Package 'BRGenomics'

March 29, 2021

Type Package

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

Version 1.2.0

Description This package provides useful and efficient utilites for the analysis of high-resolution genomic data using standard Bioconductor methods and classes. BRGenomics is feature-rich and simplifies a number of post-alignment processing steps and data handling. Emphasis is on efficient analysis of multiple datasets, with support for normalization and blacklisting. Included are functions for: spike-in normalizing data; generating basepair-resolution readcounts and coverage data (e.g. for heatmaps); importing and processing bam files (e.g. for conversion to bigWig files); generating metaplots/metaprofiles (bootstrapped mean profiles) with confidence intervals; conveniently calling DESeq2 without using sample-blind estimates of genewise dispersion; among other features.

License Artistic-2.0

URL https://mdeber.github.io

BugReports https://github.com/mdeber/BRGenomics/issues

Encoding UTF-8

LazyData FALSE

RoxygenNote 7.1.0

Depends R (>= 4.0), rtracklayer, GenomeInfoDb, S4Vectors

Imports GenomicRanges, parallel, IRanges, stats, Rsamtools, GenomicAlignments, DESeq2, SummarizedExperiment, utils, methods

Suggests BiocStyle, knitr, rmarkdown, testthat, apeglm, remotes, ggplot2, reshape2, Biostrings

biocViews Software, DataImport, Sequencing, Coverage, RNASeq, ATACSeq, ChIPSeq, Transcription, GeneRegulation, GeneExpression, Normalization

VignetteBuilder knitr

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

git_branch RELEASE_3_12

git_last_commit 79bb3d5

git_last_commit_date 2020-10-27

Date/Publication 2021-03-29

Author Mike DeBerardine [aut, cre]

Maintainer Mike DeBerardine <mike.deberardine@gmail.com>

R topics documented:

`BRGenomics-package` *BRGenomics: Tools for the Efficient Analysis of High-Resolution Genomics Data*

Description

BRGenomics provides useful functions for analyzing genomics data at base-pair resolution, and for doing so in a way that maximizes compatibility with the wide array of packages available through Bioconductor.

For interactive documentation with code examples, see the online documentation: https://mdeber.github.io/

Author(s)

Mike DeBerardine <mike.deberardine@gmail.com>

applyNFsGRanges *Apply normalization factors to GRanges object*

Description

Convenience function for multiplying signal counts in one or more GRanges object by their normalization factors.

Usage

```
applyNFsGRanges(
  dataset.gr,
  NF,
  field = "score",
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

dataset.gr	A GRanges object with signal data in one or more metadata fields, or a list of such GRanges objects.
NF	One or more normalization factors to apply by multiplication. The number of normalization factors should match the number of datasets in dataset.gr.
field	The metadata field(s) in dataset.gr that contain signal to be normalized.
ncores	The number of cores to use for computations. Multicore only used if there are multiple datasets present.

Value

A GRanges object, or a list of GRanges objects.

Author(s)

Mike DeBerardine

See Also

getSpikeInNFs

Examples

```
# Apply NFs to a single GRanges
gr <- GRanges(seqnames = "chr1",
              ranges = IRanges(1:3, 3:5),
              strand = c("+", "+", "-"),
              score = c(2, 3, 4))
gr

applyNFsGRanges(gr, NF = 0.5, ncores = 1)

# Apply NFs to a list of GRanges
gr2 <- gr
```

```
ranges(gr2) <- IRanges(4:6, 5:7)
grl <- list(gr, gr2)
grl

applyNFsGRanges(grl, NF = c(0.5, 0.75), ncores = 1)

# Apply NFs to a multiplexed GRanges
gr_multi <- gr
names(mcols(gr_multi)) <- "gr1"
gr_multi$gr2 <- c(3, 5, 7)
gr_multi

applyNFsGRanges(gr_multi, NF = c(2, 3), field = c("gr1", "gr2"),
                ncores = 1)
```

`binNdimensions` *Generating and Aggregating Data Within N-dimensional Bins*

Description

Divide data along different dimensions into equally spaced bins, and summarize the datapoints that fall into any of these n-dimensional bins.

Usage

```
binNdimensions(
  dims.df,
  nbins = 10,
  use_bin_numbers = TRUE,
  ncores = getOption("mc.cores", 2L)
)

aggregateByNdimBins(
  x,
  dims.df,
  nbins = 10,
  FUN = mean,
  ...,
  ignore.na = TRUE,
  drop = FALSE,
  empty = NA,
  use_bin_numbers = TRUE,
  ncores = getOption("mc.cores", 2L)
)

densityInNdimBins(
  dims.df,
  nbins = 10,
  use_bin_numbers = TRUE,
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

dims.df	A dataframe containing one or more columns of numerical data for which bins will be generated.
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.
use_bin_numbers	A logical indicating if ordinal bin numbers should be returned (TRUE), or if in place of the bin number, the center value of that bin should be returned. For instance, if the first bin encompasses data from 1 to 3, with use_bin_numbers = TRUE, a 1 is returned, but when FALSE, 2 is returned.
ncores	Number of cores to use for computations.
x	The name of the dimension in dims.df to aggregate, or a separate numerical vector or dataframe of data to be aggregated. If x is a numerical vector, each value in x corresponds to a row of dims.df, and so length(x) must be equal to nrow(dims.df). Likewise, if x is a dataframe, nrow(x) must equal nrow(dims.df). Supplying a dataframe for x has the advantage of simultaneously aggregating different sets of data, and returning a single dataframe.
FUN	A function to use for aggregating data within each bin.
...	Additional arguments passed to FUN.
ignore.na	Logical indicating if NA values of x should be ignored. Default is TRUE.
drop	A logical indicating if empty bin combinations should be removed from the output. By default (FALSE), all possible combinations of bins are returned, and empty bins contain a value given by empty.
empty	When drop = FALSE, the value returned for empty bins. By default, empty bins return NA. However, in many circumstances (e.g. if FUN = sum), the empty value should be 0.

Details

These functions take in data along 1 or more dimensions, and for each dimension the data is divided into evenly-sized bins from the minimum value to the maximum value. For instance, if each row of dims.df were a gene, the columns (the different 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.

binNdimensions returns the bin numbers themselves. The output dataframe has the same dimensions as the input dims.df, but each input data has been replaced by its bin number (an integer). If codeuse_bin_numbers = FALSE, the center points of the bins are returned instead of the bin numbers.

aggregateByNdimBins summarizes some input data x in each combination of bins, i.e. in each n-dimensional bin. Each row of the output dataframe is a unique combination of the input bins (i.e. each n-dimensional bin), and the output columns are identical to those in dims.df, with the addition of one or more columns containing the aggregated data in each n-dimensional bin. If the input x was a vector, the column is named "value"; if the input x was a dataframe, the column names from x are maintained.

densityInNdimBins returns a dataframe just like aggregateByNdimBins, except the "value" column contains the number of observations that fall into each n-dimensional bin.

