Package: BPCells (via r-universe)

May 10, 2026

Title Single Cell Counts Matrices to PCA

Version 0.3.1

Description > Efficient operations for single cell ATAC-seq fragments and RNA counts matrices. Interoperable with standard file formats, and introduces efficient bit-packed formats that allow large storage savings and increased read speeds.

License Apache-2.0 or MIT

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

Roxygen list(markdown = TRUE)

URL https://bnprks.github.io/BPCells, https://github.com/bnprks/BPCells

LinkingTo Rcpp, RcppEigen

Imports methods, grDevices, magrittr, Matrix, Rcpp, rlang, tools, vctrs, lifecycle, stringr, tibble, dplyr (>= 1.0.0), tidyr, readr, ggplot2 (>= 3.4.0), scales, patchwork, scattermore, ggrepel, RColorBrewer, hexbin

Suggests IRanges, GenomicRanges, matrixStats, igraph, RcppHNSW, RcppAnnoy

Depends R (>= 4.0.0)

Config/Needs/website pkgdown, devtools, uwot, irlba, RcppHNSW, igraph, BiocManager, bioc::BSgenome.Hsapiens.UCSC.hg38, github::GreenleafLab/motifmatchr, github::GreenleafLab/chromVARmotifs, png, magrittr

Config/pak/sysreqs libicu-dev libx11-dev

Repository https://bimsbbioinfo.r-universe.dev

Date/Publication 2025-12-15 23:30:13 UTC

RemoteUrl https://github.com/BIMSBbioinfo/BPCells

RemoteRef HEAD

RemoteSha c293a0a34e653c395c8e9bf9ed3133107324d7b7

RemoteSubdir r

Contents

`add_rows` *Broadcasting vector arithmetic*

Description

Convenience functions for adding or multiplying each row / column of a matrix by a number.

Usage

```
add_rows(mat, vec)

add_cols(mat, vec)

multiply_rows(mat, vec)

multiply_cols(mat, vec)
```

Arguments

mat	Matrix-like object
vec	Numeric vector

Value

Matrix-like object

Examples

```
set.seed(12345)
mat <- matrix(rpois(40, lambda = 5), nrow = 4)
rownames(mat) <- paste0("gene", 1:4)

mat <- mat %>% as("dgCMatrix")
mat
mat <- mat %>% as("IterableMatrix")

########################################################################
## add_rows() example
########################################################################
add_rows(mat, 1:4) %>% as("dgCMatrix")


########################################################################
## add_cols() example
########################################################################
add_cols(mat, 1:10) %>% as("dgCMatrix")


########################################################################
## multiply_rows() example
########################################################################
multiply_rows(mat, 1:4) %>% as("dgCMatrix")


########################################################################
## multiply_cols() example
########################################################################
multiply_cols(mat, 1:10) %>% as("dgCMatrix")
```

all_matrix_inputs *Get/set inputs to a matrix transform*

Description

A matrix object can either be an input (i.e. a file on disk or a raw matrix in memory), or it can represent a delayed operation on one or more matrices. The `all_matrix_inputs()` getter and setter functions allow accessing the base-level input matrices as a list, and changing them. This is useful if you want to re-locate data on disk without losing your transformed BPCells matrix. (Note: experimental API; potentially subject to revisions).

Usage

```
all_matrix_inputs(x)

all_matrix_inputs(x) <- value
```

Arguments

`x`	IterableMatrix
`value`	List of IterableMatrix objects

Value

List of IterableMatrix objects. If a matrix `m` is itself an input object, then `all_matrix_inputs(m)` will return `list(m)`.

apply_by_row *Apply a function to summarize rows/cols*

Description

Apply a custom R function to each row/col of a BPCells matrix. This will run slower than the builtin C++-backed functions, but will keep most of the memory benefits from disk-backed operations.

Usage

```
apply_by_row(mat, fun, ...)

apply_by_col(mat, fun, ...)
```

Arguments

mat	IterableMatrix object
fun	function(val, row, col) that takes in a row/col of values and returns a summary output. Argument details: 1. val - Vector length (# non-zero values) with the value for each non-zero matrix entry 2. row - one-based row index (apply_by_col: vector length (# non-zero values), apply_by_row: single integer) 3. col - one-based col index (apply_by_col: single integer, apply_by_row: vector length (# non-zero values)) 4. ... - Optional additional arguments (should not be named row, col, or val)
...	Optional additional arguments passed to fun

Details

These functions require row-major matrix storage for apply_by_row and col-major storage for apply_by_col, so matrices stored in the wrong order may neeed a re-ordered copy created using transpose_storage_order() first. This is required to be able to keep memory-usage low and allow calculating the result with a single streaming pass of the input matrix.

If vector/matrix outputs are desired instead of lists, calling unlist(x) or do.call(cbind, x) or do.call(rbind, x) can convert the list output.

Value

apply_by_row - A list of length nrow(matrix) with the results returned by fun() on each row

apply_by_col - A list of length ncol(matrix) with the results returned by fun() on each row

See Also

For an interface more similar to base::apply, see the BPCellsArray project. For calculating colMeans on a sparse single cell RNA matrix it is about 8x slower than apply_by_col, due to the base::apply interface not being sparsity-aware. (See pull request #104 for benchmarking.)

Examples

```
mat <- matrix(rbinom(40, 1, 0.5) * sample.int(5, 40, replace = TRUE), nrow = 4)
rownames(mat) <- paste0("gene", 1:4)
mat

mat <- mat %>% as("dgCMatrix") %>% as("IterableMatrix")

########################################################################
## apply_by_row() example
########################################################################
## Get mean of every row

## expect an error in the case that col-major matrix is passed
apply_by_row(mat, function(val, row, col) {sum(val) / nrow(mat)}) %>%
```

```
 unlist()

## Need to transpose matrix to make sure it is in row-order
mat_row_order <- transpose_storage_order(mat)

## works as expected for row major
apply_by_row(mat_row_order,
 function(val, row, col) sum(val) / ncol(mat_row_order)
) %>% unlist()

# Also analogous to running rowMeans() without names
rowMeans(mat)


##############################################################################
## apply_by_col() example
##############################################################################
## Get argmax of every col
apply_by_col(mat,
 function(val, row, col) if (length(val) > 0) row[which.max(val)] else 1L
) %>% unlist()
```

`binarize` *Convert matrix elements to zeros and ones*

Description

Binarize compares the matrix element values to the threshold value and sets the output elements to either zero or one. By default, element values greater than the threshold are set to one; otherwise, set to zero. When strict_inequality is set to FALSE, element values greater than or equal to the threshold are set to one. As an alternative, the <, <=, >, and >= operators are also supported.

Usage

```
binarize(mat, threshold = 0, strict_inequality = TRUE)
```

Arguments

`mat`	IterableMatrix
`threshold`	A numeric value that determines whether the elements of x are set to zero or one.
`strict_inequality`	A logical value determining whether the comparison to the threshold is >= (strict_inequality=FALSE) or > (strict_inequality=TRUE).

Value

binarized IterableMatrix object

`call_peaks_macs` *Call peaks using MACS2/3*

Description

Export pseudobulk bed files as input for MACS, then run MACS and read the output peaks as a tibble. Each step can can be run independently, allowing for quickly re-loading the results of an already completed call, or running MACS externally (e.g. via cluster job submisison) for increased parallelization. See details for more information.

Usage

```
call_peaks_macs(
  fragments,
  path,
  cell_groups = rlang::rep_along(cellNames(fragments), "all"),
  effective_genome_size = 2.9e+09,
  insertion_mode = c("both", "start_only", "end_only"),
  step = c("all", "prep-inputs", "run-macs", "read-outputs"),
  macs_executable = NULL,
  additional_params =
   "--call-summits --keep-dup all --shift -75 --extsize 150 --nomodel --nolambda",
  verbose = FALSE,
  threads = 1
)
```

Arguments

`fragments`	IterableFragments object
`path`	(string) Parent directory to store MACS inputs and outputs. Inputs are stored in `<path>/input/` and outputs in `<path>/output/<group>/`. See "File format" in details
`cell_groups`	Grouping vector with one entry per cell in fragments, e.g. cluster IDs
`effective_genome_size`	(numeric) Effective genome size for MACS. Default is `2.9e9` following MACS default for GRCh38. See deeptools for values for other common genomes.
`insertion_mode`	(string) Which fragment ends to use for coverage calculation. One of `both`, `start_only`, or `end_only`.
`step`	(string) Which step to run. One of `all`, `prep-inputs`, `run-macs`, `read-outputs`. If `prep-inputs`, create the input bed files for macs, and provides a shell script per cell group with the command to run macs. If `run-macs`, also run bash scripts to execute macs. If `read-outputs`, read the outputs into tibbles.
`macs_executable`	(string) Path to either MACS2/3 executable. Default (`NULL`) will autodetect from PATH.

additional_params
: (string) Additional parameters to pass to MACS2/3.

verbose
: (bool) Whether to provide verbose output from MACS. Only used if step is `run-macs` or `all`.

threads
: (int) Number of threads to use.

Details

File format:

- Inputs are written such that a bed file used as input into MACS, as well as a shell file containing a call to MACS are written for each cell group.
- Bed files containing `chr`, `start`, and `end` coordinates of insertions are written at `<path>/input/<group>.bed.gz`.
- Shell commands to run MACS manually are written at `<path>/input/<group>.sh`.

Outputs are written to an output directory with a subdirectory for each cell group. Each cell group's output directory contains a file for narrowPeaks, peaks, and summits.

- NarrowPeaks are written at `<path>/output/<group>/<group>_peaks.narrowPeak`.
- Peaks are written at `<path>/output/<group>/<group>_peaks.xls`.
- Summits are written at `<path>/output/<group>/<group>_summits.bed`.

Only the narrowPeaks file is read into a tibble and returned. For more information on outputs from MACS, visit the MACS docs

Performance:

Running on a 2600 cell dataset and taking both start and end insertions into account, written input bedfiles and MACS outputs used 364 MB and 158 MB of space respectively. With 4 threads, running this function end to end took 74 seconds, with 61 of those seconds spent on running MACS.

Running MACS manually:

To run MACS manually, you will first run `call_peaks_macs()` with `step="prep-inputs`. Then, manually run all of the shell scripts generated at `<path>/input/<group>.sh`. Finally, run `call_peaks_macs()` again with the same original arguments, but setting `step="read-outputs"`.

Value

- If step is `prep-inputs`, return script paths for each cell group given as a character vector.
- If step is `run-macs`, return `NULL`.
- If step is `read-outputs` or `all`, returns a tibble with all the peaks from each cell group concatenated. Columnns are `chr`, `start`, `end`, `group`, `name`, `score`, `strand`, `fold_enrichment`, `log10_pvalue`, `log10_qvalue`, `summit_offset`

Examples

```
macs_files <- file.path(tempdir(), "peaks")
frags <- get_demo_frags()

head(call_peaks_macs(frags, macs_files))
```

```
## Can also just run the input prep, then run macs manually
## by setting step to 'prep_inputs'
macs_script <- call_peaks_macs(frags, macs_files, step = "prep-inputs")
system2("bash", macs_script[1], stdout = FALSE, stderr = FALSE)

## Then read the narrow peaks files
list.files(file.path(macs_files, "output", "all"))

## call_peaks_macs() can also solely perform the output reading step
head(call_peaks_macs(frags, macs_files, step = "read-outputs"))
```

`call_peaks_tile` *Call peaks from tiles*

Description

Calling peaks from a pre-set list of tiles can be much faster than using dedicated peak-calling software like `macs3`. The resulting peaks are less precise in terms of exact coordinates, but should be sufficient for most analyses.

Usage

```
call_peaks_tile(
  fragments,
  chromosome_sizes,
  cell_groups = rep.int("all", length(cellNames(fragments))),
  effective_genome_size = NULL,
  peak_width = 200,
  peak_tiling = 3,
  fdr_cutoff = 0.01,
  merge_peaks = c("all", "group", "none")
)
```

Arguments

`fragments` IterableFragments object

`chromosome_sizes`
Chromosome start and end coordinates given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes:

- `chr`, `start`, `end`: genomic position

See `read_ucsc_chrom_sizes()`.

`cell_groups` Grouping vector with one entry per cell in fragments, e.g. cluster IDs

`effective_genome_size`
(Optional) effective genome size for poisson background rate estimation. See deeptools for values for common genomes. Defaults to sum of chromosome sizes, which overestimates peak significance

peak_width	Width of candidate peaks
peak_tiling	Number of candidate peaks overlapping each base of genome. E.g. peak_width = 300 and peak_tiling = 3 results in candidate peaks of 300bp spaced 100bp apart
fdr_cutoff	Adjusted p-value significance cutoff
merge_peaks	How to merge significant peaks with merge_peaks_iterative() • "all" Merge the full set of peaks • "group" Merge peaks within each group • "none" Don't perform any merging

Details

Peak calling steps:

1. Estimate the genome-wide expected insertions per tile based on peak_width, effective_genome_size, and per-group read counts
2. Tile the genome with nonoverlapping tiles of size peak_width
3. For each tile and group, calculate p_value based on a Poisson model
4. Compute adjusted p-values using BH method and using the total number of tiles as the number of hypotheses tested.
5. Repeat steps 2-4 peak_tiling times, with evenly spaced offsets
6. If merge_peaks is "all" or "group": use merge_peaks_iterative() within each group to keep only the most significant of the overlapping candidate peaks
7. If merge_peaks is "all", perform a final round of merge_peaks_iterative(), prioritizing each peak by its within-group significance rank

Value

tibble with peak calls and the following columns:

- chr, start, end: genome coordinates
- group: group ID that this peak was identified in
- p_val, q_val: Poission p-value and BH-corrected p-value
- enrichment: Enrichment of counts in this peak compared to a genome-wide background

Examples

```
## Prep data
reference_dir <- file.path(tempdir(), "references")
frags <- get_demo_frags()
## Remove blacklist regions from fragments
blacklist <- read_encode_blacklist(reference_dir, genome="hg38")
frags_filter_blacklist <- select_regions(frags, blacklist, invert_selection = TRUE)
chrom_sizes <- read_ucsc_chrom_sizes(reference_dir, genome="hg38") %>% dplyr::filter(chr %in% c("chr4", "chr11"))


## Call peaks
call_peaks_tile(frags_filter_blacklist, chrom_sizes, effective_genome_size = 2.8e9)
```

checksum *Calculate the MD5 checksum of an IterableMatrix*

Description

Calculate the MD5 checksum of an IterableMatrix and return the checksum in hexidecimal format.

