Package 'BRGenomics'

January 1, 2020

Type Package

Title Tools for the Efficient Analysis of High-Resolution Genomics Data

Version 0.4.2

Description This package provides useful and efficient utilites for the analysis of high-resolution genomic data using standard Bioconductor methods and classes.

License Artistic-2.0

Encoding UTF-8

LazyData FALSE

RoxygenNote 7.0.2

Depends GenomicRanges,
GenomeInfoDb

Imports rtracklayer,
parallel,
IRanges,
stats,
Rsamtools,
GenomicAlignments,
DESeq2,
SummarizedExperiment,
utils

Suggests BiocStyle,
knitr,
rmarkdown,
TxDb.Hsapiens.UCSC.hg38.knownGene,
TxDb.Hsapiens.UCSC.hg19.knownGene,
TxDb.Mmusculus.UCSC.mm10.knownGene,
TxDb.Mmusculus.UCSC.mm9.knownGene,
TxDb.Dmelanogaster.UCSC.dm6.ensGene,
TxDb.Dmelanogaster.UCSC.dm3.ensGene,
testthat

biocViews Software,
DataImport,
Sequencing,
Coverage,
RNASeq,
ATACSeq,

ChIPSeq,
Transcription,
GeneRegulation,
GeneExpression,
Normalization

VignetteBuilder knitr

R topics documented:

---

binNdimensions   *N-dimensional binning*

Description

This function takes in data along 1 or more dimensions, and for each dimension the data is di-
vided into evenly-sized bins from the minimum value to the maximum value, and bin numbers are
returned. For instance, if each index of the input data were a gene, the input dimensions would
be various quantitative measures of that gene, e.g. expression level, number of exons, length, etc.
If plotted in cartesian coordinates, each gene would be a single datapoint, and each measurement
would be a separate dimension. The bin numbers for each datapoint in each dimension are returned
in a dataframe, with a column for each dimension and a row for each index.

Usage

```
binNdimensions(..., nbins = 10)
```

Arguments

`...`	A single dataframe, or any number of lists or vectors containing different measurements across the same datapoints. If a dataframe is given, columns should correspond to measurements (dimensions). If lists or vectors are given, they must all have the same lengths. Other input classes will be coerced into a single dataframe.
`nbins`	Either a number giving the number of bins to use for all dimensions (default = 10), or a vector containing the number of bins to use for each dimension of input data given.

Value

A dataframe containing indices in `1:nbins` for each datapoint in each dimension.

Author(s)

Mike DeBerardine

Examples

```
data("PROseq") # import included PROseq data
data("txs_dm6_chr4") # import included transcripts

#------------------------------------------------#
# find counts in promoter, early genebody, and near CPS
#------------------------------------------------#

pr <- promoters(txs_dm6_chr4, 0, 100)
early_gb <- genebodies(txs_dm6_chr4, 500, 1000, fix.end = "start")
cps <- genebodies(txs_dm6_chr4, -500, 500, fix.start = "end")

counts_pr <- getCountsByRegions(PROseq, pr)
counts_gb <- getCountsByRegions(PROseq, early_gb)
counts_cps <- getCountsByRegions(PROseq, cps)

#------------------------------------------------#
# divide genes into 20 bins for each measurement
#------------------------------------------------#

count_bins <- binNdimensions(counts_pr, counts_gb, counts_cps, nbins = 20)

length(txs_dm6_chr4)
nrow(count_bins)
count_bins[1:10, ]
```

---

bootstrap-signal-by-position

Bootstrapping Mean Signal by Position for Metaplotting

Description

These functions perform bootstrap subsampling of mean readcounts at different positions within regions of interest (metaSubsample), or, in the more general case of metaSubsampleMatrix, column means of a matrix are bootstrapped by sampling the rows. Mean signal counts can be calculated at base-pair resolution, or over larger bins.

Usage

```
metaSubsample(
  dataset.gr,
  regions.gr,
  binsize = 1,
  first.output.xval = 1,
  sample.name = deparse(substitute(dataset.gr)),
  n.iter = 1000,
  prop.sample = 0.1,
  lower = 0.125,
  upper = 0.875,
  NF = 1,
  field = "score",
  remove.empty = FALSE,
  ncores = detectCores()
)

metaSubsampleMatrix(
  counts.mat,
  binsize = 1,
  first.output.xval = 1,
  sample.name = deparse(substitute(counts.mat)),
  n.iter = 1000,
  prop.sample = 0.1,
  lower = 0.125,
  upper = 0.875,
  NF = 1,
  remove.empty = FALSE,
  ncores = detectCores()
)
```

Arguments

dataset.gr	A GRanges object in which signal is contained in metadata (typically in the "score" field).
regions.gr	A GRanges object containing intervals over which to metaplot. All ranges must have the same width.
binsize	The size of bin (in basepairs, or number of columns for metaSubsampleMatrix) to use for counting signal. Especially important for counting signal over large or sparse regions.
first.output.xval	
	The relative start position of the first bin, e.g. if regions.gr begins at 50 bases upstream of the TSS, set first.output.xval = -50. This number only affects the x-values that are returned, which are provided as a convenience.

sample.name	Defaults to the name of the input dataset. This is included in the output as a convenience, as it allows row-binding outputs from different samples.
n.iter	Number of random subsampling iterations to perform. Default is 1000.
prop.sample	The proportion of the ranges in regions.gr (e.g. the proportion of genes) or the proportion of rows in counts.mat to sample in each iteration. The default is 0.1 (10 percent).
lower, upper	The lower and upper quantiles of subsampled signal means to return. The defaults, 0.125 and 0.875 (i.e. the 12.5th and 85.5th percentiles) return a 75 percent confidence interval about the bootstrapped mean.
NF	Optional normalization factor by which to multiply the counts.
field	The metadata field of dataset.gr to be counted.
remove.empty	A logical indicating whether regions (metaSubsample) or rows (metaSubsampleMatrix) without signal should be removed from the analysis.
ncores	Number of cores to use for computations.
counts.mat	A matrix over which to bootstrap column means by subsampling its rows. Typically, a matrix of readcounts with rows for genes and columns for positions within those genes.

Value

Dataframe containing x-values, means, lower quantiles, upper quantiles, and the sample name (as a convenience for row-binding multiple of these dataframes).

