```

`quantifyClusters` *Quantify expression of clusters (TSSs or enhancers) by summing CTSSs within clusters.*

Description

Quantify expression of clusters (TSSs or enhancers) by summing CTSSs within clusters.

Usage

```
quantifyClusters(object, clusters, inputAssay = "counts", sparse = FALSE)
```

Arguments

`object`	RangedSummarizedExperiment: CTSSs.
`clusters`	GRanges: Clusters to be quantified.
`inputAssay`	character: Name of assay holding expression values to be quantified (usually counts).
`sparse`	logical: If the input is a sparse matrix, TRUE will keep the output matrix sparse while FALSE will coerce it into a normal matrix.

Value

RangedSummarizedExperiment with row corresponding to clusters. seqinfo and colData is copied over from object.

See Also

Other Quantification functions: `quantifyCTSSs2()`, `quantifyCTSSs()`, `quantifyGenes()`

Examples

```
# CTSSs stored in a RangedSummarizedExperiment:
data(exampleCTSS)

# Clusters to be quantified as a GRanges:
data(exampleUnidirectional)
clusters <- rowRanges(exampleUnidirectional)

# Quantify clusters:
quantifyClusters(exampleCTSSs, clusters)
```

```
# For exceptionally large datasets,
# the resulting count matrix can be left sparse:
quantifyClusters(exampleCTSSs, rowRanges(exampleUnidirectional), sparse=TRUE)
```

`quantifyCTSSs` *Quantify CAGE Transcriptions Start Sites (CTSSs)*

Description

This function reads in CTSS count data from a series of BigWig-files (or bedGraph-files) and returns a CTSS-by-library count matrix. For efficient processing, the count matrix is stored as a sparse matrix (dgCMatrix from the Matrix package), and CTSSs are compressed to a GPos object if possible.

Usage

```
quantifyCTSSs(plusStrand, minusStrand, design = NULL, genome = NULL, ...)

## S4 method for signature 'BigWigFileList,BigWigFileList'
quantifyCTSSs(
  plusStrand,
  minusStrand,
  design = NULL,
  genome = NULL,
  nTiles = 1L
)

## S4 method for signature 'character,character'
quantifyCTSSs(plusStrand, minusStrand, design = NULL, genome = NULL)
```

Arguments

`plusStrand`	BigWigFileList or character: BigWig/bedGraph files with plus-strand CTSS data.
`minusStrand`	BigWigFileList or character: BigWig/bedGraph files with minus-strand CTSS data.
`design`	DataFrame or data.frame: Additional information on samples which will be added to the ouput
`genome`	Seqinfo: Genome information. If NULL the smallest common genome will be found using bwCommonGenome when BigWig-files are analyzed.
`...`	additional arguments passed to methods.
`nTiles`	integer: Number of genomic tiles to parallelize over.

Value

RangedSummarizedExperiment, where assay is a sparse matrix (dgCMatrix) of CTSS counts and design stored in colData.

See Also

Other Quantification functions: `quantifyCTSSs2()`, `quantifyClusters()`, `quantifyGenes()`

Examples

```
## Not run:
# Load the example data
data('exampleDesign')
# Use the BigWig-files included with the package:
bw_plus <- system.file('extdata', exampleDesign$BigWigPlus,
                       package = 'CAGEfightR')
bw_minus <- system.file('extdata', exampleDesign$BigWigMinus,
                        package = 'CAGEfightR')

# Create two named BigWigFileList-objects:
bw_plus <- BigWigFileList(bw_plus)
bw_minus <- BigWigFileList(bw_minus)
names(bw_plus) <- exampleDesign$Name
names(bw_minus) <- exampleDesign$Name

# Quantify CTSSs, by default this will use the smallest common genome:
CTSSs <- quantifyCTSSs(plusStrand=bw_plus,
                       minusStrand=bw_minus,
                       design=exampleDesign)

# Alternatively, a genome can be specified:
si <- seqinfo(bw_plus[[1]])
si <- si['chr18']
CTSSs_subset <- quantifyCTSSs(plusStrand=bw_plus,
                       minusStrand=bw_minus,
                       design=exampleDesign,
                       genome=si)

# Quantification can be speed up by using multiple cores:
library(BiocParallel)
register(MulticoreParam(workers=3))
CTSSs_subset <- quantifyCTSSs(plusStrand=bw_plus,
                       minusStrand=bw_minus,
                       design=exampleDesign,
                       genome=si)

# CAGEfightR also support bedGraph files, first BigWig is converted
bg_plus <- replicate(n=length(bw_plus), tempfile(fileext="_plus.bedGraph"))
bg_minus <- replicate(n=length(bw_minus), tempfile(fileext="_minus.bedGraph"))
names(bg_plus) <- names(bw_plus)
names(bg_minus) <- names(bw_minus)

convertBigWig2BedGraph(input=sapply(bw_plus, resource), output=bg_plus)
convertBigWig2BedGraph(input=sapply(bw_minus, resource), output=bg_minus)