Usage

```
checksum(matrix)
```

Arguments

matrix	IterableMatrix object

Details

`checksum()` converts the non-zero elements of the sparse input matrix to double precision, concatenates each element value with the element row and column index words, and uses these 16-byte blocks along with the matrix dimensions and row and column names to calculate the checksum. The checksum value depends on the storage order so column- and row-order matrices with the same element values give different checksum values. `checksum()` uses element and index values in little-endian CPU storage order. It converts to little-endian order on big-endian architecture although this has not been tested.

Value

MD5 checksum string in hexidecimal format.

Examples

```
library(Matrix)
library(BPCells)
m1 <- matrix(seq(1,12), nrow=3)
m2 <- as(m1, 'dgCMatrix')
m3 <- as(m2, 'IterableMatrix')
checksum(m3)
```

`cluster_cells_graph` *Cluster cell embeddings using a KNN graph-based algorithm*

Description

Take in a cell embedding matrix, then find k nearest neighbors (KNN) for each cell, convert the KNN into a graph (adjacency matrix), then run a graph-based clustering algorithm. Each of these steps can be customized by passing a function that performs the step (see details).

Usage

```
cluster_cells_graph(
  mat,
  knn_method = knn_hnsw,
  knn_to_graph_method = knn_to_geodesic_graph,
  cluster_graph_method = cluster_graph_leiden,
  threads = 0L,
  verbose = FALSE
)
```

Arguments

`mat`	(matrix) Cell embeddings matrix of shape `(cells x n_embeddings)`
`knn_method`	(function) Function to that takes an embedding matrix as the first argument and returns a k nearest neighbors (KNN) object. For example, `knn_hnsw()`, `knn_annoy()`, or a parameterized version (see Details).
`knn_to_graph_method`	(function) Function that takes a KNN object and returns a graph as an undirected graph (lower-triangular dgCMatrix adjacency matrix). For example, `knn_to_graph()`, `knn_to_snn_graph()`, `knn_to_geodesic_graph()`, or a parameterized version (see Details).
`cluster_graph_method`	(function) Function that takes an undirected graph of cell similarity and returns a factor with cluster assignments for each cell. For example, `cluster_graph_leiden()`, `cluster_graph_louvain()`, `cluster_graph_seurat()`, or a parameterized version (see Details).
`threads`	(integer) Number of threads to use in `knn_method`, `knn_to_graph_method` and `cluster_graph_method`. If these functions do not utilize a `threads` argument, this is silently ignored.
`verbose`	(logical) Whether to print progress information in `knn_method`, `knn_to_graph_method` and `cluster_graph_method`. If these functions do not utilize a `verbose` argument, this is silently ignored.

Details

Customizing clustering steps

All of the BPCells functions named like `knn_*`, `knn_to_graph_*`, and `cluster_graph_*` support customizing parameters via partial function application. For example, look for 20 neighbors during the k nearest neighbors search, setting `knn_method=knn_hnsw(k=20)` is a convenient shortcut for `knn_method=function(x) knn_hnsw(x, k=20)`. Similarly, lowering the default clustering resolution can be done as `cluster_graph_method=cluster_graph_louvain(resolution=0.5)`. This works because all these functions are written to return a partially parameterized copy of themselves as a function object when their first argument is missing.

For even more advanced customization, users can manually call the `knn`, `knn_to_graph`, and `cluster_graph` methods rather than using `cluster_cells_graph()` as a convenient wrapper.

Implementing custom clustering steps

The required interfaces for each step are as follows:

knn_method: First argument is a matrix of cell embeddings, shape `(cells x n_embeddings)`. Returns a named list of two matrices of dimension (cells x k):

- `idx`: Neighbor indices, where `idx[c, n]` is the index of the nth nearest neighbor to cell `c`.
- `dist`: Neighbor distances, where `dist[c, n]` is the distance between cell `c` and its nth nearest neighbor. Self-neighbors are allowed, so with sufficient search effort `idx[c,1] == c` for nearly all cells.

knn_to_graph_method: First argument is a KNN object as returned by `knn_method`. Returns a weighted similarity graph as a lower triangular sparse adjacency matrix (`dgCMatrix`). For cells `i` and `j`, their similarity score is in `adjacency_mat[max(i,j), min(i,j)]`.

cluster_graph_method: First argument is a weighted similarity graph as returned by `knn_to_graph_method`. Returns a factor vector of length `cells` with a cluster assignment for each cell.

Value

(factor) Factor vector containing the cluster assignment for each cell.

See Also

`knn_hnsw()` `knn_annoy()` `knn_to_graph()` `knn_to_snn_graph()` `knn_to_geodesic_graph()` `cluster_graph_leiden()` `cluster_graph_louvain()` `cluster_graph_seurat()`

`cluster_graph_leiden` *Cluster an adjacency matrix*

Description

Cluster an adjacency matrix

Usage

```
cluster_graph_leiden(
  mat,
  resolution = 1,
  objective_function = c("modularity", "CPM"),
  seed = 12531,
  ...
)

cluster_graph_louvain(mat, resolution = 1, seed = 12531)

cluster_graph_seurat(mat, resolution = 0.8, ...)
```

Arguments

`mat`	Symmetric adjacency matrix (dgCMatrix) output from e.g. `knn_to_snn_graph()` or `knn_to_geodesic_graph()`. Only the lower triangle is used.
`resolution`	Resolution parameter. Higher values result in more clusters
`objective_function`	Graph statistic to optimize during clustering. Modularity is the default as it keeps resolution independent of dataset size (see details below). For the meaning of each option, see `igraph::cluster_leiden()`.
`seed`	Random seed for clustering initialization
`...`	Additional arguments to underlying clustering function

Details

cluster_graph_leiden: Leiden clustering algorithm `igraph::cluster_leiden()`. Note that when using `objective_function = "CPM"` the number of clusters empirically scales with `cells * resolution`, so 1e-3 is a good resolution for 10k cells, but 1M cells is better with a 1e-5 resolution. A resolution of 1 is a good default when `objective_function = "modularity"` per the default.

cluster_graph_louvain: Louvain graph clustering algorithm `igraph::cluster_louvain()`

cluster_graph_seurat: Seurat's clustering algorithm `Seurat::FindClusters()`

Value

Factor vector containing the cluster assignment for each cell.

---

cluster_membership_matrix

Convert grouping vector to sparse matrix

Description

Converts a vector of membership IDs into a sparse matrix

Usage

```
cluster_membership_matrix(groups, group_order = NULL)
```

Arguments

groups	Vector with one entry per cell, specifying the cell's group
group_order	Optional vector listing ordering of groups

Value

cell x group matrix where an entry is 1 when a cell is in a given group

---

collect_features *Collect features for plotting*

Description

Helper function for data on features to plot from a diverse set of data sources.

Usage

```
collect_features(
  source,
  features = NULL,
  gene_mapping = human_gene_mapping,
  n = 1
)
```

Arguments

source	Matrix or data frame to pull features from, or a vector of feature values for a single feature. For a matrix, the features must be rows.
features	Character vector of features names to plot if source is not a vector.
gene_mapping	An optional vector for gene name matching with match_gene_symbol(). Ignored if source is a data frame.
n	Internal-use parameter marking the number of nested calls. This is used for finding the name of the "source" input variable from the caller's perspective

Details

If source is a data.frame, features will be drawn from the columns. If source is a matrix object (IterableMatrix, dgCMatrix, or matrix), features will be drawn from rows.

Value

Data frame with one column for each feature requested

convert_matrix_type *Convert the type of a matrix*

Description

Convert the type of a matrix

Usage

```
convert_matrix_type(matrix, type = c("uint32_t", "double", "float"))
```

Arguments

matrix	IterableMatrix object input
type	One of uint32_t (unsigned 32-bit integer), float (32-bit real number), or double (64-bit real number)

Value

IterableMatrix object

Examples

```
mat <- matrix(rnorm(50), nrow = 10, ncol = 5)
rownames(mat) <- paste0("gene", seq_len(10))
colnames(mat) <- paste0("cell", seq_len(5))
mat <- mat %>% as("dgCMatrix") %>% as("IterableMatrix")
mat
convert_matrix_type(mat, "float")
```

convert_to_fragments *Convert between BPCells fragments and R objects.*

Description

BPCells fragments can be interconverted with GRanges and data.frame R objects. The main conversion method is R's builtin `as()` function, though the `convert_to_fragments()` helper is also available. For all R objects except GRanges, BPCells assumes a 0-based, end-exclusive coordinate system. (See genomic-ranges-like reference for details)

Usage

```
# Convert from R to BPCells
convert_to_fragments(x, zero_based_coords = !is(x, "GRanges"))
as(x, "IterableFragments")

# Convert from BPCells to R
as.data.frame(bpcells_fragments)
as(bpcells_fragments, "data.frame")
as(bpcells_fragments, "GRanges")
```

Arguments

x	Fragment coordinates given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position • `cell_id`: cell barcodes or unique identifiers as string or factor
zero_based_coords	Whether to convert the ranges from a 1-based end-inclusive coordinate system to a 0-based end-exclusive coordinate system. Defaults to true for GRanges and false for other formats (see this archived UCSC blogpost)

Value

convert_to_fragments(): IterableFragments object

Examples

```
frags_table <- tibble::tibble(
  chr = paste0("chr", 1:10),
  start = 0,
  end = 5,
  cell_id = "cell1"
)
frags_table

frags_granges <- GenomicRanges::makeGRangesFromDataFrame(
 frags_table, keep.extra.columns = TRUE
)
frags_granges

########################################################################
## convert_to_fragments() example
########################################################################
frags <- convert_to_fragments(frags_granges)
frags


########################################################################
## as(x, "IterableFragments") example
########################################################################
```

```
frags <- as(frags_table, "IterableFragments")
frags


########################################################################
## as(bpcells_fragments, "data.frame") example
########################################################################
frags_table <- as(frags, "data.frame")
frags_table


########################################################################
## as(bpcells_fragments, "GRanges") example
########################################################################
frags_granges <- as(frags, "GRanges")
frags_granges
```

`discrete_palette` *Color palettes*

Description

These color palettes are derived from the ArchR color palettes, and provide large sets of distinguishable colors

Usage

```
discrete_palette(name, n = 1)

continuous_palette(name)
```

Arguments

`name`	Name of the color palette. Valid discrete palettes are: `stallion`, `calm`, `kelly`, `bear`, `ironMan`, `circus`, `paired`, `grove`, `summerNight`, and `captain`. Valid continuous palettes are `bluePurpleDark`
`n`	Minimum number of colors needed

Details

If the requested number of colors is too large, a new palette will be constructed via interpolation from the requested palette

Value

Character vector of hex color codes

extend_ranges *Extend genome ranges in a strand-aware fashion.*

Description

Extend genome ranges in a strand-aware fashion.

Usage

```
extend_ranges(
  ranges,
  upstream = 0,
  downstream = 0,
  metadata_cols = c("strand"),
  chromosome_sizes = NULL,
  zero_based_coords = !is(ranges, "GRanges")
)
```

Arguments

ranges	Genomic regions given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • chr, start, end: genomic position
upstream	Number of bases to extend each range upstream (negative to shrink width)
downstream	Number of bases to extend each range downstream (negative to shrink width)
metadata_cols	Optional list of metadata columns to require & extract
chromosome_sizes	(optional) Size of chromosomes as a genomic-ranges object
zero_based_coords	If true, coordinates start and 0 and the end coordinate is not included in the range. If false, coordinates start at 1 and the end coordinate is included in the range

Details

Note that ranges will be blocked from extending past the beginning of the chromosome (base 0), and if `chromosome_sizes` is given then they will also be blocked from extending past the end of the chromosome

Examples

```
## Prep data
ranges <- tibble::tibble(
 chr = "chr1",
 start = seq(50, 4050, 1000),
 end = start + 50,
```

```
 strand = "+"
)
ranges


## Extend ranges 1 bp upstream, 1 bp downstream
extend_ranges(ranges, upstream = 1, downstream = 1)
```

`footprint` *Get footprints around a set of genomic coordinates*

Description

Get footprints around a set of genomic coordinates

Usage

```
footprint(
  fragments,
  ranges,
  zero_based_coords = !is(ranges, "GRanges"),
  cell_groups = rlang::rep_along(cellNames(fragments), "all"),
  cell_weights = rlang::rep_along(cell_groups, 1),
  flank = 125L,
  normalization_width = flank%/%10L
)
```

Arguments

`fragments`	IterableFragments object
`ranges`	Footprint centers given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position • `strand`: +/- or TRUE/FALSE for positive or negative strand "+" strand ranges will footprint around the start coordinate, and "-" strand ranges around the end coordinate.
`zero_based_coords`	If true, coordinates start and 0 and the end coordinate is not included in the range. If false, coordinates start at 1 and the end coordinate is included in the range
`cell_groups`	Character or factor assigning a group to each cell, in order of `cellNames(fragments)`
`cell_weights`	Numeric vector assigning weight factors (e.g. inverse of total reads) to each cell, in order of `cellNames(fragments)`
`flank`	Number of flanking basepairs to include on either side of the motif
`normalization_width`	Number of basepairs at the upstream + downstream extremes to use for calculating enrichment