Author(s)

Mike DeBerardine

See Also

[getCountsByPositions](#)

Examples

```
data("PROseq") # import included PROseq data
data("txs_dm6_chr4") # import included transcripts

# for each transcript, use promoter-proximal region from TSS to +100
pr <- promoters(txs_dm6_chr4, 0, 100)

#-------------------------------------------------#
# Bootstrap average signal in each 5 bp bin across all transcripts,
# and get confidence bands for middle 30% of bootstrapped means
#-------------------------------------------------#

set.seed(11)
df <- metaSubsample(PROseq, pr, binsize = 5,
                    lower = 0.35, upper = 0.65,
                    ncores = 2)
df[1:10, ]

#-------------------------------------------------#
# Plot bootstrapped means with confidence intervals
#-------------------------------------------------#
```

```
plot(mean ~ x, df, type = "l", main = "PROseq Signal",
     ylab = "Mean + 30% CI", xlab = "Distance from TSS")
polygon(c(df$x, rev(df$x)), c(df$lower, rev(df$upper)),
        col = adjustcolor("black", 0.1), border = FALSE)


#================================================#
# Using a matrix as input
#================================================#

# generate a matrix of counts in each region
countsmat <- getCountsByPositions(PROseq, pr)
dim(countsmat)

#------------------------------------------------#
# bootstrap average signal in 10 bp bins across all transcripts
#------------------------------------------------#

set.seed(11)
 df <- metaSubsampleMatrix(countsmat, binsize = 10,
                             sample.name = "PROseq",
                             ncores = 2)
df[1:10, ]

#------------------------------------------------#
# the same, using a normalization factor, and changing the x-values
#------------------------------------------------#

set.seed(11)
df <- metaSubsampleMatrix(countsmat, binsize = 10,
                            first.output.xval = 0, NF = 0.75,
                            sample.name = "PROseq", ncores = 2)
df[1:10, ]
```

---

genebodies   *Extract Genebodies*

Description

This function returns ranges that are defined relative to the strand-specific start and end sites of regions of interest (usually genes).

Usage

```
genebodies(
  genelist,
  start = 300,
  end = -300,
  fix.start = "start",
  fix.end = "end",
  min.window = 0
)
```

Arguments

genelist	A GRanges object containing genes of interest.
start	Depending on fix.start, the distance from either the strand-specific start or end site to begin the returned ranges. If positive, the returned range will begin downstream of the reference position; negative numbers are used to return sites upstream of the reference. Set start = 0 to return the reference position.
end	Identical to the start argument, but defines the strand-specific end position of returned ranges. end must be downstream of start.
fix.start	The reference point to use for defining the strand-specific start positions of returned ranges, either "start" or "end".
fix.end	The reference point to use for defining the strand-specific end positions of returned ranges, either "start" or "end". Cannot be set to "start" if fix.start = "end".
min.window	When fix.start = "start" and fix.end = "end", min.window defines the minimum size (width) of a returned range. However, when fix.end = fix.start, all returned ranges have the same width, and min.window simply size-filters the input ranges.

Details

Unlike [GenomicRanges::promoters](), distances can be defined to be upstream or downstream by changing the sign of the argument, and both the start and end of the returned regions can be defined in terms of the strand-specific start or end site of the input ranges. For example, genebodies(txs,-50,150,fix.end = "start") is equivalent to promoters(txs,50,151) (the downstream edge is off by 1 because promoters keeps the downstream interval closed). The default arguments return ranges that begin 300 bases downstream of the original start positions, and end 300 bases upstream of the original end positions.

Value

A GRanges object that may be shorter than genelist due to filtering of short ranges. For example, using the default arguments, genes shorter than 600 bp would be removed.

Author(s)

Mike DeBerardine

See Also

[intra-range-methods]()

Examples

```
data("txs_dm6_chr4") # load included transcript data
txs <- txs_dm6_chr4[c(1, 2, 167, 168)]

txs

#-----------------------------------------------#
# genebody regions from 300 bp after the TSS to
# 300 bp before the polyA site
#-----------------------------------------------#
```

```
genebodies(txs, 300, -300)

#------------------------------------------------#
# promoter-proximal region from 50 bp upstream of
# the TSS to 100 bp downstream of the TSS
#------------------------------------------------#

promoters(txs, 50, 101)

genebodies(txs, -50, 100, fix.end = "start")

#------------------------------------------------#
# region from 500 to 1000 bp after the polyA site
#------------------------------------------------#

genebodies(txs, 500, 1000, fix.start = "end")
```

---

getCountsByPositions   *Get signal counts at each position within regions of interest*

Description

Get the sum of the signal in `dataset.gr` that overlaps each position within each range in `regions.gr`. If binning is used (i.e. positions are wider than 1 bp), any function can be used to summarize the signal overlapping each bin.

Usage

```
getCountsByPositions(
  dataset.gr,
  regions.gr,
  binsize = 1,
  FUN = sum,
  simplify.multi.widths = c("list", "pad 0", "pad NA"),
  field = "score",
  ncores = detectCores()
)
```

Arguments

dataset.gr	A GRanges object in which signal is contained in metadata (typically in the "score" field).
regions.gr	A GRanges object containing regions of interest.
binsize	Size of bins (in bp) to use for counting within each range of `regions.gr`. Note that counts will *not* be length-normalized.
FUN	If `binsize > 1`, the function used to aggregate the signal within each bin. By default, the signal is summed, but any function operating on a numeric vector can be used.
simplify.multi.widths	
	A string indicating the output format if the ranges in `regions.gr` have variable widths. Default is `"list"`. See details below.

field
: The metadata field of `dataset.gr` to be counted. If `length(field) > 1`, the output is a list whose elements contain the output for generated each field.

ncores
: Multiple cores can only be used if `length(field) > 1`.

Value

If the widths of all ranges in `regions.gr` are equal, a matrix is returned that contains a row for each region of interest, and a column for each position (each base if `binsize = 1`) within each region.

Use of multi-width regions of interest

If the input `regions.gr` contains ranges of varying widths, setting `simplify.multi.widths = "list"` will output a list of variable-length vectors, with each vector corresponding to an individual input region. If `simplify.multi.widths = "pad 0"` or `"pad NA"`, the output is a matrix containing a row for each range in `regions.gr`, but the number of columns is determined by the largest range in `regions.gr`. For each region of interest, columns that correspond to positions outside of the input range are set, depending on the argument, to `0` or `NA`.