Value

A dataframe.

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")

df <- data.frame(counts_pr = getCountsByRegions(PROseq, pr),
                 counts_gb = getCountsByRegions(PROseq, early_gb),
                 counts_cps = getCountsByRegions(PROseq, cps))

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

bin3d <- binNdimensions(df, nbins = 20, ncores = 1)

length(txs_dm6_chr4)
nrow(bin3d)
bin3d[1:6, ]

#-------------------------------------------------#
# get number of genes in each bin
#-------------------------------------------------#

bin_counts <- densityInNdimBins(df, nbins = 20, ncores = 1)

bin_counts[1:6, ]

#-------------------------------------------------#
# get mean cps reads in bins of promoter and genebody reads
#-------------------------------------------------#

bin2d_cps <- aggregateByNdimBins("counts_cps", df, nbins = 20,
                                 ncores = 1)

bin2d_cps[1:6, ]

subset(bin2d_cps, is.finite(counts_cps))[1:6, ]

#-------------------------------------------------#
# get median cps reads for those bins
#-------------------------------------------------#
```

```
bin2d_cps_med <- aggregateByNdimBins("counts_cps", df, nbins = 20,
                                     FUN = median, ncores = 1)

bin2d_cps_med[1:6, ]

subset(bin2d_cps_med, is.finite(counts_cps))[1:6, ]
```

---

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,
  field = "score",
  NF = NULL,
  remove.empty = FALSE,
  blacklist = NULL,
  zero_blacklisted = FALSE,
  expand_ranges = FALSE,
  ncores = getOption("mc.cores", 2L)
)

metaSubsampleMatrix(
  counts.mat,
  binsize = 1,
  first.output.xval = 1,
  sample.name = NULL,
  n.iter = 1000,
  prop.sample = 0.1,
  lower = 0.125,
  upper = 0.875,
  NF = 1,
  remove.empty = FALSE,
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

dataset.gr	A GRanges object in which signal is contained in metadata (typically in the "score" field), or a list of such GRanges objects.
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. If length(field) > 1 and the default sample.name is left, the sample names will be inferred from the field names.
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.
field	One or more metadata fields of dataset.gr to be counted.
NF	An optional normalization factor by which to multiply the counts. If given, length(NF) must be equal to length(field).
remove.empty	A logical indicating whether regions (metaSubsample) or rows (metaSubsampleMatrix) without signal should be removed from the analysis. Not recommended if using multiple fields, as the gene lists will no longer be equivalent.
blacklist	An optional GRanges object containing regions that should be excluded from signal counting.
zero_blacklisted	When set to FALSE (the default), signal at blacklisted sites is ignored from computations. If set to TRUE, signal at blacklisted sites will be treated as equal to zero. For bootstrapping, the default behavior of ignoring signal at blacklisted sites should almost always be kept.
expand_ranges	Logical indicating if ranges in dataset.gr should be treated as descriptions of single molecules (FALSE), or if ranges should be treated as representing multiple adjacent positions with the same signal (TRUE). See getCountsByRegions.
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

A dataframe containing x-values, means, lower quantiles, upper quantiles, and the sample name (as a convenience for row-binding multiple of these dataframes). If a list of GRanges is given as input, or if multiple fields are given, a single, combined dataframe is returned containing data for all fields/datasets.

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 = 1)
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 = 1)
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 = 1)
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. For a description of the critical difference between `expand_ranges = FALSE` and `expand_ranges = TRUE`, see `getCountsByRegions`.

Usage

```
getCountsByPositions(
  dataset.gr,
  regions.gr,
  binsize = 1,
  FUN = sum,
  simplify.multi.widths = c("error", "list", "pad 0", "pad NA"),
  field = "score",
  NF = NULL,
  blacklist = NULL,
  NA_blacklisted = FALSE,
  melt = FALSE,
  expand_ranges = FALSE,
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

`dataset.gr`	A GRanges object in which signal is contained in metadata (typically in the "score" field), or a named list of such GRanges objects.
`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. By default, an error is returned. 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. If `field` not found in `names(mcols(dataset.gr))`, will default to using all fields found in `dataset.gr`.
`NF`	An optional normalization factor by which to multiply the counts. If given, `length(NF)` must be equal to `length(field)`.
`blacklist`	An optional GRanges object containing regions that should be excluded from signal counting.

NA_blacklisted — A logical indicating if NA values should be returned for blacklisted regions. By default, signal in the blacklisted sites is ignored, i.e. the reads are excluded. If NA_blacklisted = TRUE, those positions are set to NA in the final output.

melt — A logical indicating if the count matrices should be melted. If set to TRUE, a dataframe is returned in containing columns for "region", "position", and "signal". If dataset.gr is a list of multiple GRanges, or if length(field) > 1, a single dataframe is returned, which contains an additional column "sample", which contains individual sample names. If used with multi-width regions.gr, the resulting dataframe will only contain positions that are found within each respective region.

expand_ranges — Logical indicating if ranges in dataset.gr should be treated as descriptions of single molecules (FALSE), or if ranges should be treated as representing multiple adjacent positions with the same signal (TRUE). See getCountsByRegions.

ncores — Multiple cores will only be used if dataset.gr is a list of multiple datasets, or 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. If dataset.gr is a list, a parallel list is returned containing a matrix for each input dataset.

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, metaSubsample

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, ], 1)
```

`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`. If `expand_regions = FALSE`, `getCountsByRegions` is written to calculate *readcounts* overlapping each region, while `expand_regions = TRUE` will calculate "coverage signal" (see details below).

Usage

```
getCountsByRegions(
  dataset.gr,
  regions.gr,
  field = "score",
  NF = NULL,
  blacklist = NULL,
  melt = FALSE,
  region_names = NULL,
  expand_ranges = FALSE,
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

`dataset.gr`	A GRanges object in which signal is contained in metadata (typically in the "score" field), or a named list of such GRanges objects. If a list is given, a dataframe is returned containing the counts in each region for each dataset.
`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. If field not found in names(mcols(dataset.gr)), will default to using all fields found in dataset.gr.
NF	An optional normalization factor by which to multiply the counts. If given, length(NF) must be equal to length(field).
blacklist	An optional GRanges object containing regions that should be excluded from signal counting.
melt	If melt = TRUE, a dataframe is returned containing a column for regions and another column for signal. If multiple datasets are given (if dataset.gr is a list or if length(field) > 1), the output dataframe is melted to contain a third column indicating the sample names. (See section on return values below).
region_names	If melt = TRUE, an optional vector of names for the regions in regions.gr. If left as NULL, indices of regions.gr are used instead.
expand_ranges	Logical indicating if ranges in dataset.gr should be treated as descriptions of single molecules (FALSE), or if ranges should be treated as representing multiple adjacent positions with the same signal (TRUE). If the ranges in dataset.gr do not all have a width of 1, this option has a substantial effect on the results returned. (See details).
ncores	Multiple cores will only be used if dataset.gr is a list of multiple datasets, or 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. If dataset.gr is a list of multiple GRanges, or if length(field) > 1, a dataframe is returned. If melt = FALSE (the default), dataframes have a column for each dataset and a row for each region. If melt = TRUE, dataframes contain one column to indicate regions (either by their indices, or by region_names, if given), another column to indicate signal, and a third column containing the sample name (unless dataset.gr is a single GRanges object).