Value

`tibble::tibble()` with columns `group`, `position`, and `count`, `enrichment`

Examples

```
## Prep data
frags <- get_demo_frags()
## Motif positions taken from taking a subset of GATA1 motifs
## positions in peaks using motifmatchr
## See basic tutorial for description of generating
## positions
motif_positions <- tibble::tibble(
 chr = rep("chr4", 3),
 start = c(338237, 498344, 499851),
 end = c(338247, 498354, 499861),
 strand = c("-", "+", "+"),
 score = c(8.1422, 8.1415, 9.59462)
)

## Run footprinting
footprint(frags, motif_positions)
```

`fragments_identical` *Check if two fragments objects are identical*

Description

Check if two fragments objects are identical

Usage

```
fragments_identical(fragments1, fragments2)
```

Arguments

`fragments1`	First IterableFragments to compare
`fragments2`	Second IterableFragments to compare

Value

boolean for whether the fragments objects are identical

Examples

```
## Prep data
frags_1 <- tibble::tibble(
  chr = "chr1",
  start = seq(10, 260, 50),
  end = start + seq(5, 30, 5),
  cell_id = paste0("cell", c(rep(1, 2), rep(2,2), rep(3,2)))
) %>% convert_to_fragments()
frags_1

frags_2_identical <- tibble::tibble(
  chr = "chr1",
  start = seq(10, 260, 50),
  end = start + seq(5, 30, 5),
  cell_id = paste0("cell", c(rep(1, 2), rep(2,2), rep(3,2)))
) %>% convert_to_fragments()

frags_3_different <- tibble::tibble(
  chr = "chr1",
  start = seq(10, 260, 50),
  end = start + seq(5, 30, 5),
  cell_id = paste0("cell", c(rep(1, 2), rep(2,2), rep(4,2)))
) %>% convert_to_fragments()


## In the case of mismatching cell ids
fragments_identical(frags_1, frags_3_different)


## In the case of two identical frag objects
fragments_identical(frags_1, frags_2_identical)
```

---

gene_region *Find gene region*

Description

Conveniently look up the region of a gene by gene symbol. The value returned by this function can be used as the `region` argument for trackplot functions such as `trackplot_coverage()` or `trackplot_gene()`

Usage

```
gene_region(
  genes,
  gene_symbol,
  extend_bp = c(10000, 10000),
  gene_mapping = human_gene_mapping
)
```

Arguments

genes	Transcipt features given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position • `strand`: +/- or TRUE/FALSE for positive or negative strand • `gene_name`: Symbol or gene ID
gene_symbol	Name of gene symbol or ID
extend_bp	Bases to extend region upstream and downstream of gene. If length 1, extension is symmetric. If length 2, provide upstream extension then downstream extension as positive distances.
gene_mapping	Named vector where names are gene symbols or IDs and values are canonical gene symbols

Value

List of chr, start, end positions for use with trackplot functions.

Examples

```
## Prep data
genes <- read_gencode_transcripts(
  file.path(tempdir(), "references"), release = "42",
  annotation_set = "basic",
  features = "transcript"
)

## Get gene region
gene_region(genes, "CD19", extend_bp = 1e5)
```

---

gene_score_tiles_archr

Calculate gene-tile distances for ArchR gene activities

Description

ArchR-style gene activity scores are based on a weighted sum of each tile according to the signed distance from the tile to a gene body. This function calculates the signed distances according to ArchR's default parameters.

Usage

```
gene_score_tiles_archr(
  genes,
  chromosome_sizes = NULL,
  tile_width = 500,
  addArchRBug = FALSE
)
```

Arguments

genes	Gene coordinates given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position • `strand`: +/- or TRUE/FALSE for positive or negative strand
chromosome_sizes	(optional) Size of chromosomes as a genomic-ranges object
tile_width	Size of tiles to consider
addArchRBug	Replicate ArchR bug in handling nested genes

Details

ArchR's tile distance algorithm works as follows

1. Genes are extended 5kb upstream
2. Genes are linked to any tiles 1kb-100kb upstream + downstream, but tiles beyond a neighboring gene are not considered

Value

Tibble with one range per tile, with additional metadata columns gene_idx (row index of the gene this tile corresponds to) and distance.

Distance is a signed distance calculated such that if the tile has a smaller start coordinate than the gene and the gene is on the + strand, distance will be negative. The distance of adjacent but non-overlapping regions is 1bp, counting up from there.

Examples

```
## Prep data
directory  <- file.path(tempdir(), "references")
genes <- read_gencode_genes(
    directory,
    release = "42",
    annotation_set = "basic",
)


## Get gene scores by tile
gene_score_tiles_archr(
    genes
)
```

`gene_score_weights_archr` *Calculate GeneActivityScores*

Description

Gene activity scores can be calculated as a distance-weighted sum of per-tile accessibility. The tile weights for each gene can be represented as a sparse matrix of dimension genes x tiles. If we multiply this weight matrix by a corresponding tile matrix (tiles x cells), then we can get a gene activity score matrix of genes x cells. `gene_score_weights_archr()` calculates the weight matrix (best if you have a pre-computed tile matrix), while `gene_score_archr()` provides a easy-to-use wrapper.

Usage

```
gene_score_weights_archr(
  genes,
  chromosome_sizes,
  blacklist = NULL,
  tile_width = 500,
  gene_name_column = "gene_id",
  addArchRBug = FALSE
)

gene_score_archr(
  fragments,
  genes,
  chromosome_sizes,
  blacklist = NULL,
  tile_width = 500,
  gene_name_column = "gene_id",
  addArchRBug = FALSE,
  tile_max_count = 4,
  scale_factor = 10000,
  tile_matrix_path = tempfile(pattern = "gene_score_tile_mat")
)
```

Arguments

`genes` Gene coordinates given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes:

- `chr`, `start`, `end`: genomic position
- `strand`: +/- or TRUE/FALSE for positive or negative strand

`chromosome_sizes` Chromosome start and end coordinates given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes:

- chr, start, end: genomic position

See read_ucsc_chrom_sizes().

blacklist Regions to exclude from calculations, given as GRanges, data.frame, or list. See help("genomic-ranges-like") for details on format and coordinate systems. Required attributes:

- chr, start, end: genomic position

tile_width Size of tiles to consider

gene_name_column
If not NULL, a column name of genes to use as row names

addArchRBug Replicate ArchR bug in handling nested genes

fragments Input fragments object

tile_max_count Maximum value in the tile counts matrix. If not null, tile counts higher than this will be clipped to tile_max_count. Equivalent to ceiling argument of ArchR::addGeneScoreMatrix()

scale_factor If not null, counts for each cell will be scaled to sum to scale_factor. Equivalent to scaleTo argument of ArchR::addGeneScoreMatrix()

tile_matrix_path
Path of a directory where the intermediate tile matrix will be saved

Details

gene_score_weights_archr:

Given a set of tile coordinates and distances returned by gene_score_tiles_archr(), calculate a weight matrix of dimensions genes x tiles. This matrix can be multiplied with a tile matrix to obtain ArchR-compatible gene activity scores.

Value

gene_score_weights_archr

Weight matrix of dimension genes x tiles

gene_score_archr

Gene score matrix of dimension genes x cells.

Examples

```
## Prep data
reference_dir <- file.path(tempdir(), "references")
frags <- get_demo_frags()
genes <- read_gencode_genes(
  reference_dir,
  release="42",
  annotation_set = "basic",
) %>% dplyr::filter(chr %in% c("chr4", "chr11"))
blacklist <- read_encode_blacklist(reference_dir, genome="hg38") %>% dplyr::filter(chr %in% c("chr4", "chr11"))
chrom_sizes <- read_ucsc_chrom_sizes(reference_dir, genome="hg38") %>% dplyr::filter(chr %in% c("chr4", "chr11"))
chrom_sizes$tile_width = 500
```

```
#########################################################################
## gene_score_weights_archr() example
#########################################################################
## Get gene score weight matrix (genes x tiles)
gene_score_weights <- gene_score_weights_archr(
    genes, chrom_sizes, blacklist
)

## Get tile matrix (tiles x cells)
tiles <- tile_matrix(frags, chrom_sizes, mode = "fragments")

## Get gene scores per cell
gene_score_weights %*% tiles


#########################################################################
## gene_score_archr() example
#########################################################################
## This is a wrapper that creates both the gene score weight
## matrix and tile matrix together
gene_score_archr(frags, genes, chrom_sizes, blacklist)
```

`genomic-ranges-like` *Genomic range formats*

Description

BPCells accepts a flexible set of genomic ranges-like objects as input, either GRanges, data.frame, lists, or character vectors. These objects must specify chromosome, start, and end coordinates along with optional metadata about each range. With the exception of `GenomicRanges::GRanges` objects, BPCells assumes all objects use a zero-based, end-exclusive coordinate system (see below for details).

Valid Range-like objects:

BPCells can interpret the following types as ranges:

- `list()`, `data.frame()`, with columns:
  - `chr`: Character or factor of chromosome names
  - `start`: Start coordinates (0-based)
  - `end`: End coordinates (exclusive)
  - (optional) `strand`: `"+"/"-"` or `TRUE/FALSE` for pos/neg strand
  - (optional) Additional metadata as named `list` entries or `data.frame` columns
- `GenomicRanges::GRanges`
  - `start(x)` is interpreted as a 1-based start coordinate

  - `end(x)` is interpreted as an inclusive end coordinate
  - `strand(x)`: `"*"` entries are interpeted as postive strand
  - (optional) `mcols(x)` holds additional metadata
- `character`
  - Given in format `"chr1:1000-2000"` or `"chr1:1,000-2,000"`
  - Uses 0-based, end-exclusive coordinate system
  - Cannot be used for ranges where additional metadata is required

Range coordinate systems:

There are two main conventions for the coordinate systems:

One-based, end-inclusive ranges

- The first base of a chromosome is numbered 1
- The last base in a range is equal to the end coordinate
- e.g. 1-5 describes the first 5 bases of the chromosome
- Used in formats such as SAM, GTF
- In BPCells, used when reading or writing `GenomicRanges::GRanges` objects

Zero-based, end-exclusive ranges

- The first base of a chromosome is numbered 0
- The last base in a range is one less than the end coordinate
- e.g. 0-5 describes the first 5 bases of the chromosome
- Used in formats such as BAM, BED
- In BPCells, used for all other range objects

`get_demo_mat` *Retrieve BPCells demo data*

Description

[Experimental]

Functions to download matrices and fragments derived from a 10X Genomics PBMC 3k dataset, with options to filter with common qc metrics, and to subset genes and fragments to only chromosome 4 and 11.

Usage

```
get_demo_mat(filter_qc = TRUE, subset = TRUE)

get_demo_frags(filter_qc = TRUE, subset = TRUE)

remove_demo_data()
```

Arguments

`filter_qc`	(bool) Whether to filter both the RNA and ATAC data using qc metrics (described in `details`).
`subset`	(bool) Whether to subset to only genes/insertions on chromosome 4 and 11.

Details

These data functions are experimental. The interface, as well as the demo dataset itself will likely undergo changes in the near future.

Data Processing:

The first time either `get_demo_mat()`, or `get_demo_frags()`, are ran demo data is downloaded and stored in the BPCells data directory (under `file.path(tools::R_user_dir("BPcells", which="data"), "demo_data")`).

Subsequent calls to this function will use the previously downloaded matrix/fragments, given that the same combination of filtering and subsetting has been performed previously.

The preparation of this matrix can be reproduced by running the internal function `prepare_demo_data()` with `directory` set to the BPCells data directory.

In the case that demo data is not pre-downloaded and demo data download fails, `prepare_demo_data()` will act as a fallback.

Both the matrix from `get_demo_mat()` and the fragments from `get_demo_frags()` may be removed by running `remove_demo_data()`.

Filtering using QC information on the fragments and matrix object chooses cells with at least 1000 reads, 1000 frags, and a minimum tss enrichment of 10. Subsetting provides only genes and insertions on chromosomes 4 and 11.

Dimensions:

Condition	RNA matrix (features x cells)	Fragments (chromosomes x cells)
Raw	(36601 x 650165)	(39 x 462264)
Filter	(36601 x 2600)	(39 x 2600)
Subset	(3582 x 650165)	(2 x 462264)
Filter + Subset	(3582 x 2600)	(2 x 2600)

Data size:

Condition	RNA matrix (MB)	Fragments (MB)
Raw	31.9	200
Filter	9.4	137
Subset	18.3	25.6
Filter + Subset	1.2	12.3

Function Description:

- `get_demo_mat()`: Retrieve a demo `IterableMatrix` object representing the 10X Genomics PBMC 3k dataset.
- `get_demo_frags()`: Retrieve a demo `IterableFragments` object representing the 10X Genomics PBMC 3k dataset.
- `remove_demo_data()`: Remove the demo data from the BPCells data directory.

Value

- `get_demo_mat()`: (IterableMatrix) A `(features x cells)` matrix.
- `get_demo_frags()`: (IterableFragments) A Fragments object.
- `remove_demo_data()`: `NULL`

Examples

```
##########################################################################
## get_demo_mat() example
##########################################################################
get_demo_mat()


##########################################################################
## get_demo_frags() example
##########################################################################
get_demo_frags()


##########################################################################
## remove_demo_data() example
##########################################################################
remove_demo_data()


## Demo data folder is now empty
data_dir <- file.path(tools::R_user_dir("BPCells", which = "data"), "demo_data")
list.files(data_dir)
```

---

human_gene_mapping *Gene Symbol Mapping data*

Description

Mapping of the canonical gene symbols corresponding to each unambiguous alias, previous symbol, ensembl ID, or entrez ID.