Author(s)

Mike DeBerardine

See Also

getCountsByRegions

Examples

```
data("PROseq") # load included PROseq data
data("txs_dm6_chr4") # load included transcripts

#------------------------------------------------#
# counts from 0 to 50 bp after the TSS
#------------------------------------------------#

txs_pr <- promoters(txs_dm6_chr4, 0, 50) # first 50 bases
countsmat <- getCountsByPositions(PROseq, txs_pr)
countsmat[10:15, 41:50] # show only 41-50 bp after TSS

#------------------------------------------------#
# redo with 10 bp bins from 0 to 100
#------------------------------------------------#

# column 5 is sums of rows shown above

txs_pr <- promoters(txs_dm6_chr4, 0, 100)
countsmat <- getCountsByPositions(PROseq, txs_pr, binsize = 10)
countsmat[10:15, ]

#------------------------------------------------#
# same as the above, but with the average signal in each bin
#------------------------------------------------#

countsmat <- getCountsByPositions(PROseq, txs_pr, binsize = 10, FUN = mean)
countsmat[10:15, ]
```

```
#------------------------------------------------#
# standard deviation of signal in each bin
#------------------------------------------------#

countsmat <- getCountsByPositions(PROseq, txs_pr, binsize = 10, FUN = sd)
round(countsmat[10:15, ], 2)
```

---

getCountsByRegions *Get signal counts in regions of interest*

Description

Get the sum of the signal in `dataset.gr` that overlaps each range in `regions.gr`.

Usage

```
getCountsByRegions(
  dataset.gr,
  regions.gr,
  field = "score",
  ncores = detectCores()
)
```

Arguments

dataset.gr	A GRanges object in which signal is contained in metadata (typically in the "score" field).
regions.gr	A GRanges object containing regions of interest.
field	The metadata field of `dataset.gr` to be counted. If `length(field) > 1`, a dataframe is returned containing the counts for each region in each field.
ncores	Multiple cores can only be used if `length(field) > 1`.

Value

An atomic vector the same length as `regions.gr` containing the sum of the signal overlapping each range of `regions.gr`.

Author(s)

Mike DeBerardine

See Also

getCountsByPositions

Examples

```
data("PROseq") # load included PROseq data
data("txs_dm6_chr4") # load included transcripts

counts <- getCountsByRegions(PROseq, txs_dm6_chr4)

length(txs_dm6_chr4)
length(counts)
head(counts)

# Assign as metadata to the transcript GRanges
txs_dm6_chr4$PROseq <- counts

txs_dm6_chr4[1:6]
```

---

getDESeqDataSet   *Get DESeqDataSet objects for downstream analysis*

Description

This is a convenience function for generating `DESeqDataSet` objects, but this function also adds support for counting reads across non-contiguous regions.

Usage

```
getDESeqDataSet(
  dataset.list,
  regions.gr,
  sample_names = names(dataset.list),
  gene_names = NULL,
  sizeFactors = NULL,
  field = "score",
  ncores = detectCores(),
  quiet = FALSE
)
```

Arguments

dataset.list   A list of GRanges datasets that can be individually passed to [getCountsByRegions](#).

regions.gr   A GRanges object containing regions of interest.

sample_names   Names for each dataset in `dataset.list` are required, and by default the names of the list elements are used. The names must each contain the string "_rep#", where "#" is a single character (usually a number) indicating the replicate. Sample names across different replicates must be otherwise identical.

gene_names   An optional character vector giving gene names, or any other identifier over which reads should be counted. Gene names are required if counting is to be performed over non-contiguous ranges, i.e. if any genes have multiple ranges. If supplied, gene names are added to the resulting `DESeqDataSet` object.

sizeFactors   DESeq2 `sizeFactors` can be optionally applied in to the `DESeqDataSet` object in this function, or they can be applied later on, either by the user or in a call to `getDESeqResults`. Applying the `sizeFactors` later is useful if multiple sets of factors will be explored, although `sizeFactors` can be overwritten at any time.

field	Argument passed to getCountsByRegions.
ncores	Number of cores to use for read counting across all samples. Default is the total number of cores available.
quiet	If TRUE, all output messages from call to [DESeqDataSet](#) will be suppressed.

Value

A DESeqData object in which rowData are given as rowRanges, which are equivalent to regions.gr, unless there are non-contiguous gene regions (see note below). Samples (as seen in colData) are factored so that samples are grouped by replicate and condition, i.e. all non-replicate samples are treated as distinct, and the DESeq2 design = ~condition.

Use of non-contiguous gene regions

In DESeq2, genes must be defined by single, contiguous chromosomal locations. This function allows individual genes to be encompassed by multiple distinct ranges in regions.gr. To use non-contiguous gene regions, provide gene_names in which some names are duplicated. For each unique gene in gene_names, this function will generate counts across all ranges for that gene, but be aware that it will only keep the largest range for each gene in the resulting DESeqDataSet object's rowRanges.

A note on DESeq2 sizeFactors

DESeq2 sizeFactors are sample-specific normalization factors that are applied by division, i.e. $counts_{norm,i} = counts_i / sizeFactor_i$. This is in contrast to normalization factors as defined in this package (and commonly elsewhere), which are applied by multiplication. Also note that DESeq2's "normalizationFactors" are not sample specific, but rather gene specific factors used to correct for ascertainment bias across different genes (e.g. as might be relevant for GSEA or Go analysis).