expand_ranges = FALSE

In this configuration, getCountsByRegions is designed to work with data in which each range represents one type of molecule, whether it's a single base (e.g. the 5' ends, 3' ends, or centers of reads) or entire reads (i.e. paired 5' and 3' ends of reads).

This is in contrast to standard run-length compressed GRanges object, as imported using rtracklayer::import.bw, in which a single range can represent multiple contiguous positions that share the same signal information.

As an example, a range of covering 10 bp with a score of 2 is treated as 2 reads (each spanning the same 10 bases), not 20 reads.

expand_ranges = TRUE

In this configuration, this function assumes that ranges in dataset.gr that cover multiple bases are compressed representations of multiple adjacent positions that contain the same signal. This type of representation is typical of "coverage" objects, including bedGraphs and bigWigs generated by many command line utilities, but *not* bigWigs as they are imported by BRGenomics::import_bigWig.

As an example, a range covering 10 bp with a score of 2 is treated as representing 20 signal counts, i.e. there are 10 adjacent positions that each contain a signal of 2.

If the data truly represents basepair-resolution coverage, the "coverage signal" is equivalent to read-counts. However, users should consider how they interpret results from whole-read coverage, as the "coverage signal" is determined by both the read counts as well as read lengths.

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 = NULL,
  gene_names = NULL,
  sizeFactors = NULL,
  field = "score",
  blacklist = NULL,
  expand_ranges = FALSE,
  ncores = getOption("mc.cores", 2L),
  quiet = FALSE
)
```

Arguments

dataset.list	An object containing GRanges datasets that can be passed to getCountsByRegions, typically a list of GRanges objects, or a multiplexed GRanges object (see details below).
regions.gr	A GRanges object containing regions of interest.
sample_names	Names for each dataset in dataset.list are required. By default (sample_names = NULL), if dataset.list is a list, the names of the list elements are used; for a multiplexed GRanges object, the field names 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. Note that DESeq2 sizeFactors are *not* the same as normalization factors defined elsewhere in this package. See details below.
field	Argument passed to getCountsByRegions. Can be used to specify fields in a single multiplexed GRanges object, or individual fields for each GRanges object in dataset.list.
blacklist	An optional GRanges object containing regions that should be excluded from signal counting. Use of this argument is distinct from the use of non-contiguous gene regions (see details below), and the two can be used simultaneously. Blacklisting doesn't affect the ranges returned as rowRanges in the output DESeqDataSet object (unlike the use of non-contiguous regions).
expand_ranges	Logical indicating if ranges in dataset.gr should be treated as descriptions of single molecules (FALSE), or if ranges should be treated as representing multiple adjacent positions with the same signal (TRUE). See getCountsByRegions.
ncores	Number of cores to use for read counting across all samples. By default, all available cores are used.
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. In contrast, 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. If the desired use is to blacklist certain sites in a genelist, note that the blacklist argument can be used.

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).

On gene names and unexpected errors

Certain gene names can cause this function to return an error. We've never encountered errors using conventional, systematic naming schemes (e.g. ensembl IDs), but we have seen errors when using Drosophila (Flybase) "symbols". We expect this is due to the unconventional use of non-alphanumeric characters in some Drosophila gene names.

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 = 1)
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,
  lfcShrink = FALSE,
  args.DESeq = NULL,
  args.results = NULL,
  args.lfcShrink = NULL,
  ncores = getOption("mc.cores", 2L),
  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 comparison, typically the perturbative condition.
`contrast.denom`	A string naming the `condition` to use as the denominator in the DESeq2 comparison, 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`, which is used for independent filtering of results (see DESeq2 documentation).

lfcShrink	Logical indicating if log2FoldChanges and their standard errors should be shrunk using lfcShrink. LFC shrinkage is very useful for making fold-change values meaningful, as low-expression/high variance genes are given low fold-changes. Set to FALSE by default.
args.DESeq	Additional arguments passed to DESeq, given as a list of argument-value pairs, e.g. list(fitType = "local",useT = TRUE). 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.
args.lfcShrink	Additional arguments passed to lfcShrink, given as a list of argument-value pairs. All arguments given here will be passed to lfcShrink except for dds, coef, contrast, 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.

Errors when ncores > 1

If this function returns an error, set ncores = 1. Whether or not this occurs can depend on whether users are using alternative BLAS libraries (e.g. OpenBLAS or Apple's Accelerate framework) and/or how DESeq2 was installed. This is because some DESeq2 functions (e.g. nbinomWaldTest) use C code that can be compiled to use parallelization, and this conflicts with our use of process forking (via the parallel package) when ncores > 1.

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 = 1)

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

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	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.signal	Logical indicating if the signal value at the max site should be reported. If set to TRUE, the values are kept as a new MaxSiteSignal metadata column in the output GRanges.
expand_ranges	Logical indicating if ranges in dataset.gr should be treated as descriptions of single molecules (FALSE), or if ranges should be treated as representing multiple adjacent positions with the same signal (TRUE). See getCountsByRegions.

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.

The output may not be the 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.

If keep.signal = TRUE, the output will also contain metadata for the signal at the max site, named MaxSiteSignal.

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

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

getMaxPositionsBySignal(PROseq, pr[1:3], binsize = 5, keep.signal = 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,
  blacklist = NULL,
  melt = FALSE,
  region_names = NULL,
  expand_ranges = FALSE,
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

`dataset.gr`	A GRanges object in which signal is contained in metadata (typically in the "score" field), or a named list of such GRanges objects.

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. If field not found in names(mcols(dataset.gr)), will default to using all fields found in dataset.gr. If dataset.gr is a list, a single field should be given, or length(field) should be the equal to the number of datasets in dataset.gr.

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. If dataset.gr is a list of datasets, or if length(field) > 1, regions are filtered unless they have promoter signal in all datasets.

blacklist An optional GRanges object containing regions that should be excluded from signal counting. If length.normalize = TRUE, blacklisted positions will be excluded from length calculations. Users should take care to note if regions of interest substantially overlap blacklisted positions.

melt If melt = TRUE, a dataframe is returned containing a column for regions and another column for pausing indices. If multiple datasets are given (if dataset.gr is a list or if length(field) > 1), the output dataframe is melted to contain a third column indicating the sample names. (See section on return values below).

region_names If melt = TRUE, an optional vector of names for the regions in regions.gr. If left as NULL, indices of regions.gr are used instead.

expand_ranges Logical indicating if ranges in dataset.gr should be treated as descriptions of single molecules (FALSE), or if ranges should be treated as representing multiple adjacent positions with the same signal (TRUE). See getCountsByRegions.

ncores Multiple cores will only be used if dataset.gr is a list of multiple datasets, or 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 dataset.gr is a list, or if length(field) > 1, a dataframe is returned, containing a column for each field. However, if melt = TRUE, dataframes contain one column to indicate regions (either by their indices, or by region_names, if given), another column to indicate signal, and a third column containing the sample name (unless dataset.gr is a single GRanges object).

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)
```