Usage

```
human_gene_mapping

mouse_gene_mapping
```

Format

human_gene_mapping

A named character vector. Names are aliases or IDs and values are the corresponding canonical gene symbol

mouse_gene_mapping

A named character vector. Names are aliases or IDs and values are the corresponding canonical gene symbol

Details

See the source code in `data-raw/human_gene_mapping.R` and `data-raw/mouse_gene_mapping.R` for exactly how these mappings were made.

Source

human_gene_mapping

http://ftp.ebi.ac.uk/pub/databases/genenames/hgnc/tsv/non_alt_loci_set.txt

mouse_gene_mapping

http://www.informatics.jax.org/downloads/reports/MGI_EntrezGene.rpt http://www.informatics.jax.org/downloads/reports/MRK_ENSEMBL.rpt

Examples

```
########################################################################
## human_gene_mapping
head(human_gene_mapping)
########################################################################
########################################################################
## mouse_gene_mapping
head(mouse_gene_mapping)
```

`import_matrix_market` *Import MatrixMarket files*

Description

Read a sparse matrix from a MatrixMarket file. This is a text-based format used by 10x, Parse, and others to store sparse matrices. Format details on the NIST website.

Usage

```
import_matrix_market(
  mtx_path,
  outdir = tempfile("matrix_market"),
  row_names = NULL,
  col_names = NULL,
  row_major = FALSE,
  tmpdir = tempdir(),
  load_bytes = 4194304L,
  sort_bytes = 1073741824L
)

import_matrix_market_10x(
  mtx_dir,
  outdir = tempfile("matrix_market"),
  feature_type = NULL,
  row_major = FALSE,
  tmpdir = tempdir(),
  load_bytes = 4194304L,
  sort_bytes = 1073741824L
)
```

Arguments

mtx_path	Path of mtx or mtx.gz file
outdir	Directory to store the output
row_names	Character vector of row names
col_names	Character vector of col names
row_major	If true, store the matrix in row-major orientation
tmpdir	Temporary directory to use for intermediate storage
load_bytes	The minimum contiguous load size during the merge sort passes
sort_bytes	The amount of memory to allocate for re-sorting chunks of entries
mtx_dir	Directory holding matrix.mtx.gz, barcodes.tsv.gz, and features.tsv.gz
feature_type	String or vector of feature types to include. (cellranger 3.0 and newer)

Details

Import MatrixMarket mtx files to the BPCells format. This implementation ensures fixed memory usage even for very large inputs by doing on-disk sorts. It will be much slower than hdf5 inputs, so only use MatrixMarket format when absolutely necessary.

As a rough speed estimate, importing the 17GB Parse 1M PBMC `DGE_1M_PBMC.mtx` file takes about 4 minutes and 1.3GB of RAM, producing a compressed output matrix of 1.5GB. `mtx.gz` files will be slower to import due to gzip decompression.

When importing from 10x mtx files, the row and column names can be read automatically using the `import_matrix_market_10x()` convenience function.

Value

MatrixDir object with the imported matrix

`IterableFragments-methods` *IterableFragments methods*

Description

Methods for IterableFragments objects

Usage

```
## S4 method for signature 'IterableFragments'
show(object)

cellNames(x)

cellNames(x, ...) <- value

chrNames(x)

chrNames(x, ...) <- value
```

Arguments

`object`	IterableFragments object
`x`	an IterableFragments object
`value`	Character vector of new names

Details

- `cellNames<-` It is only possible to replace names, not add new names.
- `chrNames<-` It is only possible to replace names, not add new names.

Value

- `cellNames()` Character vector of cell names, or NULL if none are known
- `chrNames()`: Character vector of chromosome names, or NULL if none are known

Functions

- `show(IterableFragments)`: Print IterableFragments
- `cellNames()`: Get cell names
- `cellNames(x, ...) <- value`: Set cell names
- `chrNames()`: Set chromosome names
- `chrNames(x, ...) <- value`: Set chromosome names

Examples

```
## Prep data
frags <- tibble::tibble(
  chr = paste0("chr", c(rep(1,3), rep(2,3))),
  start = seq(10, 260, 50),
  end = start + 30,
  cell_id = paste0("cell", c(rep(1, 2), rep(2,2), rep(3,2)))
)
frags
frags <- frags %>% convert_to_fragments()


########################################################################
## show(IterableFragments) example
########################################################################
show(frags)


########################################################################
## cellNames(IterableFragments) example
########################################################################
cellNames(frags)


########################################################################
## cellNames(IterableFragments)<- example
########################################################################
cellNames(frags) <- paste0("cell", 5:7)
cellNames(frags)


########################################################################
## chrNames(IterableFragments) example
########################################################################
chrNames(frags)


########################################################################
## chrNames(IterableFragments)<- example
########################################################################
chrNames(frags) <- paste0("chr", 5:6)
chrNames(frags)
```

IterableMatrix-methods

IterableMatrix methods

Description

Generic methods and built-in functions for IterableMatrix objects

Usage

```
matrix_type(x)

storage_order(x)

## S4 method for signature 'IterableMatrix'
show(object)

## S4 method for signature 'IterableMatrix'
t(x)

## S4 method for signature 'IterableMatrix,matrix'
x %*% y

## S4 method for signature 'IterableMatrix'
rowSums(x)

## S4 method for signature 'IterableMatrix'
colSums(x)

## S4 method for signature 'IterableMatrix'
rowMeans(x)

## S4 method for signature 'IterableMatrix'
colMeans(x)

colVars(
  x,
  rows = NULL,
  cols = NULL,
  na.rm = FALSE,
  center = NULL,
  ...,
  useNames = TRUE
)

rowVars(
  x,
  rows = NULL,
  cols = NULL,
  na.rm = FALSE,
  center = NULL,
  ...,
  useNames = TRUE
```

```
)

rowMaxs(x, rows = NULL, cols = NULL, na.rm = FALSE, ..., useNames = TRUE)

colMaxs(x, rows = NULL, cols = NULL, na.rm = FALSE, ..., useNames = TRUE)

rowQuantiles(
  x,
  rows = NULL,
  cols = NULL,
  probs = seq(from = 0, to = 1, by = 0.25),
  na.rm = FALSE,
  type = 7L,
  digits = 7L,
  ...,
  useNames = TRUE,
  drop = TRUE
)

colQuantiles(
  x,
  rows = NULL,
  cols = NULL,
  probs = seq(from = 0, to = 1, by = 0.25),
  na.rm = FALSE,
  type = 7L,
  digits = 7L,
  ...,
  useNames = TRUE,
  drop = TRUE
)

## S4 method for signature 'IterableMatrix'
log1p(x)

log1p_slow(x)

## S4 method for signature 'IterableMatrix'
expm1(x)

expm1_slow(x)

## S4 method for signature 'IterableMatrix,numeric'
e1 ^ e2

## S4 method for signature 'numeric,IterableMatrix'
e1 < e2
```

```
## S4 method for signature 'IterableMatrix,numeric'
e1 > e2

## S4 method for signature 'numeric,IterableMatrix'
e1 <= e2

## S4 method for signature 'IterableMatrix,numeric'
e1 >= e2

## S4 method for signature 'IterableMatrix'
round(x, digits = 0)

## S4 method for signature 'IterableMatrix,numeric'
e1 * e2

## S4 method for signature 'IterableMatrix,numeric'
e1 + e2

## S4 method for signature 'IterableMatrix,numeric'
e1 / e2

## S4 method for signature 'IterableMatrix,numeric'
e1 - e2
```

Arguments

x	IterableMatrix object or a matrix-like object.
object	IterableMatrix object
y	matrix
probs	(Numeric) Quantile value(s) to be computed, between 0 and 1.
type	(Integer) between 4 and 9 selecting which quantile algorithm to use, detailed in `matrixStats::rowQuantiles()`

Value

- `t()` Transposed object
- `x %*% y`: dense matrix result
- `rowSums()`: vector of row sums
- `colSums()`: vector of col sums
- `rowMeans()`: vector of row means
- `colMeans()`: vector of col means
- `colVars()`: vector of col variance

- `rowVars()`: vector of row variance
- `rowMaxs()`: vector of maxes for every row
- `colMaxs()`: vector of column maxes
- `rowQuantiles()`: If `length(probs) == 1`, return a numeric with number of entries equal to the number of rows in the matrix. Else, return a Matrix of quantile values, with cols representing each quantile, and each row representing a row in the input matrix.
- `colQuantiles()`: If `length(probs) == 1`, return a numeric with number of entries equal to the number of columns in the matrix. Else, return a Matrix of quantile values, with cols representing each quantile, and each row representing a col in the input matrix.

Functions

- `matrix_type()`: Get the matrix data type (mat_uint32_t, mat_float, or mat_double for now)
- `storage_order()`: Get the matrix storage order ("row" or "col")
- `show(IterableMatrix)`: Display an IterableMatrix
- `t(IterableMatrix)`: Transpose an IterableMatrix
- `x %*% y`: Multiply by a dense matrix
- `rowSums(IterableMatrix)`: Calculate rowSums
- `colSums(IterableMatrix)`: Calculate colSums
- `rowMeans(IterableMatrix)`: Calculate rowMeans
- `colMeans(IterableMatrix)`: Calculate colMeans
- `colVars()`: Calculate colVars (replacement for `matrixStats::colVars()`)
- `rowVars()`: Calculate rowVars (replacement for `matrixStats::rowVars()`)
- `rowMaxs()`: Calculate rowMaxs (replacement for `matrixStats::rowMaxs()`)
- `colMaxs()`: Calculate colMax (replacement for `matrixStats::colMax()`)
- `rowQuantiles()`: Calculate rowQuantiles (replacement for `matrixStats::rowQuantiles`)
- `colQuantiles()`: Calculate colQuantiles (replacement for `matrixStats::colQuantiles`)
- `log1p(IterableMatrix)`: Calculate log(x + 1)
- `log1p_slow()`: Calculate log(x + 1) (non-SIMD version)
- `expm1(IterableMatrix)`: Calculate exp(x) - 1
- `expm1_slow()`: Calculate exp(x) - 1 (non-SIMD version)
- `e1^e2`: Calculate x^y (elementwise; y > 0)
- `e1 < e2`: Binarize matrix according to numeric < matrix comparison
- `e1 > e2`: Binarize matrix according to matrix > numeric comparison
- `e1 <= e2`: Binarize matrix according to numeric <= matrix comparison
- `e1 >= e2`: Binarize matrix according to matrix >= numeric comparison
- `round(IterableMatrix)`: round to nearest integer (digits must be 0)
- `e1 * e2`: Multiply by a constant, or multiply rows by a vector length nrow(mat)
- `e1 + e2`: Add a constant, or row-wise addition with a vector length nrow(mat)
- `e1 / e2`: Divide by a constant, or divide rows by a vector length nrow(mat)
- `e1 - e2`: Subtract a constant, or row-wise subtraction with a vector length nrow(mat)

Examples

```
## Prep data
mat <- matrix(1:25, nrow = 5) %>% as("dgCMatrix")
mat
mat <- as(mat, "IterableMatrix")
mat


########################################################################
## matrix_type() example
########################################################################
matrix_type(mat)


########################################################################
## storage_order() example
########################################################################
storage_order(mat)


########################################################################
## show() example
########################################################################
show(mat)


########################################################################
## t() example
########################################################################
t(mat)


########################################################################
## `x %*% y` example
########################################################################
mat %*% as(matrix(1:50, nrow = 5), "dgCMatrix")


########################################################################
## rowSums() example
########################################################################
rowSums(mat)


########################################################################
## colSums() example
########################################################################
colSums(mat)


########################################################################
## rowMeans() example
```

```
########################################################################
rowMeans(mat)


########################################################################
## colMeans() example
########################################################################
colMeans(mat)


########################################################################
## colVars() example
########################################################################
colVars(mat)


########################################################################
## rowMaxs() example
########################################################################
rowMaxs(mat)


########################################################################
## colMaxs() example
########################################################################
colMaxs(mat)


########################################################################
## rowQuantiles() example
########################################################################
rowQuantiles(transpose_storage_order(mat))


########################################################################
## colQuantiles() example
########################################################################
colQuantiles(mat)


########################################################################
## log1p() example
########################################################################
log1p(mat)


########################################################################
## log1p_slow() example
########################################################################
log1p_slow(mat)


########################################################################
```

```
## expm1() example
################################################################################
expm1(mat)


################################################################################
## expm1_slow() example
################################################################################
expm1_slow(mat)


################################################################################
## `e1 < e2` example
################################################################################
5 < mat


################################################################################
## `e1 > e2` example
################################################################################
mat > 5


################################################################################
## `e1 <= e2` example
################################################################################
5 <= mat


################################################################################
## `e1 >= e2` example
################################################################################
mat >= 5


################################################################################
## round() example
################################################################################
round(mat)


################################################################################
## `e1 * e2` example
################################################################################
## Multiplying by a constant
mat * 5

## Multiplying by a vector of length `nrow(mat)`
mat * 1:nrow(mat)


################################################################################
## `e1 + e2` example
```

```
##------------------------------------------------------------------------
## Add by a constant
mat + 5

## Adding row-wise by a vector of length `nrow(mat)`
mat + 1:nrow(mat)


##------------------------------------------------------------------------
## `e1 / e2` example
##------------------------------------------------------------------------
## Divide by a constant
mat / 5

## Divide by a vector of length `nrow(mat)`
mat / 1:nrow(mat)


##------------------------------------------------------------------------
## `e1 - e2` example
##------------------------------------------------------------------------
## Subtracting by a constant
mat - 5

## Subtracting by a vector of length `nrow(mat)`
mat - 1:nrow(mat)
```

---

knn_hnsw *Get a knn object from reduced dimensions*

Description

Search for approximate nearest neighbors between cells in the reduced dimensions (e.g. PCA), and return the k nearest neighbors (knn) for each cell. Optionally, we can find neighbors between two separate sets of cells by utilizing both data and query.