Author(s)

Mike DeBerardine

See Also

[DESeq2::DESeqDataSet](#), [getDESeqResults](#)

Examples

```
suppressPackageStartupMessages(require(DESeq2))
data("PROseq") # import included PROseq data
data("txs_dm6_chr4") # import included transcripts

# divide PROseq data into 6 toy datasets
ps_a_rep1 <- PROseq[seq(1, length(PROseq), 6)]
ps_b_rep1 <- PROseq[seq(2, length(PROseq), 6)]
ps_c_rep1 <- PROseq[seq(3, length(PROseq), 6)]

ps_a_rep2 <- PROseq[seq(4, length(PROseq), 6)]
ps_b_rep2 <- PROseq[seq(5, length(PROseq), 6)]
ps_c_rep2 <- PROseq[seq(6, length(PROseq), 6)]

ps_list <- list(A_rep1 = ps_a_rep1,
```

```
                       A_rep2 = ps_a_rep2,
                       B_rep1 = ps_b_rep1,
                       B_rep2 = ps_b_rep2,
                       C_rep1 = ps_c_rep1,
                       C_rep2 = ps_c_rep2)

# make flawed dataset (ranges in txs_dm6_chr4 not disjoint)
#    this means there is double-counting
# also using discontinuous gene regions, as gene_ids are repeated
dds <- getDESeqDataSet(ps_list,
                       txs_dm6_chr4,
                       gene_names = txs_dm6_chr4$gene_id,
                       quiet = TRUE,
                       ncores = 2)
dds
```

---

getDESeqResults   *Get DESeq2 results using reduced dispersion matrices*

Description

This function calls [DESeq2::DESeq](#) and [DESeq2::results](#) on a pre-existing DESeqDataSet object and returns a DESeqResults table for one or more pairwise comparisons. However, unlike a standard call to DESeq2::results using the contrast argument, this function subsets the dataset so that DESeq2 only estimates dispersion for the samples being compared, and not for all samples present.

Usage

```
getDESeqResults(
  dds,
  contrast.numer,
  contrast.denom,
  comparisons = NULL,
  sizeFactors = NULL,
  alpha = 0.1,
  args.DESeq = NULL,
  args.results = NULL,
  ncores = detectCores(),
  quiet = FALSE
)
```

Arguments

dds   A DESeqDataSet object, produced using either [getDESeqDataSet](#) from this
                package or [DESeqDataSet](#) from DESeq2. If dds was not created using getDESeqDataSet,
                dds must be made with design = ~condition such that a unique condition
                level exists for each sample/treatment condition.

contrast.numer A string naming the condition to use as the numerator in the DESeq2 compar-
                ison, typically the perturbative condition.

contrast.denom A string naming the condition to use as the denominator in the DESeq2 com-
                parison, typically the control condition.

comparisons	As an optional alternative to supplying a single `contrast.numer` and `contrast.denom`, users can supply a list of character vectors containing numerator-denominator pairs, e.g. `list(c("B","A"),c("C","A"),c("C","B"))`. `comparisons` can also be a dataframe in which each row is a comparison, the first column contains the numerators, and the second column contains the denominators.
sizeFactors	A vector containing DESeq2 sizeFactors to apply to each sample. Each sample's readcounts are *divided* by its respective DESeq2 sizeFactor. A warning will be generated if the DESeqDataSet already contains sizeFactors, and the previous sizeFactors will be over-written.
alpha	The significance threshold passed to DESeqResults. This won't affect the output results, but is used as a performance optimization by DESeq2.
args.DESeq	Additional arguments passed to [DESeq](#), given as a list of argument-value pairs, e.g. `list(test = "LRT", fitType = "local")`. All arguments given here will be passed to DESeq except for `object` and `parallel`. If no arguments are given, all defaults will be used.
args.results	Additional arguments passed to [DESeq2::results](#), given as a list of argument-value pairs, e.g. `list(altHypothesis = "greater", lfcThreshold = 1.5)`. All arguments given here will be passed to `results` except for `object`, `contrast`, `alpha`, and `parallel`. If no arguments are given, all defaults will be used.
ncores	The number of cores to use for parallel processing. Multicore processing is only used if more than one comparison is being made (i.e. argument `comparisons` is used), and the number of cores utilized will not be greater than the number of comparisons being performed.
quiet	If `TRUE`, all output messages from calls to DESeq and results will be suppressed, although passing option `quiet` in `args.DESeq` will supersede this option for the call to DESeq.

Value

For a single comparison, the output is the DESeqResults result table. If a `comparisons` is used to make multiple comparisons, the output is a named list of DESeqResults objects, with elements named following the pattern `"X_vs_Y"`, where `X` is the name of the numerator condition, and `Y` is the name of the denominator condition.

Author(s)

Mike DeBerardine

See Also

[getDESeqDataSet](#), [DESeq2::results](#)

Examples

```
#-------------------------------------------------#
# getDESeqDataSet
#-------------------------------------------------#
suppressPackageStartupMessages(require(DESeq2))
data("PROseq") # import included PROseq data
data("txs_dm6_chr4") # import included transcripts

# divide PROseq data into 6 toy datasets
```

```
ps_a_rep1 <- PROseq[seq(1, length(PROseq), 6)]
ps_b_rep1 <- PROseq[seq(2, length(PROseq), 6)]
ps_c_rep1 <- PROseq[seq(3, length(PROseq), 6)]

ps_a_rep2 <- PROseq[seq(4, length(PROseq), 6)]
ps_b_rep2 <- PROseq[seq(5, length(PROseq), 6)]
ps_c_rep2 <- PROseq[seq(6, length(PROseq), 6)]

ps_list <- list(A_rep1 = ps_a_rep1,
                A_rep2 = ps_a_rep2,
                B_rep1 = ps_b_rep1,
                B_rep2 = ps_b_rep2,
                C_rep1 = ps_c_rep1,
                C_rep2 = ps_c_rep2)

# make flawed dataset (ranges in txs_dm6_chr4 not disjoint)
#     this means there is double-counting
# also using discontinuous gene regions, as gene_ids are repeated
dds <- getDESeqDataSet(ps_list,
                       txs_dm6_chr4,
                       gene_names = txs_dm6_chr4$gene_id,
                       ncores = 2)

dds

#-----------------------------------------------#
# getDESeqResults
#-----------------------------------------------#

res <- getDESeqResults(dds, "B", "A")

res

reslist <- getDESeqResults(dds,
                           comparisons = list(c("B", "A"), c("C", "A")),
                           ncores = 1)
names(reslist)

reslist$B_vs_A

# or using a dataframe
reslist <- getDESeqResults(dds, comparisons = data.frame(num = c("B", "C"),
                                                         den = c("A", "A")),
                           ncores = 1)
reslist$B_vs_A
```

---

getMaxPositionsBySignal

Find sites with max signal in regions of interest

Description

For each signal-containing region of interest, find the single site with the most signal. Sites can be found at base-pair resolution, or defined for larger bins.