# Then analyze: Note a genome MUST be supplied here!
si <- bwCommonGenome(bw_plus, bw_minus)
```

```
CTSSs_via_bg <- quantifyCTSSs(plusStrand=bg_plus,
                        minusStrand=bg_minus,
                        design=exampleDesign,
                        genome=si)

# Confirm that the two approaches yield the same results
all(assay(CTSSs_via_bg) == assay(CTSSs))

## End(Not run)
```

`quantifyCTSSs2` *Quantify CAGE Transcriptions Start Sites (CTSSs)*

Description

This function reads in CTSS count data from a series of BigWig-files and returns a CTSS-by-library count matrix. For efficient processing, the count matrix is stored as a sparse matrix (dgCMatrix).

Usage

```
quantifyCTSSs2(
  plusStrand,
  minusStrand,
  design = NULL,
  genome = NULL,
  tileWidth = 100000000L
)
```

Arguments

`plusStrand`	BigWigFileList: BigWig files with plus-strand CTSS data.
`minusStrand`	BigWigFileList: BigWig files with minus-strand CTSS data.
`design`	DataFrame or data.frame: Additional information on samples.
`genome`	Seqinfo: Genome information. If NULL the smallest common genome will be found using bwCommonGenome.
`tileWidth`	integer: Size of tiles to parallelize over.

Value

RangedSummarizedExperiment, where assay is a sparse matrix (dgCMatrix) of CTSS counts..

See Also

Other Quantification functions: `quantifyCTSSs()`, `quantifyClusters()`, `quantifyGenes()`

Examples

```
## Not run:
# Load the example data
data('exampleDesign')
# Use the BigWig-files included with the package:
bw_plus <- system.file('extdata', exampleDesign$BigWigPlus,
                       package = 'CAGEfightR')
bw_minus <- system.file('extdata', exampleDesign$BigWigMinus,
                        package = 'CAGEfightR')

# Create two named BigWigFileList-objects:
bw_plus <- BigWigFileList(bw_plus)
bw_minus <- BigWigFileList(bw_minus)
names(bw_plus) <- exampleDesign$Name
names(bw_minus) <- exampleDesign$Name

# Quantify CTSSs, by default this will use the smallest common genome:
CTSSs <- quantifyCTSSs(plusStrand=bw_plus,
                       minusStrand=bw_minus,
                       design=exampleDesign)

# Alternatively, a genome can be specified:
si <- seqinfo(bw_plus[[1]])
si <- si['chr18']
CTSSs <- quantifyCTSSs(plusStrand=bw_plus,
                       minusStrand=bw_minus,
                       design=exampleDesign,
                       genome=si)

# Quantification can be speed up by using multiple cores:
library(BiocParallel)
register(MulticoreParam(workers=3))
CTSSs <- quantifyCTSSs(plusStrand=bw_plus,
                       minusStrand=bw_minus,
                       design=exampleDesign,
                       genome=si)

## End(Not run)
```

`quantifyGenes` *Quantify expression of genes*

Description

Obtain gene-level expression estimates by summing clusters annotated to the same gene. Unannotated transcripts (NAs) are discarded.

Usage

```
quantifyGenes(object, genes, inputAssay = "counts", sparse = FALSE)
```

Arguments

`object`	RangedSummarizedExperiment: Cluster-level expression values.
`genes`	character: Name of column in rowData holding gene IDs (NAs will be discarded).
`inputAssay`	character: Name of assay holding values to be quantified, (usually counts).
`sparse`	logical: If the input is a sparse matrix, TRUE will keep the output matrix sparse while FALSE will coerce it into a normal matrix.

Value

RangedSummarizedExperiment with rows corresponding to genes. Location of clusters within genes is stored as a GRangesList in rowRanges. seqinfo and colData is copied over from object.

See Also

Other Quantification functions: `quantifyCTSSs2()`, `quantifyCTSSs()`, `quantifyClusters()`

Examples

```
data(exampleUnidirectional)

# Annotate clusters with geneIDs:
library(TxDb.Mmusculus.UCSC.mm9.knownGene)
txdb <- TxDb.Mmusculus.UCSC.mm9.knownGene
exampleUnidirectional <- assignGeneID(exampleUnidirectional,
                                      geneModels=txdb,
                                      outputColumn='geneID')

# Quantify counts within genes:
quantifyGenes(exampleUnidirectional, genes='geneID', inputAssay='counts')

# For exceptionally large datasets,
# the resulting count matrix can be left sparse:
quantifyGenes(exampleUnidirectional,
              genes='geneID',
              inputAssay='counts',
              sparse=TRUE)
```

`quickEnhancers` *Identify and quantify enhancers.*

Description

A convienient wrapper around clusterBidirectionally, subsetByBidirectionality and quantifyClusters.

Usage

```
quickEnhancers(object)
```

Arguments

`object`	RangedSummarizedExperiment: Location and counts of CTSSs, usually found by calling quantifyCTSSs.

Value

RangedSummarizedExperiment containing location and counts of enhancers.

See Also

Other Wrapper functions: `quickGenes()`, `quickTSSs()`

Examples

```
# See the CAGEfightR vignette for an overview!
```

`quickGenes` *Identify and quantify genes.*

Description

A convienient wrapper around assignGeneID, and quantifyGenes. Also removes unstranded features

Usage