Usage

```
knn_hnsw(
  data,
  query = NULL,
  k = 10,
  metric = c("euclidean", "cosine"),
  verbose = TRUE,
  threads = 1,
  ef = 100
)
```

```
knn_annoy(
  data,
  query = NULL,
  k = 10,
  metric = c("euclidean", "cosine", "manhattan", "hamming"),
  n_trees = 50,
  search_k = -1
)
```

Arguments

data	cell x dims matrix for reference dataset
query	cell x dims matrix for query dataset (optional)
k	number of neighbors to calculate
metric	distance metric to use
verbose	whether to print progress information during search
threads	Number of threads to use. Note that result is non-deterministic if threads > 1
ef	ef parameter for `RcppHNSW::hnsw_search()`. Increase for slower search but improved accuracy
n_trees	Number of trees during index build time. More trees gives higher accuracy
search_k	Number of nodes to inspect during the query, or -1 for default value. Higher number gives higher accuracy

Details

knn_hnsw: Use RcppHNSW as knn engine

knn_annoy: Use RcppAnnoy as knn engine

Value

Named list of two matrices of dimension (cells x k):

- `idx`: Neighbor indices, where `idx[c, n]` is the index of the nth nearest neighbor to cell `c`.
- `dist`: Neighbor distances, where `dist[c, n]` is the distance between cell `c` and its nth nearest neighbor.

If no query is given, nearest neighbors are found by mapping the data matrix to itself, likely including self-neighbors (i.e. `idx[c,1] == c` for most cells).

knn_to_graph *K Nearest Neighbor (KNN) Graph*

Description

Convert a KNN object (e.g. returned by `knn_hnsw()` or `knn_annoy()`) into a graph. The graph is represented as a sparse adjacency matrix.

Usage

```
knn_to_graph(knn, use_weights = FALSE, self_loops = TRUE)

knn_to_snn_graph(
  knn,
  min_val = 1/15,
  self_loops = FALSE,
  return_type = c("matrix", "list")
)

knn_to_geodesic_graph(knn, return_type = c("matrix", "list"), threads = 0L)
```

Arguments

`knn`	List of 2 matrices – idx for cell x K neighbor indices, dist for cell x K neighbor distances
`use_weights`	boolean for whether to replace all distance weights with 1
`self_loops`	Whether to allow self-loops in the output graph
`min_val`	minimum jaccard index between neighbors. Values below this will round to 0
`return_type`	Whether to return a sparse adjacency matrix or an edge list
`threads`	Number of threads to use during calculations

Details

knn_to_graph Create a knn graph

knn_to_snn_graph Convert a knn object into a shared nearest neighbors adjacency matrix. This follows the algorithm that Seurat uses to compute SNN graphs

knn_to_geodesic_graph Convert a knn object into an undirected weighted graph, using the same geodesic distance estimation method as the UMAP package. This matches the output of `umap._umap.fuzzy_simplicial_se` from the `umap-learn` python package, used by default in `scanpy.pp.neighbors`. Because this only re-weights and symmetrizes the KNN graph, it will usually use less memory and return a sparser graph than `knn_to_snn_graph` which computes 2nd-order neighbors. Note: when cells don't have themselves listed as the nearest neighbor, results may differ slightly from `umap._umap.fuzzy_simplicial_set`, which assumes self is always successfully found in the approximate nearest neighbor search.

Value

knn_to_graph Sparse matrix (dgCMatrix) where `mat[i,j]` = distance from cell `i` to cell `j`, or 0 if cell `j` is not in the K nearest neighbors of `i`

knn_to_snn_graph

- `return_type == "matrix"`: Sparse matrix (dgCMatrix) where `mat[i,j]` = jaccard index of the overlap in nearest neigbors between cell `i` and cell `j`, or 0 if the jaccard index is < `min_val`. Only the lower triangle is filled in, which is compatible with the BPCells clustering methods
- `return_type == "list"`: List of 3 equal-length vectors `i`, `j`, and `weight`, along with an integer `dim`. These correspond to the rows, cols, and values of non-zero entries in the lower triangle adjacency matrix. `dim` is the total number of vertices (cells) in the graph

knn_to_geodesic_graph

- `return_type == "matrix"`: Sparse matrix (dgCMatrix) where `mat[i,j]` = normalized similarity between cell `i` and cell `j`. Only the lower triangle is filled in, which is compatible with the BPCells clustering methods
- `return_type == "list"`: List of 3 equal-length vectors `i`, `j`, and `weight`, along with an integer `dim`. These correspond to the rows, cols, and values of non-zero entries in the lower triangle adjacency matrix. `dim` is the total number of vertices (cells) in the graph

---

marker_features *Test for marker features*

Description

Given a features x cells matrix, perform one-vs-all differential tests to find markers.

Usage

```
marker_features(mat, groups, method = "wilcoxon")
```

Arguments

`mat`	IterableMatrix object of dimensions features x cells
`groups`	Character/factor vector of cell groups/clusters. Length #cells
`method`	Test method to use. Current options are: • `wilcoxon`: Wilconxon rank-sum test a.k.a Mann-Whitney U test

Details

Tips for using the values from this function:

- Use `dplyr::mutate()` to add columns for e.g. adjusted p-value and log fold change.
- Use `dplyr::filter()` to get only differential genes above some given threshold
- To get adjusted p-values, use R `p.adjust()`, recommended method is "BH"
- To get log2 fold change: if your input matrix was already log-transformed, calculate `(foreground_mean - background_mean)/log(2)`. If your input matrix was not log-transformed, calculate `log2(forground_mean/backgr`

Value

tibble with the following columns:

- **foreground**: Group ID used for the foreground
- **background**: Group ID used for the background (or NA if comparing to rest of cells)
- **feature**: ID of the feature
- **p_val_raw**: Unadjusted p-value for differential test
- **foreground_mean**: Average value in the foreground group
- **background_mean**: Average value in the background group

Examples

```
mat <- get_demo_mat()
groups <- sample(c("A", "B", "C", "D"), ncol(mat), replace = TRUE)
marker_feats <- marker_features(mat, groups)

## to see the results of one specific group vs all other groups
marker_feats %>% dplyr::filter(foreground == "A")

## get only differential genes given a threshold value
marker_feats %>% dplyr::filter(p_val_raw < 0.05)
```

---

match_gene_symbol *Gene symbol matching*

Description

Correct alias gene symbols, Ensembl IDs, and Entrez IDs to canonical gene symbols. This is useful for matching gene names between different datasets which might not always use the same gene naming conventions.

Usage

```
match_gene_symbol(query, subject, gene_mapping = human_gene_mapping)

canonical_gene_symbol(query, gene_mapping = human_gene_mapping)
```

Arguments

query	Character vector of gene symbols or IDs
subject	Vector of gene symbols or IDs to index into
gene_mapping	Named vector where names are gene symbols or IDs and values are canonical gene symbols

Value

match_gene_symbol

Integer vector of indices v such that `subject[v]` corresponds to the gene symbols in `query`

canonical_gene_symbol

Character vector of canonical gene symbols for each symbol in `query`

Examples

```
########################################################################
## match_gene_symbol() example
########################################################################
match_gene_symbol(
 c("CD8", "CD4", "CD45"),
 c("ENSG00000081237.19", "ENSG00000153563.15", "ENSG00000010610.9", "ENSG00000288825")
)


########################################################################
## canonical_gene_symbol() example
########################################################################
canonical_gene_symbol(c("CD45", "CD8", "CD4"))
```

`matrix_R_conversion` *Convert between BPCells matrix and R objects.*

Description

BPCells matrices can be interconverted with Matrix package dgCMatrix sparse matrices, as well as base R dense matrices (though this may result in high memory usage for large matrices)

Usage

```
# Convert to R from BPCells
as(bpcells_mat, "dgCMatrix") # Sparse matrix conversion
as.matrix(bpcells_mat) # Dense matrix conversion

# Convert to BPCells from R
as(dgc_mat, "IterableMatrix")
```

Examples

```
mat <- get_demo_mat()[1:2, 1:2]
mat
```

```
########################################################################
## as(bpcells_mat, "dgCMatrix") example
########################################################################
mat_dgc <- as(mat, "dgCMatrix")
mat_dgc


## as.matrix(bpcells_mat) example
as.matrix(mat)

## Alternatively, can also use function as()
as(mat, "matrix")

########################################################################
## as(dgc_mat, "IterableMatrix") example
########################################################################
as(mat_dgc, "IterableMatrix")
```

`matrix_stats` *Calculate matrix stats*

Description

Calculate matrix stats

Usage

```
matrix_stats(
  matrix,
  row_stats = c("none", "nonzero", "mean", "variance"),
  col_stats = c("none", "nonzero", "mean", "variance"),
  threads = 0L
)
```

Arguments

`matrix`	Input matrix object
`row_stats`	Which row statistics to compute
`col_stats`	Which col statistics to compute
`threads`	Number of threads to use during execution

Details

The statistics will be calculated in a single pass over the matrix, so this method is desirable to use for efficiency purposes compared to the more standard rowMeans or colMeans if multiple statistics are needed. The stats are ordered by complexity: nonzero, mean, then variance. All less complex

stats are calculated in the process of calculating a more complicated stat. So to calculate mean and variance simultaneously, just ask for variance, which will compute mean and nonzero counts as a side-effect

Value

List of row_stats: matrix of n_stats x n_rows, col_stats: matrix of n_stats x n_cols

Examples

```
mat <- matrix(rpois(100, lambda = 5), nrow = 10)
rownames(mat) <- paste0("gene", 1:10)
colnames(mat) <- paste0("cell", 1:10)
mat <- mat %>% as("dgCMatrix") %>% as("IterableMatrix")

## By default, no row or column stats are calculated
res_none <- matrix_stats(mat)
res_none

## Request row variance (automatically computes mean and nonzero too)
res_row_var <- matrix_stats(mat, row_stats = "variance")
res_row_var

## Request both row variance and column variance
res_both_var <- matrix_stats(
  mat = mat,
  row_stats = "variance",
  col_stats = "mean"
)
res_both_var
```

---

merge_cells *Merge cells into pseudobulks*

Description

Peak and tile matrix calculations can be sped up by reducing the number of cells. For cases where the outputs are going to be added together afterwards, this can provide a performance improvement

Usage

```
merge_cells(fragments, cell_groups)
```

Arguments

fragments	Input fragments object
cell_groups	Character or factor vector providing a group for each cell. Ordering is the same as `cellNames(fragments)`

Examples

```
frags <- tibble::tibble(
  chr = "chr1",
  start = seq(10, 260, 50),
  end = start + 30,
  cell_id = paste0("cell", c(rep(1, 2), rep(2,2), rep(3,2)))
) %>% convert_to_fragments()
frags


## Pseudobulk into two groups
merge_cells(frags, as.factor(c(rep(1,3), rep(2,3))))
```

`merge_peaks_iterative` *Merge peaks*

Description

Merge peaks according to ArchR's iterative merging algorithm. More details on the ArchR website

Usage

```
merge_peaks_iterative(peaks)
```

Arguments

`peaks`	Peaks given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position Must be ordered by priority and have columns chr, start, end.

Details

Properties of merged peaks:

- No peaks in the merged set overlap
- Peaks are prioritized according to their order in the original input
- The output peaks are a subset of the input peaks, with no peak boundaries changed

Value

`tibble::tibble()` with a nonoverlapping subset of the rows in peaks. All metadata columns are preserved

Examples

```
## Create example peaks
peaks <- tibble::tibble(
 chr = "chr1",
 start = as.integer(1:10),
 end = start + 2L
)
peaks

## Merge peaks
merge_peaks_iterative(peaks)
```

`min_scalar` *Elementwise minimum*

Description

min_scalar: Take minumum with a global constant

min_by_row: Take the minimum with a per-row constant

min_by_col: Take the minimum with a per-col constant

Usage

```
min_scalar(mat, val)

min_by_row(mat, vals)

min_by_col(mat, vals)
```

Arguments

`mat`	IterableMatrix
`val`	Single positive numeric value

Details

Take the minimum value of a matrix with a per-row, per-col, or global constant. This constant must be >0 to preserve sparsity of the matrix. This has the effect of capping the maximum value in the matrix.

Value

IterableMatrix

Examples

```
set.seed(12345)
mat <- matrix(rpois(40, lambda = 5), nrow = 4)
rownames(mat) <- paste0("gene", 1:4)

mat <- mat %>% as("dgCMatrix")
mat
mat <- mat %>% as("IterableMatrix")

#############################################################################
## min_scalar() example
#############################################################################
min_scalar(mat, 4) %>% as("dgCMatrix")


#############################################################################
## min_by_row() example
#############################################################################
min_by_row(mat, 1:4) %>% as("dgCMatrix")


#############################################################################
## min_by_col() example
#############################################################################
min_by_col(mat, 1:10) %>% as("dgCMatrix")
```

`normalize_ranges` *Normalize an object representing genomic ranges*

Description

Normalize an object representing genomic ranges

Usage

```
normalize_ranges(
  ranges,
  metadata_cols = character(0),
  zero_based_coords = !is(ranges, "GRanges"),
  n = 1
)
```

Arguments

`ranges`	Genomic regions given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position

`metadata_cols` Optional list of metadata columns to require & extract

`zero_based_coords`
: If true, coordinates start and 0 and the end coordinate is not included in the range. If false, coordinates start at 1 and the end coordinate is included in the range

Value

data frame with zero-based coordinates, and elements chr (factor), start (int), and end (int). If `ranges` does not have chr level information, chr levels are the sorted unique values of chr.