Usage

```
getMaxPositionsBySignal(
  regions.gr,
  dataset.gr,
  binsize = 1,
  bin.centers = FALSE,
  field = "score",
  keep.score = FALSE
)
```

Arguments

regions.gr	A GRanges object containing regions of interest.
dataset.gr	A GRanges object in which signal is contained in metadata (typically in the "score" field).
binsize	The size of bin in which to calculate signal scores.
bin.centers	Logical indicating if the centers of bins are returned, as opposed to the entire bin. By default, entire bins are returned.
field	The metadata field of dataset.gr to be counted.
keep.score	Logical indicating if the signal value at the max site should be reported. If set to TRUE, the values are kept as a new metadata column in regions.gr.

Value

Output is a GRanges object with regions.gr metadata, but each range only contains the site within each regions.gr range that had the most signal. If binsize > 1, the entire bin is returned, unless bin.centers = TRUE, in which case a single-base site is returned. The site is set to the center of the bin, and if the binsize is even, the site is rounded to be closer to the beginning of the range.

If keep.score = TRUE, the output will also contain metadata for the signal at the max site. The output is *not* necessarily same length as regions.gr, as regions without signal are not returned. If *no regions* have signal (e.g. as could happen if running this function on single regions), the function will return an empty GRanges object with intact metadata columns.

Author(s)

Mike DeBerardine

See Also

getCountsByPositions

Examples

```
data("PROseq") # load included PROseq data
data("txs_dm6_chr4") # load included transcripts

#------------------------------------------------#
# first 50 bases of transcripts
#------------------------------------------------#

pr <- promoters(txs_dm6_chr4, 0, 50)
pr[1:3]
```

```
#------------------------------------------------#
# max sites
#------------------------------------------------#

getMaxPositionsBySignal(pr[1:3], PROseq, keep.score = TRUE)

#------------------------------------------------#
# max sites in 5 bp bins
#------------------------------------------------#

getMaxPositionsBySignal(pr[1:3], PROseq, binsize = 5, keep.score = TRUE)
```

---

getPausingIndices   *Calculate pausing indices from user-supplied promoters & genebodies*

Description

Pausing index (PI) is calculated for each gene (within matched `promoters.gr` and `genebodies.gr`) as promoter-proximal (or pause region) signal counts divided by genebody signal counts. If `length.normalize` = TRUE (recommended), the signal counts within each range in `promoters.gr` and `genebodies.gr` are divided by their respective range widths (region lengths) before pausing indices are calculated.

Usage

```
getPausingIndices(
  dataset.gr,
  promoters.gr,
  genebodies.gr,
  field = "score",
  length.normalize = TRUE,
  remove.empty = FALSE,
  ncores = detectCores()
)
```

Arguments

dataset.gr	A GRanges object in which signal is contained in metadata (typically in the "score" field).
promoters.gr	A GRanges object containing promoter-proximal regions of interest.
genebodies.gr	A GRanges object containing genebody regions of interest.
field	The metadata field of `dataset.gr` to be counted. If `length(field) > 1`, a dataframe is returned containing the pausing indices for each region in each field.
length.normalize	
	A logical indicating if signal counts within regions of interest should be length normalized. The default is `TRUE`, which is recommended, especially if input regions don't all have the same width.
remove.empty	A logical indicating if genes without any signal in `promoters.gr` should be removed. No genes are filtered by default.
ncores	Multiple cores can only be used if `length(field) > 1`.

Value

A vector parallel to the input genelist, unless `remove.empty = TRUE`, in which case the vector may
be shorter. If `length(field) > 1`, a dataframe is returned, containing a column for each field.

Author(s)

Mike DeBerardine

See Also

[getCountsByRegions](#)

Examples

```
data("PROseq") # load included PROseq data
data("txs_dm6_chr4") # load included transcripts

#-------------------------------------------------#
# Get promoter-proximal and genebody regions
#-------------------------------------------------#

# genebodies from +300 to 300 bp before the poly-A site
gb <- genebodies(txs_dm6_chr4, 300, -300, min.window = 400)

# get the transcripts that are large enough (>1kb in size)
txs <- subset(txs_dm6_chr4, tx_name %in% gb$tx_name)

# for the same transcripts, promoter-proximal region from 0 to +100
pr <- promoters(txs, 0, 100)

#-------------------------------------------------#
# Calculate pausing indices
#-------------------------------------------------#

pidx <- getPausingIndices(PROseq, pr, gb)

length(txs)
length(pidx)
head(pidx)

#-------------------------------------------------#
# Without length normalization
#-------------------------------------------------#

head( getPausingIndices(PROseq, pr, gb, length.normalize = FALSE) )

#-------------------------------------------------#
# Removing empty means the values no longer match the genelist
#-------------------------------------------------#

pidx_signal <- getPausingIndices(PROseq, pr, gb, remove.empty = TRUE)

length(pidx_signal)
```

getStrandedCoverage *Get strand-specific coverage*

Description

Computes strand-specific coverage signal, and returns a GRanges object. Function also works for non-strand-specific data.

Usage

```
getStrandedCoverage(dataset.gr, field = "score")
```

Arguments

dataset.gr   A GRanges object either containing ranges for each read, or one in which read-counts for individual ranges are contained in metadata (typically in the "score" field).

field   The name of the metadata field that contains readcounts. If no metadata field contains readcounts, and each range represents a single read, set to NULL.

Value

A GRanges object with signal in the "score" metadata column. Note that the output is *not* automatically converted into a "basepair-resolution" GRanges object.