---

getSpikeInCounts *Filtering and counting spike-in reads*

Description

Filtering and counting spike-in reads

Usage

```
getSpikeInCounts(
  dataset.gr,
  si_pattern = NULL,
  si_names = NULL,
```

```
  field = "score",
  sample_names = NULL,
  expand_ranges = FALSE,
  ncores = getOption("mc.cores", 2L)
)

removeSpikeInReads(
  dataset.gr,
  si_pattern = NULL,
  si_names = NULL,
  field = "score",
  ncores = getOption("mc.cores", 2L)
)

getSpikeInReads(
  dataset.gr,
  si_pattern = NULL,
  si_names = NULL,
  field = "score",
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

dataset.gr	A GRanges object or a list of GRanges objects.
si_pattern	A regular expression that matches spike-in chromosomes. Can be used in addition to, or as an alternative to si_names.
si_names	A character vector giving the names of the spike-in chromosomes. Can be used in addition to, or as an alternative to si_pattern.
field	The metadata field in dataset.gr that contains readcounts. If each range is an individual read, set field = NULL.
sample_names	An optional character vector used to rename the datasets in dataset.gr
expand_ranges	Logical indicating if ranges in dataset.gr should be treated as descriptions of single molecules (FALSE), or if ranges should be treated as representing multiple adjacent positions with the same signal (TRUE). See getCountsByRegions.
ncores	The number of cores to use for computations.

Value

A dataframe containing total readcounts, experimental (non-spike-in) readcounts, and spike-in readcounts.

Author(s)

Mike DeBerardine

Examples

```
#------------------------------------------------#
# Make list of dummy GRanges
#------------------------------------------------#
```

```
gr1_rep1 <- GRanges(seqnames = c("chr1", "chr2", "spikechr1", "spikechr2"),
                    ranges = IRanges(start = 1:4, width = 1),
                    strand = "+")
gr2_rep2 <- gr2_rep1 <- gr1_rep2 <- gr1_rep1

# set readcounts
score(gr1_rep1) <- c(1, 1, 1, 1) # 2 exp + 2 spike = 4 total
score(gr2_rep1) <- c(2, 2, 1, 1) # 4 exp + 2 spike = 6 total
score(gr1_rep2) <- c(1, 1, 2, 1) # 2 exp + 3 spike = 5 total
score(gr2_rep2) <- c(4, 4, 2, 2) # 8 exp + 4 spike = 12 total

grl <- list(gr1_rep1, gr2_rep1,
            gr1_rep2, gr2_rep2)

names(grl) <- c("gr1_rep1", "gr2_rep1",
                "gr1_rep2", "gr2_rep2")

grl

#------------------------------------------------#
# Count spike-in reads
#------------------------------------------------#

# by giving names of all spike-in chromosomes
getSpikeInCounts(grl, si_names = c("spikechr1", "spikechr2"), ncores = 1)

# or by matching the string/regular expression "spike" in chromosome names
getSpikeInCounts(grl, si_pattern = "spike", ncores = 1)

#------------------------------------------------#
# Filter out spike-in reads
#------------------------------------------------#

removeSpikeInReads(grl, si_pattern = "spike", ncores = 1)

#------------------------------------------------#
# Return spike-in reads
#------------------------------------------------#

getSpikeInReads(grl, si_pattern = "spike", ncores = 1)
```

`getSpikeInNFs` *Calculating spike-in normalization factors*

Description

Use `getSpikeInNFs` to obtain the spike-in normalization factors, or `spikeInNormGRanges` to return the input GRanges objects with their readcounts spike-in normalized.

Usage

```
getSpikeInNFs(
  dataset.gr,
  si_pattern = NULL,
```

```
  si_names = NULL,
  method = c("SRPMC", "SNR", "RPM"),
  batch_norm = TRUE,
  ctrl_pattern = NULL,
  ctrl_names = NULL,
  field = "score",
  sample_names = NULL,
  expand_ranges = FALSE,
  ncores = getOption("mc.cores", 2L)
)

spikeInNormGRanges(
  dataset.gr,
  si_pattern = NULL,
  si_names = NULL,
  method = c("SRPMC", "SNR", "RPM"),
  batch_norm = TRUE,
  ctrl_pattern = NULL,
  ctrl_names = NULL,
  field = "score",
  sample_names = NULL,
  expand_ranges = FALSE,
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

dataset.gr	A GRanges object, or (more typically) a list of GRanges objects.
si_pattern	A regular expression that matches spike-in chromosomes. Can be used in addition to, or as an alternative to si_names.
si_names	A character vector giving the names of the spike-in chromosomes. Can be used in addition to, or as an alternative to si_pattern.
method	One of the shown methods, which generate normalization factors for converting raw readcounts into "Spike-in normalized Reads Per Million mapped in Control" (the default), "Spike-in Normalized Read counts", or "Reads Per Million mapped". See descriptions below.
batch_norm	A logical indicating if batch normalization should be used (TRUE by default). See descriptions below. If batch normalization is used, sample names must end with "rep#", wherein "#" is one or more characters (usually a number) giving the replicate. If this is not the case, users can use the sample_names argument to make the names conform.
ctrl_pattern	A regular expression that matches negative control sample names.
ctrl_names	A character vector giving the names of the negative control samples. Can be used as an alternative to ctrl_pattern.
field	The metadata field in dataset.gr that contains raw readcounts. If each range is an individual read, set field = NULL.
sample_names	An optional character vector that can be used to rename the samples in dataset.gr. Intended use is if dataset.gr is an unnamed list, or if batch_norm = TRUE but the sample names don't conform to the required naming scheme.

expand_ranges	Logical indicating if ranges in dataset.gr should be treated as descriptions of single molecules (FALSE), or if ranges should be treated as representing multiple adjacent positions with the same signal (TRUE). See getCountsByRegions.
ncores	The number of cores to use for computations.