If strand is in metadata_cols, then the output strand element will be TRUE for positive strand, and FALSE for negative strand. (Converted from a character vector of "+"/"-" if necessary)

Examples

```
## Prep data
ranges <- GenomicRanges::GRanges(
  seqnames = S4Vectors::Rle(c("chr1", "chr2", "chr3"), c(1, 2, 2)),
  ranges = IRanges::IRanges(101:105, end = 111:115, names = head(letters, 5)),
  strand = S4Vectors::Rle(GenomicRanges::strand(c("-", "+", "*")), c(1, 2, 2)),
  score = 1:5,
  GC = seq(1, 0, length=5))
ranges


## Normalize ranges
normalize_ranges(ranges)


## With metadata information
normalize_ranges(ranges, metadata_cols = c("strand", "score", "GC"))
```

---

nucleosome_counts *Count fragments by nucleosomal size*

Description

Count fragments by nucleosomal size

Usage

```
nucleosome_counts(fragments, nucleosome_width = 147)
```

Arguments

`fragments` Fragments object

`nucleosome_width`
: Integer cutoff to use as nucleosome width

Details

Shorter than nucleosome_width is subNucleosomal, nucleosome_width to 2*nucleosome_width-1 is monoNucleosomal, and anything longer is multiNucleosomal. The sum of all fragments is given as nFrags

Value

List with names subNucleosomal, monoNucleosomal, multiNucleosomal, and nFrags, containing the count vectors of fragments in each class per cell.

Examples

```
## Prep data
frags_sub_nucleosomal <- tibble::tibble(
  chr = 1,
  start = seq(0, 3000, by = 1000),
  end = start + 146,
  cell_id = c(rep("cell1", 3), rep("cell2", 1))
)
frags_sub_nucleosomal

frags_nucleosomal <- tibble::tibble(
  chr = 1,
  start = seq(5000, 7000, by = 1000),
  end = start + 147, # Value equal to nucleosome_width is inclusive
  cell_id = c(rep("cell1", 1), rep("cell2", 2))
)
frags_nucleosomal

frags_multi_nucleosomal <- tibble::tibble(
  chr = 1,
  start = seq(12000, 15000, by = 1000),
  end = start + 294,  # Value equal to 2x nucleosome_width
  cell_id = c(rep("cell1", 2), rep("cell2", 2))
)
frags_multi_nucleosomal

frags <- dplyr::bind_rows(
  frags_sub_nucleosomal,
  frags_nucleosomal,
  frags_multi_nucleosomal
) %>% convert_to_fragments()


## Get nucleosome counts
head(nucleosome_counts(frags))
```

`open_fragments_10x` *Read/write a 10x fragments file*

Description

10x fragment files come in a bed-like format, with columns chr, start, end, cell_id, and pcr_duplicates. Unlike a standard bed format, the format from cellranger has an *inclusive* end-coordinate, meaning the end coordinate itself is what should be counted as the tagmentation site, rather than offset by 1.

Usage

```
open_fragments_10x(path, comment = "#", end_inclusive = TRUE)

write_fragments_10x(
  fragments,
  path,
  end_inclusive = TRUE,
  append_5th_column = FALSE
)
```

Arguments

`path`	File path (e.g. fragments.tsv or fragments.tsv.gz)
`comment`	Skip lines at beginning of file which start with comment string
`end_inclusive`	Whether the end coordinate of the bed is inclusive – i.e. there was an insertion at the end coordinate rather than the base before the end coordinate. This is the 10x default, though it's not quite standard for the bed file format.
`fragments`	Input fragments object
`append_5th_column`	Whether to include 5th column of all 0 for compatibility with 10x fragment file outputs (defaults to 4 columns chr,start,end,cell)

Details

open_fragments_10x

No disk operations will take place until the fragments are used in a function

write_fragments_10x

Fragments will be written to disk immediately, then returned in a readable object.

Value

10x fragments file object

Examples

```
## Download example fragments from pbmc 500 dataset and save in temp directory
data_dir <- file.path(tempdir(), "frags_10x")
dir.create(data_dir, recursive = TRUE, showWarnings = FALSE)
url_base <- "https://cf.10xgenomics.com/samples/cell-atac/2.0.0/atac_pbmc_500_nextgem/"
frags_file <- "atac_pbmc_500_nextgem_fragments.tsv.gz"
atac_raw_url <- paste0(url_base, frags_file)
if (!file.exists(file.path(data_dir, frags_file))) {
 download.file(atac_raw_url, file.path(data_dir, frags_file), mode="wb")
}

########################################################################
## open_fragments_10x() example
########################################################################
frags <- open_fragments_10x(
 file.path(data_dir, frags_file)
)
## A Fragments object imported from 10x will not have cell/chromosome
## information directly known unless written as a BPCells fragment object
frags

frags %>% write_fragments_dir(
 file.path(data_dir, "demo_frags_from_h5"),
 overwrite = TRUE
)


########################################################################
## write_fragments_10x() example
########################################################################
frags <- write_fragments_10x(
 frags,
 file.path(data_dir, paste0("new_", frags_file))
)
frags
```

---

open_matrix_10x_hdf5 *Read/write a 10x feature matrix*

Description

Read/write a 10x feature matrix

Usage

```
open_matrix_10x_hdf5(path, feature_type = NULL, buffer_size = 16384L)
```

```
write_matrix_10x_hdf5(
  mat,
  path,
  barcodes = colnames(mat),
  feature_ids = rownames(mat),
  feature_names = rownames(mat),
  feature_types = "Gene Expression",
  feature_metadata = list(),
  buffer_size = 16384L,
  chunk_size = 1024L,
  gzip_level = 0L,
  type = c("uint32_t", "double", "float", "auto")
)
```

Arguments

path	Path to the hdf5 file on disk
feature_type	Optional selection of feature types to include in output matrix. For multiome data, the options are "Gene Expression" and "Peaks". This option is only compatible with files from cellranger 3.0 and newer.
buffer_size	For performance tuning only. The number of items to be buffered in memory before calling writes to disk.
mat	IterableMatrix
barcodes	Vector of names for the cells
feature_ids	Vector of IDs for the features
feature_names	Vector of names for the features
feature_types	String or vector of feature types
feature_metadata	Named list of additional metadata vectors to store for each feature
chunk_size	For performance tuning only. The chunk size used for the HDF5 array storage.
gzip_level	Gzip compression level. Default is 0 (no compression)
type	Data type of the output matrix. Default is `uint32_t` to match a matrix of 10x UMI counts. Non-integer data types include `float` and `double`. If `auto`, will use the data type of `mat`.

Details

The 10x format makes use of gzip compression for the matrix data, which can slow down read performance. Consider writing into another format if the read performance is important to you.

Input matrices must be in column-major storage order, and if the rownames and colnames are not set, names must be provided for the relevant metadata parameters. Some of the metadata parameters are not read by default in BPCells, but it is possible to export them for use with other tools.

Value

BPCells matrix object

Examples

```
## Download example matrices from pbmc 500 dataset and save in temp directory
data_dir <- file.path(tempdir(), "mat_10x")
dir.create(data_dir, recursive = TRUE, showWarnings = FALSE)
url_base <- "https://cf.10xgenomics.com/samples/cell-exp/6.1.0/500_PBMC_3p_LT_Chromium_X/"
mat_file <- "500_PBMC_3p_LT_Chromium_X_filtered_feature_bc_matrix.h5"
rna_url <- paste0(url_base, mat_file)
if (!file.exists(file.path(data_dir, mat_file))) {
 download.file(rna_url, file.path(data_dir, mat_file), mode="wb")
}

########################################################################
## open_matrix_10x_hdf5() example
########################################################################
mat <- open_matrix_10x_hdf5(
 file.path(data_dir, mat_file)
)
mat


########################################################################
## write_matrix_10x_hdf5() example
########################################################################
mat <- write_matrix_10x_hdf5(
 mat,
 file.path(data_dir, paste0("new", mat_file))
)
mat
```

`open_matrix_anndata_hdf5` *Read/write AnnData matrix*

Description

Read or write a matrix from an anndata hdf5 file. These functions will automatically transpose matrices when converting to/from the AnnData format. This is because the AnnData convention stores cells as rows, whereas the R convention stores cells as columns. If this behavior is undesired, call `t()` manually on the matrix inputs and outputs of these functions.

Most users writing to AnnData files should default to `write_matrix_anndata_hdf5()` rather than the dense variant (see details for more information).

Usage

```
open_matrix_anndata_hdf5(path, group = "X", buffer_size = 16384L)

write_matrix_anndata_hdf5(
```

```
  mat,
  path,
  group = "X",
  buffer_size = 16384L,
  chunk_size = 1024L,
  gzip_level = 0L
)

write_matrix_anndata_hdf5_dense(
  mat,
  path,
  dataset = "X",
  buffer_size = 16384L,
  chunk_size = 1024L,
  gzip_level = 0L
)
```

Arguments

path	Path to the hdf5 file on disk
group	The group within the hdf5 file to write the data to. If writing to an existing hdf5 file this group must not already be in use
buffer_size	For performance tuning only. The number of items to be buffered in memory before calling writes to disk.
chunk_size	For performance tuning only. The chunk size used for the HDF5 array storage.
gzip_level	Gzip compression level. Default is 0 (no compression)
dataset	The dataset within the hdf5 file to write the matrix to. Used for `write_matrix_anndata_hdf5_dense`

Details

Efficiency considerations: Reading from a dense AnnData matrix will generally be slower than sparse for single cell datasets, so it is recommended to re-write any dense AnnData inputs to a sparse format early in processing.

`write_matrix_anndata_hdf5()` should be used by default, as it always writes in the more efficient sparse format. `write_matrix_anndata_hdf5_dense()` writes in the AnnData dense format, and can be used for smaller matrices when efficiency and file size are less of a concern than increased portability (e.g. writing to `obsm` or `varm` matrices). See the AnnData docs for format details.

Dimension names: Dimnames are inferred from `obs/_index` or `var/_index` based on length matching. This helps to infer dimnames for `obsp`, `varm`, etc. If the number of `len(obs) == len(var)`, dimname inference will be disabled.

Signed integers: When int32 and int64 matrices are read, they will be converted to float and double matrices respectively. This is because BPCells only supports unsigned integer matrices, and signed integer matrices would have their negative values misinterpreted as zeros.

Value

AnnDataMatrixH5 object, with cells as the columns.

Examples

```
## Create temporary directory to keep demo matrix
data_dir <- file.path(tempdir(), "mat_anndata")
if (dir.exists(data_dir)) unlink(data_dir, recursive = TRUE)
dir.create(data_dir, recursive = TRUE, showWarnings = FALSE)
mat <- get_demo_mat()


#############################################################################
## write_matrix_anndata_hdf5() example
#############################################################################
mat <- write_matrix_anndata_hdf5(
 mat,
 file.path(data_dir, paste0("new_demo_mat.h5"))
)
mat


#############################################################################
## open_matrix_anndata_hdf5() example
#############################################################################
mat <- open_matrix_anndata_hdf5(
 file.path(data_dir, paste0("new_demo_mat.h5"))
)
mat


#############################################################################
## write_matrix_anndata_hdf5_dense() example
#############################################################################
mat <- write_matrix_anndata_hdf5_dense(
 mat,
 file.path(data_dir, paste0("new_demo_mat_dense.h5"))
)
mat
```

`order_ranges` *Get end-sorted ordering for genome ranges*

Description

Use this function to order regioins prior to calling `peak_matrix()` or `tile_matrix()`.

Usage

```
order_ranges(ranges, chr_levels, sort_by_end = TRUE)
```

Arguments

ranges	Genomic regions given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position
chr_levels	Ordering of chromosome names
sort_by_end	If TRUE (defualt), sort by (chr, end, start). Else sort by (chr, start, end)

Value

Numeric vector analagous to the `order` function. Provides an index selection that will reorder the input ranges to be sorted by chr, end, start

Examples

```
## Prep data
ranges <- tibble::tibble(
  chr = "chr1",
  start = seq(10, 260, 50),
  end = start + seq(310, 0, -60),
  cell_id = paste0("cell1")
) %>% as("GRanges")
ranges


## Get end-sorted ordering
order_ranges(ranges, levels(GenomicRanges::seqnames(ranges)))
```

---

peak_matrix *Calculate ranges x cells overlap matrix*

Description

Calculate ranges x cells overlap matrix

Usage

```
peak_matrix(
  fragments,
  ranges,
  mode = c("insertions", "fragments", "overlaps"),
  zero_based_coords = !is(ranges, "GRanges"),
  explicit_peak_names = TRUE
)
```

Arguments

`fragments`	Input fragments object. Must have cell names and chromosome names defined
`ranges`	Peaks/ranges to overlap, given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position
`mode`	Mode for counting peak overlaps. (See "value" section for more details)
`zero_based_coords`	Whether to convert the ranges from a 1-based end-inclusive coordinate system to a 0-based end-exclusive coordinate system. Defaults to true for GRanges and false for other formats (see this archived UCSC blogpost)
`explicit_peak_names`	Boolean for whether to add rownames to the output matrix in format e.g chr1:500-1000, where start and end coords are given in a 0-based coordinate system. Note that either way, peak names will be written when the matrix is saved.

Value

Iterable matrix object with dimension ranges x cells. When saved, the column names of the output matrix will be in the format chr1:500-1000, where start and end coords are given in a 0-based coordinate system.