Author(s)

Mike DeBerardine

See Also

makeGRangesBRG, GenomicRanges::coverage

Examples

```
#------------------------------------------------#
# Using included full-read data
#------------------------------------------------#
# -> whole-read coverage sacrifices meaningful readcount
#    information, but can be useful for visualization,
#    e.g. for looking at RNA-seq data in a genome browser

data("PROseq_paired")

PROseq_paired[1:6]

getStrandedCoverage(PROseq_paired)[1:6]

#------------------------------------------------#
# Getting coverage from single bases of single reads
#------------------------------------------------#

# included PROseq data is already single-base coverage
```

```
data("PROseq")
range(width(PROseq))

# undo coverage for the first 100 positions
ps <- PROseq[1:100]
ps_reads <- rep(ps, times = ps$score)
mcols(ps_reads) <- NULL

ps_reads[1:6]

# re-create coverage
getStrandedCoverage(ps_reads, field = NULL)[1:6]

#-----------------------------------------------#
# Reversing makeGRangesBRG
#-----------------------------------------------#
# -> getStrandedCoverage doesn't return single-width
#    GRanges, which is useful because getting coverage
#    will merge adjacent bases with equivalent scores

# included PROseq data is already single-width
range(width(PROseq))
isDisjoint(PROseq)

ps_cov <- getStrandedCoverage(PROseq)

range(width(ps_cov))
sum(score(PROseq)) == sum(score(ps_cov) * width(ps_cov))

# -> Look specifically at ranges that could be combined
neighbors <- c(shift(PROseq, 1), shift(PROseq, -1))
hits <- findOverlaps(PROseq, neighbors)
idx <- unique(from(hits)) # indices for PROseq with neighbor

PROseq[idx]

getStrandedCoverage(PROseq[idx])
```

---

import-functions   *Import basepair-resolution files*

Description

Import functions for plus/minus pairs of `bigWig` or `bedGraph` files.

Usage

```
import_bigWig(
  plus_file,
  minus_file,
  genome = NULL,
  keep.X = TRUE,
  keep.Y = TRUE,
  keep.M = FALSE,
```

```
    keep.nonstandard = FALSE
)

import_bedGraph(
  plus_file,
  minus_file,
  genome = NULL,
  keep.X = TRUE,
  keep.Y = TRUE,
  keep.M = FALSE,
  keep.nonstandard = FALSE
)
```

Arguments

`plus_file, minus_file`
> Paths for strand-specific input files.

`genome`   Optional string for UCSC reference genome, e.g. "hg38". If given, non-standard chromosomes are trimmed, and options for sex and mitochondrial chromosomes are applied.

`keep.X, keep.Y, keep.M, keep.nonstandard`
> Logicals indicating which non-autosomes should be kept. By default, sex chromosomes are kept, but mitochondrial and non-standard chromosomes are removed.

Details

For `import_bigWig`, the output GRanges is formatted by [makeGRangesBRG](#), such that all ranges are disjoint and have width = 1, and the `score` is single-base coverage, i.e. the number of reads for each position.

`import_bedGraph` is useful for when both 5'- and 3'-end information is to be maintained for each sequenced molecule. It effectively imports the entire read, and the `score` represents the number of reads sharing identical 5' and 3' ends.

Value

Imports a GRanges object containing base-pair resolution data, with the `score` metadata column indicating the number of reads represented by each range.

Author(s)

Mike DeBerardine

See Also

[tidyChromosomes](#), [rtracklayer::import](#)

Examples

```
#-----------------------------------------------#
# Import PRO-seq bigWigs -> coverage of 3' bases
#-----------------------------------------------#
```

```
# get local address for included bigWig files
p.bw <- system.file("extdata", "PROseq_dm6_chr4_plus.bw",
                    package = "BRGenomics")
m.bw <- system.file("extdata", "PROseq_dm6_chr4_minus.bw",
                    package = "BRGenomics")

# import bigWigs
PROseq <- import_bigWig(p.bw, m.bw, genome = "dm6")
PROseq

#-----------------------------------------------#
# Import PRO-seq bedGraphs -> whole reads (matched 5' and 3' ends)
#-----------------------------------------------#

# get local address for included bedGraph files
p.bg <- system.file("extdata", "PROseq_dm6_chr4_plus.bedGraph",
                    package = "BRGenomics")
m.bg <- system.file("extdata", "PROseq_dm6_chr4_minus.bedGraph",
                    package = "BRGenomics")

# import bedGraphs
PROseq_paired <- import_bedGraph(p.bg, m.bg, genome = "dm6")
PROseq_paired
```

---

import_bam   *Import bam files*

Description

Import bam files

Usage

```
import_bam(
  file,
  mapq = 20,
  revcomp = FALSE,
  trim.to = c("whole", "5p", "3p", "center"),
  ignore.strand = FALSE,
  field = "score"
)
```

Arguments

file	Path of a bam file.
mapq	Filter reads by a minimum MapQ score. This is the correct way to filter multi-aligners.
revcomp	Logical indicating if aligned reads should be reverse-complemented.
trim.to	Option for selecting specific bases from the reads, applied after the revcomp option. By default, the entire read is maintained. Other options are to take only the 5' base, only the 3' base, or the only the center base of the read.

ignore.strand   Logical indicating if the strand information should be discarded. If TRUE, strand information is discarded *after* revcomp and trim.to options are applied.

field   Metadata field name to use for readcounts, usually "score". If set to NULL, identical reads (or identical positions if trim.to options applied) are not combined, and the length of the output GRanges will be equal to the number of input reads.

Value

A GRanges object.

Examples

```
# get local address for included bam file
ps.bam <- system.file("extdata", "PROseq_dm6_chr4.bam",
                      package = "BRGenomics")

#-----------------------------------------------#
# Import entire reads
#-----------------------------------------------#

# Note that PRO-seq reads are sequenced as reverse complement
import_bam(ps.bam, revcomp = TRUE)

#-----------------------------------------------#
# Import entire reads, 1 range per read
#-----------------------------------------------#

import_bam(ps.bam, revcomp = TRUE, field = NULL)

#-----------------------------------------------#
# Import PRO-seq reads at basepair-resolution
#-----------------------------------------------#

# the typical manner to import PRO-seq data:
import_bam(ps.bam, revcomp = TRUE, trim.to = "3p")

#-----------------------------------------------#
# Import 5' ends of PRO-seq reads
#-----------------------------------------------#

# will include bona fide TSSes as well as hydrolysis products
import_bam(ps.bam, revcomp = TRUE, trim.to = "5p")
```

---

makeGRangesBRG   *Make base-pair resolution GRanges object*

Description

Splits up all ranges in dataset.gr to be each 1 basepair wide. For any range that is split up, all metadata information belonging to that range is inherited by its daughter ranges, and therefore the transformation is non-destructive.