```
quickGenes(object, geneModels = NULL, ...)
```

Arguments

`object`	RangedSummarizedExperiment: Location and counts of clusters, usually found by calling quantifyClusters.
`geneModels`	TxDb or GRanges: Gene models via a TxDb, or manually specified as a GRangesList.
`...`	additional arguments passed to assignGeneID.

Value

RangedSummarizedExperiment containing gene expression and clusters assigned within each gene.

See Also

Other Wrapper functions: `quickEnhancers()`, `quickTSSs()`

Examples

```
# See the CAGEfightR vignette for an overview!
```

`quickTSSs` *Identify and quantify Transcription Start Sites (TSSs).*

Description

A convienient wrapper around calcTPM, calcPooled, tuneTagClustering, clusterUnidirectionally and quantifyClusters.

Usage

```
quickTSSs(object)
```

Arguments

`object`	RangedSummarizedExperiment: Location and counts of CTSSs, usually found by calling quantifyCTSSs.

Value

RangedSummarizedExperiment containing location and counts of TSSs

See Also

Other Wrapper functions: `quickEnhancers()`, `quickGenes()`

Examples

```
# See the CAGEfightR vignette for an overview!
```

`shapeEntropy` *Shape statistic: Shannon Entropy*

Description

Calculates the Shannon Entropy (base log2) for a vector. Zeros are removed before calculation.

Usage

```
shapeEntropy(x)
```

Arguments

`x`	numeric Rle vector: Coverage series.

Value

Numeric.

See Also

Other Shape functions: calcShape(), shapeIQR(), shapeMean()

Examples

```
# Hypothetical shard/broad clusters:
x_sharp <- Rle(c(1,1,1,4,5,2,1,1))
x_broad <- Rle(c(1,2,3,5,4,3,2,1))

# Calculate Entropy
shapeEntropy(x_sharp)
shapeEntropy(x_broad)

# See calcShape for more usage examples
```

---

shapeIQR *Shape statitic: Interquartile range*

Description

Calculates the interquartile range of a vector.

Usage

```
shapeIQR(x, lower = 0.25, upper = 0.75)
```

Arguments

x	numeric Rle vector: Coverage series.
lower	numeric: Lower quartile.
upper	numeric: Upper quartile.

Value

Numeric

See Also

Other Shape functions: calcShape(), shapeEntropy(), shapeMean()

Examples

```
# Hypothetical shard/broad clusters:
x_sharp <- Rle(c(1,1,1,4,5,2,1,1))
x_broad <- Rle(c(1,2,3,5,4,3,2,1))

# Calculate IQR
shapeIQR(x_sharp)
shapeIQR(x_broad)

# See calcShape for more usage examples
```

---

shapeMean *Shape statistic: Mean*

Description

Calculates the mean of a vector.

Usage

```
shapeMean(x)
```

Arguments

x	numeric Rle vector: Coverage series.

Value

Numeric

See Also

Other Shape functions: `calcShape()`, `shapeEntropy()`, `shapeIQR()`

Examples

```
# Hypothetical shard/broad clusters:
x_sharp <- Rle(c(1,1,1,4,5,2,1,1))
x_broad <- Rle(c(1,2,3,5,4,3,2,1))

# Calculate mean
shapeMean(x_sharp)
shapeMean(x_broad)

# See calcShape for more usage examples
```

`shapeMultimodality` *Shape statistic: Multimodality*

Description

Shape statistic: Multimodality

Usage

```
shapeMultimodality(x)
```

Arguments

`x`	numeric Rle vector: Coverage series.

Value

Numeric.

Examples

```
# See calcShape for usage examples
```

`subsetByBidirectionality` *Subset by sample-wise bidirectionality of clusters.*

Description

A convenient wrapper around calcBidirectionality and subset.

Usage

```
subsetByBidirectionality(object, ...)

## S4 method for signature 'GRanges'
subsetByBidirectionality(
  object,
  samples,
  inputAssay = "counts",
  outputColumn = "bidirectionality",
  minSamples = 0
)

## S4 method for signature 'GPos'
subsetByBidirectionality(object, ...)
```

```
## S4 method for signature 'RangedSummarizedExperiment'
subsetByBidirectionality(object, ...)
```

Arguments

`object`	GRanges or RangedSummarizedExperiment: Unstranded clusters with peaks stored in the 'thick' column.
`...`	additional arguments passed to methods.
`samples`	RangedSummarizedExperiment: Sample-wise CTSSs stored as an assay.
`inputAssay`	character: Name of assay in samples holding input CTSS values.
`outputColumn`	character: Name of column in object to hold bidirectionality values.
`minSamples`	integer: Only regions with bidirectionality above this value are retained.

Value

object with bidirectionality values added as a column, and low bidirectionaly regions removed.

See Also

Other Subsetting functions: `subsetByComposition()`, `subsetBySupport()`

Other Calculation functions: `calcBidirectionality()`, `calcComposition()`, `calcPooled()`, `calcShape()`, `calcSupport()`, `calcTPM()`, `calcTotalTags()`, `subsetByComposition()`, `subsetBySupport()`

Examples