`mode` **options**

- `"insertions"`: Start and end coordinates are separately overlapped with each peak
- `"fragments"`: Like `"insertions"`, but each fragment can contribute at most 1 count to each peak, even if both the start and end coordinates overlap
- `"overlaps"`: Like `"fragments"`, but an overlap is also counted if the fragment fully spans the peak even if neither the start or end falls within the peak

Note

When calculating the matrix directly from a fragments tsv, it's necessary to first call `select_chromosomes()` in order to provide the ordering of chromosomes to expect while reading the tsv.

Examples

```
## Prep demo data
frags <- get_demo_frags(subset = FALSE)
chrom_sizes <- read_ucsc_chrom_sizes(file.path(tempdir(), "references"), genome="hg38")
blacklist <- read_encode_blacklist(file.path(tempdir(), "references"), genome="hg38")
frags_filter_blacklist <- frags %>% select_regions(blacklist, invert_selection = TRUE)
peaks <- call_peaks_tile(
  frags_filter_blacklist,
  chrom_sizes,
  effective_genome_size = 2.8e9
)
top_peaks <- head(peaks, 5000)
top_peaks <- top_peaks[order_ranges(top_peaks, chrNames(frags)),]
```

```
## Get peak matrix
peak_matrix(frags_filter_blacklist, top_peaks, mode="insertions")
```

---

plot_dot *Dotplot*

Description

Plot feature levels per group or cluster as a grid of dots. Dots are colored by z-score normalized average expression, and sized by percent non-zero.

Usage

```
plot_dot(
  source,
  features,
  groups,
  group_order = NULL,
  gene_mapping = human_gene_mapping,
  colors = c("lightgrey", "#4682B4"),
  return_data = FALSE,
  apply_styling = TRUE
)
```

Arguments

source	Feature x cell matrix or data.frame with features. For best results, features should be sparse and log-normalized (e.g. run `log1p()` so zero raw counts map to zero)
features	Character vector of features to plot
groups	Vector with one entry per cell, specifying the cell's group
group_order	Optional vector listing ordering of groups
gene_mapping	An optional vector for gene name matching with match_gene_symbol().
colors	Color scale for plot
return_data	If true, return data from just before plotting rather than a plot.
apply_styling	If false, return a plot without pretty styling applied

Examples

```
## Prep data
mat <- get_demo_mat()
cell_types <- paste("Group", rep(1:3, length.out = length(colnames(mat))))

## Plot dot
plot <- plot_dot(mat, c("MS4A1", "CD3E"), cell_types)
```

```
BPCells:::render_plot_from_storage(
  plot, width = 4, height = 5
)
```

---

plot_embedding *Plot UMAP or embeddings*

Description

Plot one or more features by coloring cells in a UMAP plot.

Usage

```
plot_embedding(
  source,
  embedding,
  features = NULL,
  quantile_range = c(0.01, 0.99),
  randomize_order = TRUE,
  smooth = NULL,
  smooth_rounds = 3,
  gene_mapping = human_gene_mapping,
  size = NULL,
  rasterize = FALSE,
  raster_pixels = 512,
  legend_continuous = c("auto", "quantile", "value"),
  labels_quantile_range = TRUE,
  colors_continuous = c("lightgrey", "#4682B4"),
  legend_discrete = TRUE,
  labels_discrete = TRUE,
  colors_discrete = discrete_palette("stallion"),
  return_data = FALSE,
  return_plot_list = FALSE,
  apply_styling = TRUE
)
```

Arguments

source	Matrix, or data frame to pull features from, or a vector of feature values for a single feature. For a matrix, the features must be rows.
embedding	A matrix of dimensions cells x 2 with embedding coordinates
features	Character vector of features to plot if source is not a vector.
quantile_range	(optional) Length 2 vector giving the quantiles to clip the minimum and maximum color scale values, as fractions between 0 and 1. NULL or NA values to skip clipping

`randomize_order`
: If TRUE, shuffle cells to prevent overplotting biases. Can pass an integer instead to specify a random seed to use.

`smooth`
: (optional) Sparse matrix of dimensions cells x cells with cell-cell distance weights for smoothing.

`smooth_rounds`
: Number of multiplication rounds to apply when smoothing.

`gene_mapping`
: An optional vector for gene name matching with match_gene_symbol(). Ignored if `source` is a data frame.

`size`
: Point size for plotting

`rasterize`
: Whether to rasterize the point drawing to speed up display in graphics programs.

`raster_pixels`
: Number of pixels to use when rasterizing. Can provide one number for square dimensions, or two numbers for width x height.

`legend_continuous`
: Whether to label continuous features by quantile or value. "auto" labels by quantile only when all features are continuous and `quantile_range` is not NULL. Quantile labeling adds text annotation listing the range of displayed values.

`labels_quantile_range`
: Whether to add a text label with the value range of each feature when the legend is set to quantile

`colors_continuous`
: Vector of colors to use for continuous color palette

`legend_discrete`
: Whether to show the legend for discrete (categorical) features.

`labels_discrete`
: Whether to add text labels at the center of each group for discrete (categorical) features.

`colors_discrete`
: Vector of colors to use for discrete (categorical) features.

`return_data`
: If true, return data from just before plotting rather than a plot.

`return_plot_list`
: If `TRUE`, return multiple plots as a list, rather than a single plot combined using patchwork::wrap_plots()

`apply_styling`
: If false, return a plot without pretty styling applied

Details

Smoothing:

Smoothing is performed as follows: first, the smoothing matrix is normalized so the sum of incoming weights to every cell is 1. Then, the raw data values are repeatedly multiplied by the smoothing matrix and re-scaled so the average value stays the same.

Value

By default, returns a ggplot2 object with all the requested features plotted in a grid. If `return_data` or `return_plot_list` is called, the return value will match that argument.

Examples

```
set.seed(123)
mat <- get_demo_mat()
## Normalize matrix
mat_norm <- log1p(multiply_cols(mat, 1/colSums(mat)) * 10000) %>% write_matrix_memory(compress = FALSE)
## Get variable genes
stats <- matrix_stats(mat, row_stats = "variance")
variable_genes <- order(stats$row_stats["variance",], decreasing=TRUE) %>%
  head(1000) %>%
  sort()
# Z score normalize genes
mat_norm <- mat[variable_genes, ]
gene_means <- stats$row_stats['mean', variable_genes]
gene_vars <- stats$row_stats['variance', variable_genes]
mat_norm <- (mat_norm - gene_means) / gene_vars
## Save matrix to memory
mat_norm <- mat_norm %>% write_matrix_memory(compress = FALSE)
## Run SVD
svd <- BPCells::svds(mat_norm, k = 10)
pca <- multiply_cols(svd$v, svd$d)
## Get UMAP
umap <- uwot::umap(pca)
## Get clusters
clusts <- knn_hnsw(pca, ef = 500) %>%
  knn_to_snn_graph() %>%
  cluster_graph_louvain()


## Plot embeddings
print(length(clusts))

plot_embedding(clusts, umap)


### Can also plot by features
#plot_embedding(
#  source = mat,
#  umap,
#  features = c("MS4A1", "CD3E"),
#)
```

`plot_fragment_length` *Fragment size distribution*

Description

Plot the distribution of fragment lengths, with length in basepairs on the x-axis, and proportion of fragments on the y-axis. Typical plots will show 10-basepair periodicity, as well as humps spaced at multiples of a nucleosome width (about 150bp).

Usage

```
plot_fragment_length(
  fragments,
  max_length = 500,
  return_data = FALSE,
  apply_styling = TRUE
)
```

Arguments

fragments	Fragments object
max_length	Maximum length to show on the plot
return_data	If true, return data from just before plotting rather than a plot.
apply_styling	If false, return a plot without pretty styling applied

Value

Numeric vector where index i contans the number of length-i fragments

Examples

```
frags <- get_demo_frags(filter_qc = FALSE, subset = FALSE)
plot_fragment_length(frags)
```

---

plot_read_count_knee *Knee plot of single cell read counts*

Description

Plots read count rank vs. number of reads on a log-log scale.

Usage

```
plot_read_count_knee(
  read_counts,
  cutoff = NULL,
  return_data = FALSE,
  apply_styling = TRUE
)
```

Arguments

read_counts	Vector of read counts per cell
cutoff	(optional) Read cutoff to mark on the plot
return_data	If true, return data from just before plotting rather than a plot.
apply_styling	If false, return a plot without pretty styling applied

Details

Performs logarithmic downsampling to reduce the number of points plotted

Value

ggplot2 plot object

Examples

```
## Prep data
mat <- get_demo_mat(filter_qc = FALSE, subset = FALSE)
reads_per_cell <- colSums(mat)

# Render knee plot
plot_read_count_knee(reads_per_cell, cutoff = 1e3)
```

---

plot_tf_footprint *Plot TF footprint*

Description

Plot the footprinting around TF motif sites

Usage

```
plot_tf_footprint(
  fragments,
  motif_positions,
  cell_groups = rlang::rep_along(cellNames(fragments), "all"),
  flank = 250L,
  smooth = 0L,
  zero_based_coords = !is(genes, "GRanges"),
  colors = discrete_palette("stallion"),
  return_data = FALSE,
  apply_styling = TRUE
)
```

Arguments

fragments	IterableFragments object
motif_positions	Coordinate ranges for motifs (must include strand) and have constant width
cell_groups	Character or factor assigning a group to each cell, in order of cellNames(fragments)
flank	Number of flanking basepairs to include on either side of the motif
smooth	(optional) Sparse matrix of dimensions cells x cells with cell-cell distance weights for smoothing.

zero_based_coords
: If true, coordinates start and 0 and the end coordinate is not included in the range. If false, coordinates start at 1 and the end coordinate is included in the range

return_data
: If true, return data from just before plotting rather than a plot.

apply_styling
: If false, return a plot without pretty styling applied

See Also

footprint(), plot_tss_profile()

---

plot_tss_profile *Plot TSS profile*

Description

Plot the enrichmment of insertions relative to transcription start sites (TSS). Typically, this plot shows strong enrichment of insertions near a TSS, and a small bump downstream around 220bp downstream of the TSS for the +1 nucleosome.

Usage

```
plot_tss_profile(
  fragments,
  genes,
  cell_groups = rlang::rep_along(cellNames(fragments), "all"),
  flank = 2000L,
  smooth = 0L,
  zero_based_coords = !is(genes, "GRanges"),
  colors = discrete_palette("stallion"),
  return_data = FALSE,
  apply_styling = TRUE
)
```

Arguments

fragments
: IterableFragments object

genes
: Coordinate ranges for genes (must include strand)

cell_groups
: Character or factor assigning a group to each cell, in order of cellNames(fragments)

flank
: Number of flanking basepairs to include on either side of the motif

smooth
: Number of bases to smooth over (rolling average)

zero_based_coords
: If true, coordinates start and 0 and the end coordinate is not included in the range. If false, coordinates start at 1 and the end coordinate is included in the range

return_data
: If true, return data from just before plotting rather than a plot.

apply_styling
: If false, return a plot without pretty styling applied

See Also

footprint(), plot_tf_footprint()

Examples

```
## Prep data
frags <- get_demo_frags(filter_qc = FALSE, subset = FALSE)
genes <- read_gencode_transcripts(
  file.path(tempdir(), "references"), release = "42",
  annotation_set = "basic",
  features = "transcript"
)

## Plot tss profile
plot_tss_profile(frags, genes)
```

---

plot_tss_scatter *TSS Enrichment vs. Fragment Counts plot*

Description

Density scatter plot with log10(fragment_count) on the x-axis and TSS enrichment on the y-axis. This plot is most useful to select which cell barcodes in an experiment correspond to high-quality cells

Usage

```
plot_tss_scatter(
  atac_qc,
  min_frags = NULL,
  min_tss = NULL,
  bins = 100,
  apply_styling = TRUE
)
```

Arguments

atac_qc	Tibble as returned by qc_scATAC(). Must have columns nFrags and TSSEnrichment
min_frags	Minimum fragment count cutoff
min_tss	Minimum TSS Enrichment cutoff
bins	Number of bins for density calculation
apply_styling	If false, return a plot without pretty styling applied

Examples

```
## Prep data
frags <- get_demo_frags(filter_qc = FALSE, subset = FALSE)
genes <- read_gencode_transcripts(
  file.path(tempdir(), "references"), release = "42",
  annotation_set = "basic",
  features = "transcript"
)
blacklist <- read_encode_blacklist(file.path(tempdir(), "references"), genome="hg38")
atac_qc <- qc_scATAC(frags, genes, blacklist)


## Render tss enrichment vs fragment plot
plot_tss_scatter(atac_qc, min_frags = 1000, min_tss = 10)
```

---

prefix_cell_names *Add sample prefix to cell names*

Description

Rename cells by adding a prefix to the names. Most commonly this will be a sample name. All cells will recieve the exact text of `prefix` added to the beginning, so any separator characters like "_" must be included in the given `prefix`. Use this prior to merging fragments from different experiments with `c()` in order to help prevent cell name clashes.

Usage

```
prefix_cell_names(fragments, prefix)
```

Arguments

`fragments`	Input fragments object.
`prefix`	String to add as the prefix

Value

Fragments object with prefixed names

Examples

```
## Prep data
frags <- tibble::tibble(
  chr = "chr1",
  start = seq(10, 260, 50),
  end = start + 30,
  cell_id = paste0("cell", c(rep(1, 2), rep(2,2), rep(3,2)))
) %>% convert_to_fragments()
frags
```

```
## Prefix cells with foo
prefix_cell_names(frags, "foo_") %>% as("GRanges")
```

`pseudobulk_matrix` *Aggregate counts matrices by cell group or feature.*

Description

Given a (`features x cells`) matrix, group cells by `cell_groups` and aggregate counts by `method` for each feature.