Usage

```
makeGRangesBRG(dataset.gr)
```

Arguments

dataset.gr   A disjoint GRanges object

Details

Note that this function doesn't perform any transformation on the metadata in the input. This function assumes that for an input GRanges object, any metadata for each range is equally correct when inherited by each individual base in that range. In other words, the dataset's "signal" (usually readcounts) fundamentally belongs to a single basepair position.

Value

A GRanges object for which `length(output) == sum(width(dataset.gr))`, and for which `all(width(output) == 1)`.

Motivation

The motivating case for this function is a bigWig file (e.g. one imported by rtracklayer), as big-Wig files typically use run-length compression on the data signal (the 'score' column), such that adjacent bases sharing the same signal are combined into a single range. As basepair-resolution genomic data is typically sparse, this compression has a minimal impact on memory usage, and removing it greatly enhances data handling as each index (each range) of the GRanges object corresponds to a single genomic position.

Generating basepair-resolution GRanges from whole reads

If working with a GRanges object containing whole reads, one can obtain base-pair resolution information by using the strand-specific function `GenomicRanges::resize` to select a single base from each read: set `width = 1` and use the `fix` argument to choose the strand-specific 5' or 3' end. Then, strand-specific coverage can be calculated using `getStrandedCoverage`.

On the use of GRanges instead of GPos

The `GPos` class is a more suitable container for data of this type, as the GPos class is specific to 1-bp-wide ranges. However, in early testing, we encountered some kind of compatibility limitations with the newer GPos class, and have not re-tested it since. If you have feedback on switching to this class, please contact the author. Users can readily coerce a basepair-resolution GRanges object to a GPos object via `gp <-GPos(gr,score = score(gr))`.

Author(s)

Mike DeBerardine

See Also

`getStrandedCoverage`, `GenomicRanges::resize()`

Examples

```
#-------------------------------------------------#
# Make a bigWig file single width
#-------------------------------------------------#

# get local address for an included bigWig file
bw_file <- system.file("extdata", "PROseq_dm6_chr4_plus.bw",
                       package = "BRGenomics")

# BRGenomics::import_bigWig automatically applies makeGRangesBRG;
# therefore will import using rtracklayer
bw <- rtracklayer::import.bw(bw_file)
strand(bw) <- "+"

range(width(bw))
length(bw)

# make basepair-resolution (single-width)
gr <- makeGRangesBRG(bw)

range(width(gr))
length(gr)
length(gr) == sum(width(bw))
sum(score(gr)) == sum(score(bw) * width(bw))

#-------------------------------------------------#
# Reverse using getStrandedCoverage
#-------------------------------------------------#
# -> for more examples, see getStrandedCoverage

undo <- getStrandedCoverage(gr)

range(width(undo))
length(undo) == length(bw)
all(score(undo) == score(bw))
```

---

mergeGRangesData   *Merge basepair-resolution GRanges objects*

Description

Merges 2 or more basepair-resolution (single-width) GRanges objects by combining all of their ranges and associated signal (e.g. readcounts).

Usage

```
mergeGRangesData(..., field = "score", ncores = detectCores())
```

Arguments

...	Any number of GRanges objects in which signal (e.g. readcounts) are contained within metadata. GRanges not single-width will be coerced using makeGRangesBRG.
field	One or more metadata fields to be combined, typically the "score" field. Fields typically contain coverage information.

ncores   More than one core can be used to coerce non-single-width GRanges objects
                using makeGRangesBRG.

Value

A disjoint, basepair-resolution (single-width) GRanges object comprised of all ranges found in the
input GRanges objects, and whose total readcount is the sum of all input GRanges.

Author(s)

Mike DeBerardine

See Also

[makeGRangesBRG](#)

Examples

```
data("PROseq") # load included PROseq data

#-----------------------------------------------#
# divide & recombine PROseq (no overlapping positions)
#-----------------------------------------------#

thirds <- floor( (1:3)/3 * length(PROseq) )
ps_1 <- PROseq[1:thirds[1]]
ps_2 <- PROseq[(thirds[1]+1):thirds[2]]
ps_3 <- PROseq[(thirds[2]+1):thirds[3]]

# re-merge
length(PROseq)
length(ps_1)
length(mergeGRangesData(ps_1, ps_2))
length(mergeGRangesData(ps_1, ps_2, ps_3))

#-----------------------------------------------#
# combine PRO-seq with overlapping positions
#-----------------------------------------------#

gr1 <- PROseq[10:13]
gr2 <- PROseq[12:15]

PROseq[10:15]

mergeGRangesData(gr1, gr2)
```

PROseq   *PRO-seq data from Drosophila S2 cells*

Description

PRO-seq data of Drosophila S2 cells, chromosome 4.

Usage

```
data(PROseq)
```

Format

A disjoint GRanges object with 47533 ranges with 1 metadata column:

score coverage of PRO-seq read 3'-ends ...

Details

Hojoong Kwak, Nicholas J. Fuda, Leighton J. Core, John T. Lis (2013). Precise Maps of RNA Polymerase Reveal How Promoters Direct Initiation and Pausing. Science, 339(6122), 950–953. https://doi.org/10.1126/science.1229386

Source

GEO Accession GSM1032758, run SRR611828.

PROseq_paired   *Paired PRO-seq data from Drosophila S2 cells*

Description

PRO-seq data of Drosophila S2 cells, chromosome 4. Entire mapped reads kept.

Usage

```
data(PROseq_paired)
```

Format

A GRanges object with 52464 ranges with 1 metadata column:

score number of reads sharing the same mapped 5' and 3' ends ...

Details

Hojoong Kwak, Nicholas J. Fuda, Leighton J. Core, John T. Lis (2013). Precise Maps of RNA Polymerase Reveal How Promoters Direct Initiation and Pausing. Science, 339(6122), 950–953. https://doi.org/10.1126/science.1229386

Source

GEO Accession GSM1032758, run SRR611828.

---

subsampleGRanges   *Randomly subsample reads from GRanges dataset*

Description

Random subsampling is not performed on ranges, but on reads. Readcounts should be given as a metadata field (usually "score"). This function can also subsample ranges directly if field = NULL, but the sample function can be used in this scenario.