```
data(exampleCTSSs)
data(exampleBidirectional)

# Keep only clusters that are bidirectional in at least one sample:
subsetByBidirectionality(exampleBidirectional, samples=exampleCTSSs)
```

`subsetByComposition` *Subset by composition across samples*

Description

A convenient wrapper around calcComposition and subset.

Usage

```
subsetByComposition(
  object,
  inputAssay = "counts",
  outputColumn = "composition",
  unexpressed = 0.1,
  genes = "geneID",
  minSamples = 1
)
```

Arguments

`object`	RangedSummarizedExperiment: CAGE data quantified at CTSS, cluster or gene-level.
`inputAssay`	character: Name of assay holding input expression values.
`outputColumn`	character: Name of column in rowRanges to hold composition values.
`unexpressed`	numeric: Composition will be calculated based on features larger than this cut-off.
`genes`	character: Name of column in rowData holding genes (NAs are not allowed.)
`minSamples`	numeric: Only features with composition in more than this number of samples will be kept.

Value

RangedSummarizedExperiment with composition values added as a column in rowData and features with less composition than minSamples removed.

See Also

Other Subsetting functions: `subsetByBidirectionality()`, `subsetBySupport()`

Other Calculation functions: `calcBidirectionality()`, `calcComposition()`, `calcPooled()`, `calcShape()`, `calcSupport()`, `calcTPM()`, `calcTotalTags()`, `subsetByBidirectionality()`, `subsetBySupport()`

Examples

```
data(exampleUnidirectional)

# Annotate clusters with geneIDs:
library(TxDb.Mmusculus.UCSC.mm9.knownGene)
txdb <- TxDb.Mmusculus.UCSC.mm9.knownGene

exampleUnidirectional <- assignGeneID(exampleUnidirectional,
                                      geneModels=txdb,
                                      outputColumn='geneID')
exampleUnidirectional <- subset(exampleUnidirectional, !is.na(geneID))

# Keep only clusters more than 10% in more than one sample:
calcComposition(exampleUnidirectional)

# Keep only clusters more than 5% in more than 2 samples:
subsetByComposition(exampleUnidirectional, unexpressed = 0.05, minSamples=2)
```

`subsetBySupport` *Subset by support across samples*

Description

A convienient wrapper around calcSupport and subset.

Usage

```
subsetBySupport(
  object,
  inputAssay = "counts",
  outputColumn = "support",
  unexpressed = 0,
  minSamples = 1
)
```

Arguments

`object`	RangedSummarizedExperiment: CAGE data quantified at CTSS, cluster or gene-level.
`inputAssay`	character: Name of assay holding input expression values.
`outputColumn`	character: Name of column in rowRanges to hold support values.
`unexpressed`	numeric: Support will be calculated based on features larger than this cutoff.
`minSamples`	numeric: Only features with support in more than this number of samples will be kept.

Value

RangedSummarizedExperiment with support added as a column in rowRanges and features with less support than minSamples removed.

See Also

Other Subsetting functions: `subsetByBidirectionality()`, `subsetByComposition()`

Other Calculation functions: `calcBidirectionality()`, `calcComposition()`, `calcPooled()`, `calcShape()`, `calcSupport()`, `calcTPM()`, `calcTotalTags()`, `subsetByBidirectionality()`, `subsetByComposition()`

Examples

```
data(exampleBidirectional)

# Keep clusters with at least one tag in two samples
subsetBySupport(exampleBidirectional)

# Keep clusters with at least two tags in four samples
subsetBySupport(exampleBidirectional, unexpressed=1, minSamples=2)
```

swapRanges *Swap ranges in a GRanges.*

Description

Swap out the range of a GRanges-object with another IRanges-object stored inside the same object. I.e., swapping cluster widths with cluster peaks.

Usage

```
swapRanges(object, ...)

## S4 method for signature 'GenomicRanges'
swapRanges(object, inputColumn = "thick", outputColumn = NULL)

## S4 method for signature 'RangedSummarizedExperiment'
swapRanges(object, ...)
```

Arguments

object	GRanges or RangedSummarizedExperiment: Primary ranges to be swapped out.
...	additional arguments passed to methods.
inputColumn	character: Name of column holding IRanges to be swapped in.
outputColumn	character or NULL: Name of column to hold swapped out ranges, if NULL original ranges are not saved.

Value

GRanges with inputColumn swapped in as ranges.

See Also

Other Swapping functions: swapScores()

Examples

```
data(exampleUnidirectional)
gr <- rowRanges(exampleUnidirectional)

# Swap in peaks as main ranges
peaks <- swapRanges(gr)
head(width(gr))
head(width(peaks))

# swapRanges() can also be directly called on a RangedSummarizedExperiment:
swapRanges(exampleUnidirectional)

# The original can optionally be saved in the output object
swapRanges(gr, outputColumn = 'swapped')
```

swapScores *Swap scores in SummarizedExperiment*

Description

Take scores for a specific sample and a specific assay and put them into rowData.

Usage