Value

A numeric vector of normalization factors for each sample in dataset.gr. Normalization factors are to be applied by multiplication.

Spike-in normalized Reads Per Million mapped in Control (SRPMC)

This is the default spike-in normalization method, as its meaning is the most portable and generalizable. Experimental Reads Per Spike-in read (RPS) are calculated for each sample, i:

$$RPS_i = \frac{experimental\_reads_i}{spikein\_reads_i}$$

RPS for each sample is divided by RPS for the negative control, which measures the change in total material vs. the negative control. This global adjustment is applied to standard RPM normalization for each sample:

$$NF_i = \frac{RPS_i}{RPS_{control}} \cdot \frac{1x10^6}{experimental\_reads_i}$$

Thus, the negative control(s) are simply RPM-normalized, while the other conditions are in equivalent, directly-comparable units ("Reads Per Million mapped reads in a negative control").

If batch_norm = TRUE (the default), all negative controls will be RPM-normalized, and the global changes in material for all other samples are calculated *within each batch* (vs. the negative control within the same batch).

If batch_norm = FALSE, all samples are compared to the average RPS of the negative controls. This method can only be justified if batch has less effect on RPS than other sources of variation.

Spike-in Normalized Reads (SNR)

If batch_norm = FALSE, these normalization factors act to scale down the readcounts in each sample to make the spike-in read counts match the sample with the lowest number of spike-in reads:

$$NF_i = \frac{min(spikein\_reads)}{spikein\_reads_i}$$

If batch_norm = TRUE, such normalization factors are calculated within each batch, but a final batch (replicate) adjustment is performed that results in the negative controls having the same normalized readcounts. In this way, the negative controls are used to adjust the normalized readcounts of their entire replicate. Just as when batch_norm = FALSE, one of the normalization factors will be 1, while the rest will be <1.

One use for these normalization factors is for normalizing-by-subsampling; see subsampleBySpikeIn.

Reads Per Million mapped reads (RPM)

A simple convenience wrapper for calculating normalization factors for RPM normalization:

$$NF_i = \frac{1x10^6}{experimental\_reads_i}$$

If spike-in reads are present, they're removed before the normalization factors are calculated.

Author(s)

Mike DeBerardine

See Also

getSpikeInCounts, applyNFsGRanges, subsampleBySpikeIn

Examples

```
#-------------------------------------------------#
# Make list of dummy GRanges
#-------------------------------------------------#
gr1_rep1 <- GRanges(seqnames = c("chr1", "chr2", "spikechr1", "spikechr2"),
                    ranges = IRanges(start = 1:4, width = 1),
                    strand = "+")
gr2_rep2 <- gr2_rep1 <- gr1_rep2 <- gr1_rep1

# set readcounts
score(gr1_rep1) <- c(1, 1, 1, 1) # 2 exp + 2 spike = 4 total
score(gr2_rep1) <- c(2, 2, 1, 1) # 4 exp + 2 spike = 6 total
score(gr1_rep2) <- c(1, 1, 2, 1) # 2 exp + 3 spike = 5 total
score(gr2_rep2) <- c(4, 4, 2, 2) # 8 exp + 4 spike = 12 total

grl <- list(gr1_rep1, gr2_rep1,
            gr1_rep2, gr2_rep2)
names(grl) <- c("gr1_rep1", "gr2_rep1",
                "gr1_rep2", "gr2_rep2")

grl

#-------------------------------------------------#
# Get RPM NFs
#-------------------------------------------------#

# can use the names of all spike-in chromosomes
getSpikeInNFs(grl, si_names = c("spikechr1", "spikechr2"),
              method = "RPM", ncores = 1)

# or use a regular expression that matches the spike-in chromosome names
grep("spike", as.vector(seqnames(gr1_rep1)))

getSpikeInNFs(grl, si_pattern = "spike", method = "RPM", ncores = 1)


#-------------------------------------------------#
# Get simple spike-in NFs ("SNR")
#-------------------------------------------------#

# without batch normalization, NFs make all spike-in readcounts match
getSpikeInNFs(grl, si_pattern = "spike", ctrl_pattern = "gr1",
              method = "SNR", batch_norm = FALSE, ncores = 1)

# with batch normalization, controls will have the same normalized counts;
# other samples are normalized to have same spike-in reads as their matched
# control
getSpikeInNFs(grl, si_pattern = "spike", ctrl_pattern = "gr1",
```

```
              method = "SNR", batch_norm = TRUE, ncores = 1)

#-------------------------------------------------#
# Get spike-in NFs with more meaningful units ("RPMC")
#-------------------------------------------------#

# compare to raw RPM NFs above; takes into account spike-in reads;
# units are directly comparable to the negative controls

# with batch normalization, these negative controls are the same, as they
# have the same number of non-spike-in readcounts (they're simply RPM)
getSpikeInNFs(grl, si_pattern = "spike", ctrl_pattern = "gr1", ncores = 1)

# batch_norm = FALSE, the average reads-per-spike-in for the negative
# controls are used to calculate all NFs; unless the controls have the exact
# same ratio of non-spike-in to spike-in reads, nothing is precisely RPM
getSpikeInNFs(grl, si_pattern = "spike", ctrl_pattern = "gr1",
              batch_norm = FALSE, ncores = 1)

#-------------------------------------------------#
# Apply NFs to the GRanges
#-------------------------------------------------#

spikeInNormGRanges(grl, si_pattern = "spike", ctrl_pattern = "gr1",
                   ncores = 1)
```

`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",
  ncores = getOption("mc.cores", 2L)
)
```

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). `dataset.gr` can also be a list of such GRanges objects.
`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.
`ncores`	Number of cores to use for calculating coverage. For a single dataset, the max that will be used is 3, one for each possible strand (plus, minus, and unstranded). More cores can be used if `dataset.gr` is a list.

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, ncores = 1)[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, ncores = 1)[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, ncores = 1)

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], ncores = 1)
```

`import-functions` *Import basepair-resolution files*

Description

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

Usage

```
import_bigWig(
  plus_file = NULL,
  minus_file = NULL,
  genome = NULL,
  keep.X = TRUE,
  keep.Y = TRUE,
  keep.M = FALSE,
  keep.nonstandard = FALSE,
  makeBRG = TRUE,
  ncores = getOption("mc.cores", 2L)
)

import_bedGraph(
  plus_file = NULL,
  minus_file = NULL,
  genome = NULL,
  keep.X = TRUE,
  keep.Y = TRUE,
  keep.M = FALSE,
  keep.nonstandard = FALSE,
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

`plus_file, minus_file`
: Paths for strand-specific input files, or a vector of such paths. If vectors are given, the user should take care that the orders match!

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.

makeBRG — If TRUE (the default), the output ranges are made single-width using makeGRangesBRG

ncores — Number of cores to use, if importing multiple objects simultaneously.

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. For this purpose, one use bedGraphs to store entire reads, with the score representing the number of reads sharing identical 5' and 3' ends. However, import_bedGraph doesn't modify the information in the bedGraph files. If the bedGraph file represents basepair-resolution coverage data, then users can coerce it to a basepair-resolution GRanges object by using getStrandedCoverage followed by makeGRangesBRG.

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 (not supported on windows)
if (.Platform$OS.type == "unix") {
    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 single-end or paired-end bam files as GRanges objects, with various processing options. It is highly recommend to index the BAM file first.

Usage