Usage

```
subsampleGRanges(dataset.gr, n = NULL, prop = NULL, field = "score")
```

Arguments

dataset.gr   A GRanges object in which signal (e.g. readcounts) are contained within meta-
              data.

n   Number of reads to subsample. Either n or prop can be given.

prop   Proportion of total signal to subsample.

field   The metadata field of dataset.gr that contains readcounts for reach position.
              If each range represents a single read, set field = NULL

Value

A GRanges object identical in format to dataset.gr, but containing a random subset of its data. If field != NULL, the length of the output cannot be known *a priori*, but the sum of its score can.

Use with normalized readcounts

If the metadata field contains normalized readcounts, an attempt will be made to infer the normalization factor based on the least-common-multiple of the signal found in the specified field.

Author(s)

Mike DeBerardine

Examples

```
data("PROseq") # load included PROseq data

#-----------------------------------------------#
# sample 10% of the reads of a GRanges with signal coverage
#-----------------------------------------------#

ps_sample <- subsampleGRanges(PROseq, prop = 0.1)

# cannot predict number of ranges (positions) that will be sampled
length(PROseq)
length(ps_sample)

# 1/10th the score is sampled
sum(score(PROseq))
```

```
sum(score(ps_sample))

#------------------------------------------------#
# Sample 10% of ranges (e.g. if each range represents one read)
#------------------------------------------------#

ps_sample <- subsampleGRanges(PROseq, prop = 0.1, field = NULL)

length(PROseq)
length(ps_sample)

# Alternatively
ps_sample <- sample(PROseq, 0.1 * length(PROseq))
length(ps_sample)
```

---

subsetRegionsBySignal   *Subset regions of interest by quantiles of overlapping signal*

Description

A convenience function to subset regions of interest by the amount of signal they contain, according to their quantile (i.e. their signal ranks).

Usage

```
subsetRegionsBySignal(
  regions.gr,
  dataset.gr,
  quantiles = c(0.5, 1),
  field = "score",
  order.by.rank = FALSE,
  density = FALSE
)
```

Arguments

regions.gr	A GRanges object containing regions of interest.
dataset.gr	A GRanges object in which signal is contained in metadata (typically in the "score" field).
quantiles	A value pair giving the lower quantile and upper quantile of regions to keep. Regions with signal quantiles below the lower quantile are removed, and likewise for regions with signal quantiles above the upper quantile. Quantiles must be in range (0,1). An empty GRanges object is returned if the lower quantile is set to 1 or if the upper quantile is set to 0.
field	The metadata field of dataset.gr to be counted, typically "score".
order.by.rank	If TRUE, the output regions are sorted based on the amount of overlapping signal (in decreasing order). If FALSE (the default), genes are sorted by their positions.
density	A logical indicating whether signal counts should be normalized to the width (chromosomal length) of ranges in regions.gr. By default, no length normalization is performed.

Value

A GRanges object of length length(regions.gr) * (upper_quantile -lower_quantile).

Author(s)

Mike DeBerardine

See Also

[getCountsByRegions](#)

Examples

```
data("PROseq") # load included PROseq data
data("txs_dm6_chr4") # load included transcripts

txs_dm6_chr4

#-----------------------------------------------#
# get the top 50% of transcripts by signal
#-----------------------------------------------#

subsetRegionsBySignal(txs_dm6_chr4, PROseq)

#-----------------------------------------------#
# get the middle 50% of transcripts by signal
#-----------------------------------------------#

subsetRegionsBySignal(txs_dm6_chr4, PROseq, quantiles = c(0.25, 0.75))

#-----------------------------------------------#
# get the top 10% of transcripts by signal, and sort them by highest signal
#-----------------------------------------------#

subsetRegionsBySignal(txs_dm6_chr4, PROseq,
                      quantiles = c(0.9, 1),
                      order.by.rank = TRUE)

#-----------------------------------------------#
# remove the most extreme 10% of regions
#-----------------------------------------------#

subsetRegionsBySignal(txs_dm6_chr4, PROseq,
                      quantiles = c(0.05, 0.95))
```

---

tidyChromosomes *Remove odd chromosomes from GRanges objects*

Description

This convenience function removes non-standard, mitochondrial, and/or sex chromosomes from any GRanges object.

Usage

```
tidyChromosomes(
  gr,
  keep.X = TRUE,
  keep.Y = TRUE,
  keep.M = FALSE,
  keep.nonstandard = FALSE
)
```

Arguments

gr   Any GRanges object, however the object should have a standard genome set,
                        e.g. genome(gr) <-"hg38"

keep.X, keep.Y, keep.M, keep.nonstandard
                        Logicals indicating which non-autosomes should be kept. By default, sex chro-
                        mosomes are kept, but mitochondrial and non-standard chromosomes are re-
                        moved.

Details

Standard chromosomes are defined using the [standardChromosomes](#) function from the GenomeInfoDb
package.

Value

A GRanges object in which both ranges and seqinfo associated with trimmed chromosomes have
been removed.

Author(s)

Mike DeBerardine

See Also

[GenomeInfoDb::standardChromosomes](#)

Examples

```
# make a GRanges
chrom <- c("chr2", "chr3", "chrX", "chrY", "chrM", "junk")
gr <- GRanges(seqnames = chrom,
              ranges = IRanges(start = 2*(1:6), end = 3*(1:6)),
              strand = "+",
              seqinfo = Seqinfo(chrom))
genome(gr) <- "hg38"

gr

tidyChromosomes(gr)

tidyChromosomes(gr, keep.M = TRUE)

tidyChromosomes(gr, keep.M = TRUE, keep.Y = FALSE)

tidyChromosomes(gr, keep.nonstandard = TRUE)
```

txs_dm6_chr4	*Ensembl transcripts for Drosophila melanogaster, dm6, chromosome 4.*

Description

Transcripts obtained from annotation package TxDb.Dmelanogaster.UCSC.dm6.ensGene, which was in turn made by the Bioconductor Core Team from UCSC resources on 2019-04-25. Metadata columns were obtained from "TXNAME" and "GENEID" columns. Data exported from the TxDb package using GenomicFeatures version 1.35.11 on 2019-12-19.

Usage

```
data(txs_dm6_chr4)
```

Format

A GRanges object with 339 ranges and 2 metadata columns:

tx_name Flybase unique identifiers for transcripts

gene_id FLybase unique identifiers for the associated genes

Source

TxDb.Dmelanogaster.UCSC.dm6.ensGene version 3.4.6

Index