```
swapScores(object, outputColumn = "score", inputAssay, sample)
```

Arguments

`object`	SummarizedExperiment: CAGE-data
`outputColumn`	character: Column in rowData to to hold swapped in scores.
`inputAssay`	character: Name of assay to take scores from.
`sample`	character: Name of sample to take scores from.

Value

SummarizedExperiment with sample scores from inputAssay in rowRata.

See Also

Other Swapping functions: `swapRanges()`

Examples

```
data(exampleCTSSs)
sample_names <- colnames(exampleCTSSs)

# Replace scores with values from the first sample:
x <- swapScores(exampleCTSSs, inputAssay='counts', sample=sample_names[1])
rowRanges(x)
```

trackBalance *Create Genome Browser Track of bidirectional balance scores*

Description

Visualize balance scores used for detectiong of bidirectional sites. Mainly intended as diagnostic tools for expert user.

Usage

```
trackBalance(object, ...)

## S4 method for signature 'GRanges'
trackBalance(
  object,
  window = 199,
  plusColor = "cornflowerblue",
  minusColor = "tomato",
  balanceColor = "forestgreen",
  ...
)

## S4 method for signature 'GPos'
trackBalance(object, ...)

## S4 method for signature 'RangedSummarizedExperiment'
trackBalance(object, ...)
```

Arguments

`object`	GenomicRanges or RangedSummarizedExperiment: Ranges with CTSSs in the score column.
`...`	additional arguments passed to DataTrack.
`window`	integer: Width of sliding window used for calculating windowed sums.
`plusColor`	character: Color for plus-strand coverage.
`minusColor`	character: Color for minus-strand coverage.
`balanceColor`	character: Color for bidirectional balance.

Value

list of 3 DataTracks for upstream, downstream and balance.

Note

Potentially consumes a large amount of memory!

See Also

Other Genome Browser functions: `trackCTSS()`, `trackClusters()`, `trackLinks()`

Examples

```
## Not run:
library(Gviz)
data(exampleCTSSs)
data(exampleBidirectional)
```

```
# Calculate pooled CTSSs
exampleCTSSs <- calcTPM(exampleCTSSs, totalTags='totalTags')
exampleCTSSs <- calcPooled(exampleCTSSs)

# Find a bidirectional cluster to plot:
BC <- rowRanges(exampleBidirectional[10,])
start(BC) <- start(BC) - 250
end(BC) <- end(BC) + 250
subsetOfCTSSs <- subsetByOverlaps(exampleCTSSs, BC)

# Build balance track
balance_track <- trackBalance(subsetOfCTSSs)

# Plot
plotTracks(balance_track, from=start(BC), to=end(BC),
           chromosome=seqnames(BC))

## End(Not run)
```

`trackClusters` *Create genome browser track of clusters.*

Description

Create a Gviz-track of clusters (unidirectional TCs or bidirectional enhancers), where cluster strand and peak is indicated.

Usage

```
trackClusters(object, ...)

## S4 method for signature 'GRanges'
trackClusters(
  object,
  plusColor = "cornflowerblue",
  minusColor = "tomato",
  unstrandedColor = "hotpink",
  ...
)

## S4 method for signature 'RangedSummarizedExperiment'
trackClusters(object, ...)
```

Arguments

`object`	GRanges: GRanges with peaks in the thick-column.
`...`	additional arguments passed on to GeneRegionTrack.
`plusColor`	character: Color for plus-strand features.

`minusColor`	character: Color for minus-strand features.
`unstrandedColor`	character: Color for unstranded features.

Value

GeneRegionTrack-object.

See Also

Other Genome Browser functions: `trackBalance()`, `trackCTSS()`, `trackLinks()`

Examples

```
library(Gviz)
data(exampleUnidirectional)

# Find some wide unidirectional clusters:
TCs <- subset(exampleUnidirectional, width >= 100)

# Create track
clusters_track <- trackClusters(TCs[1:2,], name='Tag clusters', col=NULL)

# Plot
plotTracks(clusters_track)

# See vignette for examples on how to combine multiple Gviz tracks
```

`trackCTSS` *Create Genome Browser track of CTSSs.*

Description

Create a Gviz-track of CTSSs, where Plus/minus strand signal is shown positive/negative. This representation makes it easy to identify bidirectional peaks.

Usage

```
trackCTSS(object, ...)

## S4 method for signature 'GRanges'
trackCTSS(object, plusColor = "cornflowerblue", minusColor = "tomato", ...)

## S4 method for signature 'RangedSummarizedExperiment'
trackCTSS(object, ...)

## S4 method for signature 'GPos'
trackCTSS(object, ...)
```

Arguments

`object`	GenomicRanges or RangedSummarizedExperiment: Ranges with CTSSs in the score column.
`...`	additional arguments passed on to DataTrack.
`plusColor`	character: Color for plus-strand coverage.
`minusColor`	character: Color for minus-strand coverage.

Value

DataTrack-object.

See Also

Other Genome Browser functions: `trackBalance()`, `trackClusters()`, `trackLinks()`

Examples

```
library(Gviz)
data(exampleCTSSs)
data(exampleUnidirectional)
data(exampleBidirectional)