```
import_bam(
  file,
  mapq = 20,
  revcomp = FALSE,
  shift = 0L,
  trim.to = c("whole", "5p", "3p", "center"),
  ignore.strand = FALSE,
  field = "score",
  paired_end = NULL,
  yieldSize = NA,
  ncores = 1
)

import_bam_PROseq(
  file,
  mapq = 20,
  revcomp = TRUE,
  shift = -1L,
  trim.to = "3p",
  ignore.strand = FALSE,
  field = "score",
  paired_end = NULL,
  yieldSize = NA,
  ncores = 1
)

import_bam_PROcap(
  file,
```

```
  mapq = 20,
  revcomp = FALSE,
  shift = 0L,
  trim.to = "5p",
  ignore.strand = FALSE,
  field = "score",
  paired_end = NULL,
  yieldSize = NA,
  ncores = 1
)

import_bam_ATACseq(
  file,
  mapq = 20,
  revcomp = FALSE,
  shift = c(4, -5),
  trim.to = "whole",
  ignore.strand = TRUE,
  field = "score",
  paired_end = TRUE,
  yieldSize = NA,
  ncores = 1
)
```

Arguments

file	Path of a bam file, or a vector of paths.
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.
shift	Either an integer giving the number of bases by which to shift the entire read upstream or downstream, or a pair of integers indicating shifts to be applied to the 5' and 3' ends of the reads, respectively. Shifting is strand-specific, with negative numbers shifting the reads upstream, and positive numbers shiftem them downstream. This option is applied *after* the revcomp, but before trim.to and ignore.strand options are applied.
trim.to	Option for selecting specific bases from the reads, applied after the revcomp and shift options. 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, trim.to, and shift 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.
paired_end	Logical indicating if reads should be treated as paired end reads. When set to NULL (the default), the first 100k reads are checked.
yieldSize	The number of bam file records to process simultaneously, e.g. the "chunk size". Setting a higher chunk size will use more memory, which can increase speed if there is enough memory available. If chunking is not necessary, set to NA.

ncores — Number of cores to use for importing bam files. Currently, multicore is only implemented for simultaneously importing multiple bam files. For smaller datasets or machines with higher memory, this can increase performance, but can otherwise lead to substantial performance penalties.

Value

A GRanges object.

Author(s)

Mike DeBerardine

References

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

Jason D. Buenrostro, Paul G. Giresi, Lisa C. Zaba, Howard Y. Chang, William J. Greenleaf (2013). Transposition of native chromatin for fast and sensitive epigenomic profiling of open chromatin, dna-binding proteins and nucleosome position. *Nature Methods* 10: 1213–1218. https://doi.org/10.1038/nmeth.2688

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, paired_end = FALSE)

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

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

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

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

#-------------------------------------------------#
# Import PRO-seq reads, removing the run-on base
#-------------------------------------------------#

# the best way to import PRO-seq data; removes the
```

```
# most 3' base, which was added in the run-on
import_bam(ps.bam, revcomp = TRUE, trim.to = "3p",
           shift = -1, paired_end = FALSE)

#-------------------------------------------------#
# 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",
           paired_end = FALSE)
```

`intersectByGene` *Intersect or reduce ranges according to gene names*

Description

These functions divide up regions of interest according to associated names, and perform an inter-range operation on them. `intersectByGene` returns the "consensus" segment that is common to all input ranges, and returns no more than one range per gene. `reduceByGene` collapses the input ranges into one or more non-overlapping ranges that encompass all segments from the input ranges.

Usage

```
intersectByGene(regions.gr, gene_names)

reduceByGene(regions.gr, gene_names, disjoin = FALSE)
```

Arguments

`regions.gr`	A GRanges object containing regions of interest. If `regions.gr` has the class `list`, `GRangesList`, or `CompressedGRangesList`, it will be treated as if each list element is a gene, and the GRanges within are the ranges associated with that gene.
`gene_names`	A character vector with the same length as `regions.gr`.
`disjoin`	Logical. If `disjoin = TRUE`, the output GRanges is disjoint, and each output range will match a single gene name. If `FALSE`, segments from different genes can overlap.

Details

These functions modify regions of interest that have associated names, such that several ranges share the same name, e.g. transcripts with associated gene names. Both functions "combine" the ranges on a gene-by-gene basis.

intersectByGene

For each unique gene, the segment that overlaps *all* input ranges is returned. If no single range can be constructed that overlaps all input ranges, no range is returned for that gene (i.e. the gene is effectively filtered).

In other words, for all the ranges associated with a gene, the most-downstream start site is selected, and the most upstream end site is selected.

reduceByGene

For each unique gene, the associated ranges are `reduced` to produce one or more non-overlapping ranges. The output range(s) are effectively a `union` of the input ranges, and cover every input base.

With `disjoin = FALSE`, no genomic segment is overlapped by more than one range *of the same gene*, but ranges from different genes can overlap. With `disjoin = TRUE`, the output ranges are disjoint, and no genomic position is overlapped more than once. Any segment that overlaps more than one gene is removed, but any segment (i.e. any section of an input range) that overlaps only one gene is still maintained.

Value

A GRanges object whose individual ranges are named for the associated gene.

Typical Uses

A typical use for `intersectByGene` is to avoid transcript isoform selection, as the returned range is found in every isoform.

`reduceByGene` can be used to count any and all reads that overlap any part of a gene's annotation, but without double-counting any of them. With `disjoin = FALSE`, no reads will be double-counted for the same gene, but the same read can be counted for multiple genes. With `disjoin = TRUE`, no read can be double-counted.