# Example uni- and bidirectional clusters
TC <- rowRanges(subset(exampleUnidirectional, width>=100)[3,])
BC <- rowRanges(exampleBidirectional[3,])

# Create pooled track
subsetOfCTSSs <- subsetByOverlaps(rowRanges(exampleCTSSs), c(BC, TC, ignore.mcols=TRUE))
pooledTrack <- trackCTSS(subsetOfCTSSs)

# Plot
plotTracks(pooledTrack, from=start(TC)-100, to=end(TC)+100,
           chromosome=seqnames(TC), name='TC')
plotTracks(pooledTrack, from=start(BC)-100, to=end(BC)+100,
           chromosome=seqnames(BC), name='BC')

# See vignette for examples on how to combine multiple Gviz tracks
```

`trackLinks` *Create a genome browser track of links.*

Description

Create a Gviz-track of links (e.g. between TSSs and enhancers), where arches connect the different pairs of clusters. The height of arches can be set to scale the strength of the interaction (for example indicating higher correlation). This function is a thin wrapper around the InteractionTrack-class from the GenomicInteractions package. Currently, only scaling arch height by p-value is supported.

Usage

```
trackLinks(object, ...)
```

Arguments

`object`	GInteractions: Links or pairs between clusters.
`...`	additional arguments passed to InteractionTrack via displayPars.

Value

InteractionTrack-object from the GenomicInteractions package.

See Also

Other Genome Browser functions: `trackBalance()`, `trackCTSS()`, `trackClusters()`

Other Spatial functions: `findLinks()`, `findStretches()`

Examples

```
library(InteractionSet)
library(Gviz)
library(GenomicInteractions)

# Links between highly expressed unidirectional clusters
TCs <- subset(exampleUnidirectional, score > 10)
TC_links <- findLinks(TCs, inputAssay="counts", maxDist=10000L)
link_track <- trackLinks(TC_links, name="TSS links", interaction.measure="p.value")

# Plot region
plot_region <- GRanges(seqnames="chr18",
                       ranges = IRanges(start=start(anchors(TC_links[1],
                                                            "first")),
                                        end=end(anchors(TC_links[1],
                                                        "second"))))
# Plot using Gviz
plotTracks(link_track,
           from=start(plot_region),
           to=end(plot_region),
           chromosome = as.character(seqnames(plot_region)))
# See vignette for examples on how to combine multiple Gviz tracks
```

`trimToPeak` *Trim width of TCs by distance from TC peak*

Description

Trim the width of TCs by distance from the TC peaks.

Usage

```
trimToPeak(object, pooled, ...)

## S4 method for signature 'GRanges,GRanges'
trimToPeak(object, pooled, upstream, downstream)

## S4 method for signature 'GRanges,GPos'
trimToPeak(object, pooled, ...)

## S4 method for signature 'RangedSummarizedExperiment,GenomicRanges'
trimToPeak(object, pooled, ...)

## S4 method for signature 'GRanges,RangedSummarizedExperiment'
trimToPeak(object, pooled, ...)

## S4 method for signature
## 'RangedSummarizedExperiment,RangedSummarizedExperiment'
trimToPeak(object, pooled, ...)
```

Arguments

`object`	GenomicRanges or RangedSummarizedExperiment: Tag clusters.
`pooled`	GenomicRanges or RangedSummarizedExperiment: Basepair-wise pooled CTSS (stored in the score column).
`...`	additional arguments passed to methods.
`upstream`	integer: Maximum upstream distance from TC peak.
`downstream`	integer: Maximum downstream distance from TC peak.

Value

data.frame with two columns: threshold and nTCs (number of Tag Clusters)

See Also

Other Clustering functions: `clusterBidirectionally()`, `clusterUnidirectionally()`, `trimToPercentiles()`, `tuneTagClustering()`

Other Trimming functions: `trimToPercentiles()`

Examples

```
data(exampleCTSSs)
data(exampleBidirectional)

# Calculate pooled CTSSs
exampleCTSSs <- calcTPM(exampleCTSSs, totalTags='totalTags')
exampleCTSSs <- calcPooled(exampleCTSSs)

# Choose a few wide clusters:
```

```
TCs <- subset(exampleUnidirectional, width >= 100)

# Trim to +/- 10 bp of TC peak
trimToPeak(TCs, pooled=exampleCTSSs, upstream=10, downstream=10)
```

`trimToPercentiles` *Trim width of TCs to expression percentiles*

Description

Given a set of TCs and genome-wide CTSS coverage, reduce the width of TC until a certain amount of expression has been removed.

Usage

```
trimToPercentiles(object, pooled, ...)

## S4 method for signature 'GRanges,GRanges'
trimToPercentiles(object, pooled, percentile = 0.1, symmetric = FALSE)

## S4 method for signature 'GRanges,GPos'
trimToPercentiles(object, pooled, ...)

## S4 method for signature 'RangedSummarizedExperiment,GenomicRanges'
trimToPercentiles(object, pooled, ...)

## S4 method for signature 'GRanges,RangedSummarizedExperiment'
trimToPercentiles(object, pooled, ...)

## S4 method for signature
## 'RangedSummarizedExperiment,RangedSummarizedExperiment'
trimToPercentiles(object, pooled, ...)
```

Arguments

`object`	GenomicRanges or RangedSummarizedExperiment: TCs to be trimmed.
`pooled`	GenomicRanges or RangedSummarizedExperiment: CTSS coverage.
`...`	additional arguments passed to methods.
`percentile`	numeric: Fraction of expression to remove from TCs.
`symmetric`	logical: Whether to trim the same amount from both edges of the TC (TRUE) or always trim from the least expressed end (FALSE).

Value

GRanges with trimmed TCs, including recalculated peaks and scores.

See Also

Other Clustering functions: `clusterBidirectionally()`, `clusterUnidirectionally()`, `trimToPeak()`, `tuneTagClustering()`

Other Trimming functions: `trimToPeak()`

Examples

```
data(exampleCTSSs)
data(exampleBidirectional)

# Calculate pooled CTSSs
exampleCTSSs <- calcTPM(exampleCTSSs, totalTags='totalTags')
exampleCTSSs <- calcPooled(exampleCTSSs)

# Choose a few wide clusters:
TCs <- subset(exampleUnidirectional, width >= 100)

# Symmetric trimming (same percentage from each side):
TCs_sym <- trimToPercentiles(TCs, pooled=exampleCTSSs, symmetric=FALSE)

# Assymmetric trimming (always trim from lowest side):
TCs_asym <- trimToPercentiles(TCs, pooled=exampleCTSSs, symmetric=TRUE)

# Compare the two results sets of widths:
summary(width(TCs_sym) - width(TCs_asym))
```

---

tuneTagClustering — *Determine the optimal pooled threshold for unidirectional tag clustering.*

Description

This function counts the number of Tag Clusters (TCs) for an series of small incremental pooled cutoffs

Usage

```
tuneTagClustering(object, ...)

## S4 method for signature 'GRanges'
tuneTagClustering(
  object,
  steps = 10L,
  mergeDist = 20L,
  searchMethod = "minUnique",
  maxExponent = 1
)
```

```
## S4 method for signature 'RangedSummarizedExperiment'
tuneTagClustering(object, ...)

## S4 method for signature 'GPos'
tuneTagClustering(object, ...)
```

Arguments

object	GenomicRanges or RangedSummarizedExperiment: Pooled CTSS.
...	additional arguments passed to methods.
steps	integer: Number of thresholds to analyze (in addition to treshold=0).
mergeDist	integer: Merge TCs within this distance.
searchMethod	character: For advanced user only, see details.
maxExponent	numeric: The maximal threshold to analyse is obtained as min(score)*2^maxExponent (only used if searchMethod='exponential').

Value

data.frame with two columns: threshold and nTCs (number of Tag Clusters)

See Also

Other Clustering functions: clusterBidirectionally(), clusterUnidirectionally(), trimToPeak(), trimToPercentiles()

Examples

```
## Not run:
data(exampleCTSSs)

# Calculate pooledTPM, using supplied number of total tags
exampleCTSSs <- calcTPM(exampleCTSSs,
                        inputAssay='counts',
                        outputAssay='TPM',
                        totalTags='totalTags')
exampleCTSSs <- calcPooled(exampleCTSSs, inputAssay='TPM')

# Set backend
library(BiocParallel)
register(SerialParam())

# Find optimal slice-threshold for reduce distance of 20:
tuneTagClustering(object=exampleCTSSs)

## End(Not run)
```

`utilsAggregateRows` *Utility: Aggregate rows*

Description

Used by quantifyClusters and quantifyGenes. Wrapper around rowsum with a few improvements: 1) Handles dgCMatrix 2) Suppresses warnings from and discards NAs in grouping 3) Checks if output can be coerced to integer (useful when aggregating a dgCMatrix), 4) For the dgCMatrix case, has the option to keep unused levels and output a sparse matrix.

Usage

```
utilsAggregateRows(x, group, drop = TRUE, sparse = FALSE)

## S4 method for signature 'matrix'
utilsAggregateRows(x, group, drop = TRUE, sparse = FALSE)

## S4 method for signature 'dgCMatrix'
utilsAggregateRows(x, group, drop = TRUE, sparse = FALSE)
```

Arguments

`x`	matrix or dgCMatrix: Matrix to be aggregated.
`group`	factor: Grouping, can cannot NAs which will be discarded.
`drop`	logical: Whether to drop unused levels (TRUE) or keep assign them 0 (FALSE).
`sparse`	logical: Whether output should be coerced to a dense matrix.

Value

matrix (or dgCMatrix if sparse=TRUE)

See Also

Other Utility functions: `utilsDeStrand()`, `utilsScoreOverlaps()`, `utilsSimplifyTxDb()`

Examples

```
library(Matrix)
data("exampleCTSSs")
data("exampleUnidirectional")

# Sparse and dense examples
sparse_matrix <- assay(exampleCTSSs)
dense_matrix <- as(sparse_matrix, "matrix")

# Groupings
grp <- findOverlaps(query = exampleCTSSs,
```

```
                  subject = exampleUnidirectional,
                  select="arbitrary")