Author(s)

Mike DeBerardine

Examples

```
# Make example data:
#  Ranges 1 and 2 overlap,
#  Ranges 3 and 4 are adjacent
gr <- GRanges(seqnames = "chr1",
              ranges = IRanges(start = c(1, 3, 7, 10),
                               end = c(4, 5, 9, 11)))
gr

#-------------------------------------------------#
# intersectByGene
#-------------------------------------------------#

intersectByGene(gr, c("A", "A", "B", "B"))

intersectByGene(gr, c("A", "A", "B", "C"))

gr2 <- gr
end(gr2)[1] <- 10
gr2

intersectByGene(gr2, c("A", "A", "B", "C"))

intersectByGene(gr2, c("A", "A", "A", "C"))

#-------------------------------------------------#
# reduceByGene
#-------------------------------------------------#
```

```
# For a given gene, overlapping/adjacent ranges are combined;
#  gaps result in multiple ranges for that gene
gr

reduceByGene(gr, c("A", "A", "A", "A"))

# With disjoin = FALSE, ranges from different genes can overlap
gnames <- c("A", "B", "B", "B")
reduceByGene(gr, gnames)

# With disjoin = TRUE, segments overlapping >1 gene are removed as well
reduceByGene(gr, gnames, disjoin = TRUE)

# Will use one more example to demonstrate how all
#  unambiguous segments are identified and returned
gr2

gnames
reduceByGene(gr2, gnames, disjoin = TRUE)

#-------------------------------------------------#
# reduceByGene, then aggregate counts by gene
#-------------------------------------------------#

# Consider if you did getCountsByRegions on the last output,
#  you can aggregate those counts according to the genes
gr2_redux <- reduceByGene(gr2, gnames, disjoin = TRUE)
counts <- c(5, 2, 3) # if these were the counts-by-regions
aggregate(counts ~ names(gr2_redux), FUN = sum)

# even more convenient if using a melted dataframe
df <- data.frame(gene = names(gr2_redux),
                 reads = counts)
aggregate(reads ~ gene, df, FUN = sum)

# can be extended to multiple samples
df <- rbind(df, df)
df$sample <- rep(c("s1", "s2"), each = 3)
df$reads[4:6] <- c(3, 1, 2)
df

aggregate(reads ~ sample*gene, df, FUN = sum)
```

---

makeGRangesBRG *Constructing and checking for base-pair resolution GRanges objects*

Description

makeGRangesBRG 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. isBRG checks whether an object is a basepair resolution GRanges object.

Usage

```
makeGRangesBRG(dataset.gr, ncores = getOption("mc.cores", 2L))

isBRG(x)
```

Arguments

dataset.gr	A disjoint GRanges object, or a list of such objects.
ncores	If dataset.gr is a list, the number of cores to use for computations.
x	Object to be tested.

Details

Note that makeGRangesBRG 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

makeGRangesBRG returns a GRanges object for which length(output) == sum(width(dataset.gr)), and for which all(width(output) == 1).

isBRG(x) returns TRUE if x is a GRanges object with the above characteristics.

Motivation

The motivating case for this function is a bigWig file (e.g. one imported by rtracklayer), as bigWig 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

```
if (.Platform$OS.type == "unix") {

    #---------------------------------------------------#
    # 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)

    isBRG(gr)
    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, ncores = 1)

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

}
```

---

mergeGRangesData *Merge GRanges objects*

Description

Merges 2 or more GRanges objects by combining all of their ranges and associated signal (e.g. readcounts). If `multiplex = TRUE`, the input datasets are reversibly combined into a multiplexed GRanges containing a field for each input dataset.

Usage

```
mergeGRangesData(
  ...,
  field = "score",
  multiplex = FALSE,
  makeBRG = TRUE,
  exact_overlaps = FALSE,
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

...	Any number of GRanges objects in which signal (e.g. readcounts) are contained within metadata. Lists of GRanges can also be passed, but they must be named lists if multiplex = TRUE. Multiple lists can be passed, but if any inputs are lists, then all inputs must be lists.
field	One or more *input* metadata fields to be combined, typically the "score" field. Fields typically contain coverage information. If only a single field is given (i.e. all input GRanges use the same field), that same field will be used for the output. Otherwise, the "score" metadata field will be used by default. The output metadata fields are different if multiplex is enabled.
multiplex	When set to FALSE (the default), input GRanges are merged irreversibly into a single new GRange, effectively combining the reads from different experiments. When multiplex = TRUE, the input GRanges data are reversibly combined into a multiplexed GRanges object, such that each input GRanges object has its own metadata field in the output.
makeBRG	If TRUE (the default), the output GRanges will be made "basepair-resolution" via makeGRangesBRG. This is always set to codeFALSE if exact_overlaps = TRUE.
exact_overlaps	By default (FALSE), any ranges that cover more than a single base are treated as "coverage" signal (see getCountsByRegions). If exact_overlaps = TRUE, all input ranges are conserved exactly as they are; ranges are only combined if they overlap exactly, and the signal of any combined range is the sum of the input ranges that were merged.
ncores	Number of cores to use for computations.

Value

A disjoint, basepair-resolution (single-width) GRanges object comprised of all ranges found in the input GRanges objects.

If multiplex = FALSE, single fields from each input are combined into a single field in the output, the total signal of which is the sum of all input GRanges.

If multiplex = TRUE, each field of the output corresponds to an input GRanges object.

Subsetting a multiplexed GRanges object

If multiplex = TRUE, the datasets are only combined into a single object, but the data themselves are not combined. To subset field_i, corresponding to input dataset_i:

multi.gr <-mergeGRangesData(gr1,gr2,multiplex = TRUE) subset(multi.gr,gr1 != 0,select = gr1) # select gr1

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, ncores = 1))
length(mergeGRangesData(ps_1, ps_2, ps_3, ncores = 1))

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

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

PROseq[10:15]

mergeGRangesData(gr1, gr2, ncores = 1)

#-------------------------------------------------#
# multiplex separate PRO-seq experiments
#-------------------------------------------------#

multi.gr <- mergeGRangesData(gr1, gr2, multiplex = TRUE, ncores = 1)
multi.gr

#-------------------------------------------------#
# subset a multiplexed GRanges object
#-------------------------------------------------#

subset(multi.gr, gr1 > 0)

subset(multi.gr, gr1 > 0, select = gr1)
```

`mergeReplicates` *Merge replicates of basepair-resolution GRanges objects*

Description

This simple convenience function uses `mergeGRangesData` to combine replicates (e.g. biological replicates) of basepair-resolution GRanges objects.