# Aggregate rows and compare
sparse_res <- utilsAggregateRows(sparse_matrix, grp)
dense_res <- utilsAggregateRows(dense_matrix, grp)
all(sparse_res == dense_res)

# Note that storage type was converted to integers!
storage.mode(sparse_res)
storage.mode(dense_res)

# You can also elect to keep a sparse representation
utilsAggregateRows(sparse_matrix, grp, sparse = TRUE)

#### Examples with unused levels ####

# Silly example
dense_mat <- replicate(5, runif(10))
sparse_mat <- as(dense_mat, "dgCMatrix")
fct_unused <- factor(c(1, 1, NA, NA, 3, 3, NA, NA, 5, 5), levels=1:5)

# The default is to drop unused levels
utilsAggregateRows(dense_mat, fct_unused, drop=TRUE)
utilsAggregateRows(sparse_mat, fct_unused, drop=TRUE)

# For dgCMatrix, one can elect to retain these:
utilsAggregateRows(sparse_mat, fct_unused, drop=FALSE)

# For matrix, a warning is produced if either drop or sparse is requested
utilsAggregateRows(dense_mat, fct_unused, drop=FALSE)
utilsAggregateRows(dense_mat, fct_unused, sparse=TRUE)
```

---

utilsDeStrand — *Utility: Split Genomic Ranges by strand*

Description

Utility function that attemps to split genomic ranges by strand with split(object, strand(object))

Usage

```
utilsDeStrand(object)
```

Arguments

`object`	Any object with a split and strand method, e.g. GRanges/GPos

Value

Object split by strand, e.g. GRangesList.

See Also

Other Utility functions: utilsAggregateRows(), utilsScoreOverlaps(), utilsSimplifyTxDb()

Examples

```
gp <- GPos(seqnames=Rle(c("chr1", "chr2", "chr1"), c(10, 6, 4)),
            pos=c(44:53, 5:10, 2:5),
            strand=c(rep("+", 10), rep("-", 10)))
gr <- as(gp, "GRanges")
utilsDeStrand(gp)
utilsDeStrand(gr)
```

---

utilsScoreOverlaps *Utility: Counting overlaps taking into account scores*

Description

Similar to countOverlaps, but takes the score column into account.

Usage

```
utilsScoreOverlaps(query, subject, ...)
```

Arguments

query	same as findOverlaps/countOverlaps
subject	same as findOverlaps/countOverlaps
...	additional arguments passed to findOverlaps

Value

vector of number of overlaps weigthed by score column.

See Also

https://support.bioconductor.org/p/87736/#87758

Other Utility functions: utilsAggregateRows(), utilsDeStrand(), utilsSimplifyTxDb()

Examples

```
gr1 <- GRanges(seqnames="chr1",
               ranges=IRanges(start = c(4, 9, 10, 30),
                              end = c(4, 15, 20, 31)),
               strand="+")
gr2 <- GRanges(seqnames="chr1",
               ranges=IRanges(start = c(1, 4, 15, 25),
                              end = c(2, 4, 20, 26)),
               strand=c("+"),
```

```
                 score=c(10, 20, 15, 5))
countOverlaps(gr1, gr2)
utilsScoreOverlaps(gr1, gr2)
```

`utilsSimplifyTxDb` *Utility: Extract annotation hierachy from a TxDb.*

Description

Used by assignTxType. This function extracts the hierachical annotations used by assignTxType from a TxDb object. If you are annotating many ranges, it can be time saving to built the hierachy first, to avoid processing the TxDb for every assignTxDb call.

Usage

```
utilsSimplifyTxDb(
  object,
  tssUpstream = 100,
  tssDownstream = 100,
  proximalUpstream = 1000,
  detailedAntisense = FALSE
)
```

Arguments

`object` TxDb: Transcript database

`tssUpstream` integer: Distance to extend annotated promoter upstream.

`tssDownstream` integer: Distance to extend annotated promoter downstream.

`proximalUpstream`
integer: Maximum distance upstream of promoter to be considered proximal.

`detailedAntisense`
logical: Wether to mirror all txType categories in the antisense direction (TRUE) or lump them all together (FALSE).

Value

GRangesList of annotation hierachy

See Also

assignTxType

Other Utility functions: `utilsAggregateRows()`, `utilsDeStrand()`, `utilsScoreOverlaps()`

Examples

```
## Not run:
data(exampleUnidirectional)

# Obtain transcript models from a TxDb-object:
library(TxDb.Mmusculus.UCSC.mm9.knownGene)
txdb <- TxDb.Mmusculus.UCSC.mm9.knownGene

# Simplify txdb
hierachy <- utilsSimplifyTxDb(txdb)

# Standard way of calling
x <- assignTxType(exampleUnidirectional,
                  txModels=txdb)

# Calling with premade hierachy
y <- assignTxType(exampleUnidirectional, txModels=hierachy)

# These are identical
stopifnot(all(rowRanges(x)$txType == rowRanges(y)$txType))

## End(Not run)
```

Index