Usage

```
mergeReplicates(
  ...,
  field = "score",
  sample_names = NULL,
  makeBRG = TRUE,
  exact_overlaps = FALSE,
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

`...`	Either a list of GRanges objects, or any number of GRanges objects (see `mergeGRangesData`). However, the names of the datasets must end in `"_rep#"`, where "#" is one or more characters indicating the replicate.
`field`	The metadata field that contains count information for each range. `length(field)` should either be 1, or equal to the number of datasets.
`sample_names`	Optional character vector with which to rename the datasets. This is useful if the sample names do not conform to the "_rep" naming scheme.
`makeBRG, exact_overlaps`	See `mergeGRangesData`.
`ncores`	The number of cores to use. This function will try to maximize the use of the `ncores` given, but care should be taken as `mergeGRangesData` can be memory intensive. Excessive memory usage can cause dramatic reductions in performance.

Value

A list of GRanges objects.

Examples

```
data("PROseq")
ps_list <- list(a_rep1 = PROseq[seq(1, length(PROseq), 4)],
                b_rep1 = PROseq[seq(2, length(PROseq), 4)],
                a_rep2 = PROseq[seq(3, length(PROseq), 4)],
                b_rep2 = PROseq[seq(4, length(PROseq), 4)])
mergeReplicates(ps_list, ncores = 1)
```

PROseq-data *PRO-seq data from Drosophila S2 cells*

Description

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

Usage

```
data(PROseq)

data(PROseq_paired)
```

Format

`PROseq` is a disjoint GRanges object with 47380 ranges and 1 metadata column, "score", which contains coverage of PRO-seq read 3' ends.

`PROseq_paired` is a GRanges object containing 53179 ranges and 1 metadata column, "score", which indicates the number of identically-mapped reads (i.e. they share the same 5' and 3' ends).

An object of class `GRanges` of length 53179.

Source

GEO Accession GSM1032758, run SRR611828.

References

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

subsampleBySpikeIn *Randomly subsample reads according to spike-in normalization*

Description

Randomly subsample reads according to spike-in normalization

Usage

```
subsampleBySpikeIn(
  dataset.gr,
  si_pattern = NULL,
  si_names = NULL,
  ctrl_pattern = NULL,
  ctrl_names = NULL,
  batch_norm = TRUE,
  RPM_units = FALSE,
  field = "score",
```

```
  sample_names = NULL,
  expand_ranges = FALSE,
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

dataset.gr, si_pattern, si_names, ctrl_pattern, ctrl_names, batch_norm, field, sample_names, expand_
: See getSpikeInNFs

RPM_units
: If set to TRUE, the final readcount values will be converted to units equivalent to/directly comparable with RPM for the negative control(s). If field = NULL, the GRanges objects will be converted to disjoint "basepair-resolution" GRanges objects, with normalized readcounts contained in the "score" metadata column.

Details

Note that if field = NULL,

Value

An object parallel to dataset.gr, but with fewer reads. E.g. if dataset.gr is a list of GRanges, the output is a list of the same GRanges, but in which each GRanges has fewer reads.

Author(s)

Mike DeBerardine

See Also

getSpikeInCounts, getSpikeInNFs

Examples

```
#-------------------------------------------------#
# Make list of dummy GRanges
#-------------------------------------------------#
gr1_rep1 <- GRanges(seqnames = c("chr1", "chr2", "spikechr1", "spikechr2"),
                    ranges = IRanges(start = 1:4, width = 1),
                    strand = "+")
gr2_rep2 <- gr2_rep1 <- gr1_rep2 <- gr1_rep1

# set readcounts
score(gr1_rep1) <- c(1, 1, 1, 1) # 2 exp + 2 spike = 4 total
score(gr2_rep1) <- c(2, 2, 1, 1) # 4 exp + 2 spike = 6 total
score(gr1_rep2) <- c(1, 1, 2, 1) # 2 exp + 3 spike = 5 total
score(gr2_rep2) <- c(4, 4, 2, 2) # 8 exp + 4 spike = 12 total

grl <- list(gr1_rep1, gr2_rep1,
            gr1_rep2, gr2_rep2)
names(grl) <- c("gr1_rep1", "gr2_rep1",
                "gr1_rep2", "gr2_rep2")

grl
```

```
#-------------------------------------------------#
# (The simple spike-in NFs)
#-------------------------------------------------#

# see examples for getSpikeInNFs for more
getSpikeInNFs(grl, si_pattern = "spike", ctrl_pattern = "gr1",
              method = "SNR", ncores = 1)

#-------------------------------------------------#
# Subsample the GRanges according to the spike-in NFs
#-------------------------------------------------#

ss <- subsampleBySpikeIn(grl, si_pattern = "spike", ctrl_pattern = "gr1",
                         ncores = 1)
ss

lapply(ss, function(x) sum(score(x))) # total reads in each

# Put in units of RPM for the negative control
ssr <- subsampleBySpikeIn(grl, si_pattern = "spike", ctrl_pattern = "gr1",
                          RPM_units = TRUE, ncores = 1)

ssr

lapply(ssr, function(x) sum(score(x))) # total signal in each
```

`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",
  expand_ranges = FALSE,
  ncores = getOption("mc.cores", 2L)
)
```

Arguments

`dataset.gr`	A GRanges object in which signal (e.g. readcounts) are contained within metadata, or a list of such GRanges objects.
`n, prop`	Either the number of reads to subsample (`n`), or the proportion of total *signal* to subsample (`prop`). Either `n` or `prop` can be given, but not both. If `dataset.gr` is a list, or if `length(field) > 1`, users can supply a vector or list of `n` or `prop`

values to match the individual datasets, but care should be taken to ensure that a value is given for each and every dataset.

`field` The metadata field of `dataset.gr` that contains readcounts for reach position. If each range represents a single read, set `field = NULL`. If multiple fields are given, and `dataset.gr` is not a list, then `dataset.gr` will be treated as a multiplexed GRanges, and each field will be treated as an indpendent dataset. See `mergeGRangesData`.

`expand_ranges` Logical indicating if ranges in `dataset.gr` should be treated as descriptions of single molecules (`FALSE`), or if ranges should be treated as representing multiple adjacent positions with the same signal (`TRUE`). See `getCountsByRegions`.

`ncores` Number of cores to use for computations. Multicore only used when `dataset.gr` is a list, or if `length(field) > 1`.

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 lowest signal value 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,
  keep.signal = FALSE,
  expand_ranges = 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.
`keep.signal`	Logical indicating if signal counts should be kept. If set to `TRUE`, the signal for each range (length-normalized if `density = TRUE`) are kept as a new `Signal` metadata column in the output GRanges object.
`expand_ranges`	Logical indicating if ranges in `dataset.gr` should be treated as descriptions of single molecules (`FALSE`), or if ranges should be treated as representing multiple adjacent positions with the same signal (`TRUE`). See `getCountsByRegions`.

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, and keep scores
#-------------------------------------------------#

subsetRegionsBySignal(txs_dm6_chr4, PROseq, quantiles = c(0.05, 0.95),
                      keep.signal = TRUE)
```

---

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,
  genome = NULL
)
```

Arguments

gr	Any GRanges object, or any another object with associated seqinfo (or a Seqinfo object itself). The object should typically have a standard genome associated with it, e.g. genome(gr) <-"hg38". gr can also be a list of such GRanges objects.
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.
genome	An optional string that, if supplied, will be used to set the genome of gr.

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