Usage

```
pseudobulk_matrix(mat, cell_groups, method = "sum", threads = 0L)
```

Arguments

`mat`	IterableMatrix object of dimensions features x cells
`cell_groups`	(Character/factor) Vector of group/cluster assignments for each cell. Length must be `ncol(mat)`.
`method`	(Character vector) Method(s) to aggregate counts. If one method is provided, the output will be a matrix. If multiple methods are provided, the output will be a named list of matrices. Current options are: `nonzeros`, `sum`, `mean`, `variance`.
`threads`	(integer) Number of threads to use.

Details

Some simpler stats are calculated in the process of calculating more complex statistics. So when calculating `variance`, `nonzeros` and `mean` can be included with no extra calculation time, and when calculating `mean`, adding `nonzeros` will take no extra time.

Value

- If `method` is length 1, returns a matrix of shape (`features x groups`).
- If `method` is greater than length 1, returns a list of matrices with each matrix representing a pseudobulk matrix with a different aggregation method. Each matrix is of shape (`features x groups`), and names are one of `nonzeros`, `sum`, `mean`, `variance`.

Examples

```
set.seed(12345)
mat <- matrix(rpois(100, lambda = 5), nrow = 10)
rownames(mat) <- paste0("gene", 1:10)
colnames(mat) <- paste0("cell", 1:10)
mat <- mat %>% as("dgCMatrix") %>% as("IterableMatrix")
groups <- rep(c("Cluster1", "Cluster2"), each = 5)

## When calculating only sum across two groups
pseudobulk_res <- pseudobulk_matrix(
  mat = mat,
  cell_groups = groups,
  method = "sum"
)
pseudobulk_res

## Can also request multiple summary statistics for pseudoulking
pseudobulk_res_multi <- pseudobulk_matrix(
  mat = mat,
  cell_groups = groups,
  method = c("mean",  "variance")
)

names(pseudobulk_res_multi)

pseudobulk_res_multi$mean
```

---

qc_scATAC *Calculate ArchR-compatible per-cell QC statistics*

Description

Calculate ArchR-compatible per-cell QC statistics

Usage

```
qc_scATAC(fragments, genes, blacklist)
```

Arguments

fragments	IterableFragments object
genes	Gene coordinates given as GRanges, data.frame, or list. See help("genomic-ranges-like") for details on format and coordinate systems. Required attributes: • chr, start, end: genomic position
blacklist	Blacklisted regions given as GRanges, data.frame, or list. See help("genomic-ranges-like") for details on format and coordinate systems. Required attributes: • chr, start, end: genomic position

Details

This implementation mimics ArchR's default parameters. For uses requiring more flexibility to tweak default parameters, the best option is to re-implement this function with required changes. Output columns of data.frame:

- `cellName`: cell name for each cell
- `nFrags`: number of fragments per cell
- `subNucleosomal`, `monoNucleosomal`, `multiNucleosomal`: number of fragments of size 1-146bp, 147-254bp, and 255bp + respectively. equivalent to ArchR's nMonoFrags, nDiFrags, nMultiFrags respectively
- `TSSEnrichment`: `AvgInsertInTSS / max(AvgInsertFlankingTSS, 0.1)`, where `AvgInsertInTSS` is `ReadsInTSS / 101` (window size), and `AvgInsertFlankingTSS` is `ReadsFlankingTSS / (100*2)` (window size). The `max(0.1)` ensures that very low-read cells do not get assigned spuriously high TSSEnrichment.
- `ReadsInPromoter`: Number of reads from 2000bp upstream of TSS to 101bp downstream of TSS
- `ReadsInBlacklist`: Number of reads in the provided blacklist region
- `ReadsInTSS`: Number of reads overlapping the 101bp centered around each TSS
- `ReadsFlankingTSS`: Number of reads overlapping 1901-2000bp +/- each TSS

Differences from ArchR: Note that ArchR by default uses a different set of annotations to derive TSS sites and promoter sites. This function uses just one annotation for gene start+end sites, so must be called twice to exactly re-calculate the ArchR QC stats.

ArchR's `PromoterRatio` and `BlacklistRatio` are not included in the output, as they can be easily calculated from `ReadsInPromoter / nFrags` and `ReadsInBlacklist / nFrags`. Similarly, ArchR's `NucleosomeRatio` can be calculated as `(monoNucleosomal + multiNucleosomal) / subNucleosomal`.

Value

data.frame with QC data

Examples

```
## Prep data
frags <- get_demo_frags(subset = FALSE)
reference_dir <- file.path(tempdir(), "references")
genes <- read_gencode_transcripts(
  reference_dir,
  release="42",
  transcript_choice="MANE_Select",
  annotation_set = "basic",
  features="transcript"
)
blacklist <- read_encode_blacklist(reference_dir, genome = "hg38")


## Run qc
head(qc_scATAC(frags, genes, blacklist))
```

`range_distance_to_nearest` *Find signed distance to nearest genomic ranges*

Description

Given a set of genomic ranges, find the distance to the nearest neighbors both upstream and downstream.

Usage

```
range_distance_to_nearest(
  ranges,
  addArchRBug = FALSE,
  zero_based_coords = !is(ranges, "GRanges")
)
```

Arguments

`ranges`	Genomic regions given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position • `strand`: +/- or TRUE/FALSE for positive or negative strand
`addArchRBug`	boolean to reproduce ArchR's bug that incorrectly handles nested genes
`zero_based_coords`	If true, coordinates start and 0 and the end coordinate is not included in the range. If false, coordinates start at 1 and the end coordinate is included in the range

Value

A 2-column data.frame with columns upstream and downstream, containing the distances to the nearest neighbor in the respective directions. For ranges on `+` or `*` strand, distance is calculated as:

- upstream = `max(start(range) - end(upstreamNeighbor), 0)`
- downstream = `max(start(downstreamNeighbor) - end(range), 0)`

For ranges on `-` strand, the definition of upstream and downstream is flipped. Note that this definition of distance is one off from `GenomicRanges::distance()`, as ranges that neighbor but don't overlap are given a distance of 1 rather than 0.

Examples

```
## Prep data
ranges <- tibble::tibble(
 chr = "chr1",
 start = seq(10, 410, 100),
```

```
 end = start + 50,
 strand = "+"
)
## Add one range that is completely nested in the other ranges
ranges_with_nesting <- ranges %>%
 tibble::add_row(chr = "chr1", start = 11, end = 20, strand = "+")


## Get range distance to nearest
range_distance_to_nearest(ranges_with_nesting)
```

---

read_bed *Read a bed file into a data frame*

Description

Bed files can contain peak or blacklist annotations. These utilities help read thos annotations

Usage

```
read_bed(
  path,
  additional_columns = character(0),
  backup_url = NULL,
  timeout = 300
)

read_encode_blacklist(
  dir,
  genome = c("hg38", "mm10", "hg19", "dm6", "dm3", "ce11", "ce10"),
  timeout = 300
)
```

Arguments

path	Path to file (or desired save location if backup_url is used)
additional_columns	Names for additional columns in the bed file
backup_url	If path does not exist, provides a URL to download the gtf from
timeout	Maximum time in seconds to wait for download from backup_url
dir	Output directory to cache the downloaded gtf file
genome	genome name

Details

read_bed

Read a bed file from disk or a url.

read_encode_blacklist

Downloads the Boyle Lab blacklist, as described in https://doi.org/10.1038/s41598-019-45839-z

Value

Data frame with coordinates using the 0-based convention.

See Also

read_gtf(), read_gencode_genes()

Examples

```
## Dummy bed file creation
data.frame(
 chrom = rep("chr1", 6),
 start = seq(20, 121, 20),
 end = seq(39, 140, 20)
) %>% write.table("./references/example.bed", row.names = FALSE, col.names = FALSE, sep = "\t")


################################################################################
## read_bed() example
################################################################################
read_bed("./references/example.bed")


################################################################################
## read_encode_blacklist() example
################################################################################
read_encode_blacklist("./reference")
```

---

read_gtf *Read GTF gene annotations*

Description

Read gene annotations from gtf format into a data frame. The source can be a URL, a gtf file on disk, or a gencode release version.

Usage

```
read_gtf(
  path,
  attributes = c("gene_id"),
  tags = character(0),
  features = c("gene"),
  keep_attribute_column = FALSE,
  backup_url = NULL,
  timeout = 300
)

read_gencode_genes(
  dir,
  release = "latest",
  annotation_set = c("basic", "comprehensive"),
  gene_type = "lncRNA|protein_coding|IG_.*_gene|TR_.*_gene",
  attributes = c("gene_id", "gene_type", "gene_name"),
  tags = character(0),
  features = c("gene"),
  timeout = 300
)

read_gencode_transcripts(
  dir,
  release = "latest",
  transcript_choice = c("MANE_Select", "Ensembl_Canonical", "all"),
  annotation_set = c("basic", "comprehensive"),
  gene_type = "lncRNA|protein_coding|IG_.*_gene|TR_.*_gene",
  attributes = c("gene_id", "gene_type", "gene_name", "transcript_id"),
  features = c("transcript", "exon"),
  timeout = 300
)
```

Arguments

path	Path to file (or desired save location if backup_url is used)
attributes	Vector of GTF attribute names to parse out as columns
tags	Vector of tags to parse out as boolean presence/absence
features	List of features types to keep from the GTF (e.g. gene, transcript, exon, intron)
keep_attribute_column	Boolean for whether to preserve the raw attribute text column
backup_url	If path does not exist, provides a URL to download the gtf from
timeout	Maximum time in seconds to wait for download from backup_url
dir	Output directory to cache the downloaded gtf file
release	release version (prefix with M for mouse versions). For most recent version, use "latest" or "latest_mouse"

`annotation_set` Either "basic" or "comprehensive" annotation sets (see details section).

`gene_type` Regular expression with which gene types to keep. Defaults to protein_coding, lncRNA, and IG/TR genes

`transcript_choice`
Method for selecting representative transcripts. Choices are:

- MANE_Select: human-only, most conservative
- Ensembl_Canonical: human+mouse, superset of MANE_Select for human
- all: Preserve all transcript models (not recommended for plotting)

Details

read_gtf

Read gtf from a file or URL

read_gencode_genes

Read gene annotations directly from GENCODE. The file name will vary depending on the release and annotation set requested, but will be of the format `gencode.v42.annotation.gtf.gz`. GENCODE currently recommends the basic set: https://www.gencodegenes.org/human/. In release 42, both the comprehensive and basic sets had identical gene-level annotations, but the comprehensive set had additional transcript variants annotated.

read_gencode_transcripts

Read transcript models from GENCODE, for use with trackplot_gene()

Value

Data frame with coordinates using the 0-based convention. Columns are:

- chr
- source
- feature
- start
- end
- score
- strand
- frame
- attributes (optional; named according to listed attributes)
- tags (named according to listed tags)

See Also

`read_bed()`, `read_encode_blacklist()`

Examples

```
#############################################################################
## read_gtf() example
#############################################################################
species <- "Saccharomyces_cerevisiae"
version <- "GCF_000146045.2_R64"
head(read_gtf(
 path = sprintf("./reference/%s_genomic.gtf.gz", version),
 backup_url = sprintf(
  "https://ftp.ncbi.nlm.nih.gov/genomes/refseq/fungi/%s/reference/%s/%s_genomic.gtf.gz",
   species, version, version
 )
))


#############################################################################
## read_gencode_genes() example
#############################################################################
read_gencode_genes("./references", release = "42")


#############################################################################
## read_gencode_transcripts() example
#############################################################################
## If read_gencode_genes() was already ran on the same release,
## will reuse previously downloaded annotations
read_gencode_transcripts("./references", release = "42")
```

`read_ucsc_chrom_sizes` *Read UCSC chromosome sizes*

Description

Read chromosome sizes from UCSC and return as a tibble with one row per chromosome. The underlying data is pulled from here: https://hgdownload.soe.ucsc.edu/downloads.html

Usage

```
read_ucsc_chrom_sizes(
  dir,
  genome = c("hg38", "mm39", "mm10", "mm9", "hg19"),
  keep_chromosomes = "chr[0-9]+|chrX|chrY",
  timeout = 300
)
```

Examples

```
read_ucsc_chrom_sizes("./reference")
```

`regress_out` *Regress out unwanted variation*

Description

Regress out the effects of confounding variables using a linear least squares regression model.

Usage

```
regress_out(mat, latent_data, prediction_axis = c("row", "col"))
```

Arguments

`mat`	Input IterableMatrix
`latent_data`	Data to regress out, as a `data.frame` where each column is a variable to regress out.
`prediction_axis`	Which axis corresponds to prediction outputs from the linear models (e.g. the gene axis in typical single cell analysis). Options include "row" (default) and "col".

Details

Conceptually, `regress_out` calculates a linear least squares best fit model for each row of the matrix. (Or column if `prediction_axis` is `"col"`). The input data for each regression model are the columns of `latent_data`, and each model tries to predict the values in the corresponding row (or column) of `mat`. After fitting each model, `regress_out` will subtract the model predictions from the input values, aiming to only retain effects that are not explained by the variables in `latent_data`.

These models can be fit efficiently since they all share the same input data and so most of the calculations for the closed-form best fit solution are shared. A QR factorization of the model matrix and a dense matrix-vector multiply are sufficient to fully calculate the residual values.

Efficiency considerations: As the output matrix is dense rather than sparse, mean and variance calculations may run comparatively slowly. However, PCA and matrix/vector multiply operations can be performed at nearly the same cost as the input matrix due to mathematical simplifications. Memory usage scales with `n_features * ((nrow(mat) + ncol(mat))`. Generally, `n_features == ncol(latent_data)`, but for categorical variables in `latent_data`, each category will be expanded into its own indicator variable. Memory usage will therefore be higher when using categorical input variables with many (i.e. >100) distinct values.

Value

IterableMatrix

rotate_x_labels *Rotate ggplot x axis labels*

Description

Rotate ggplot x axis labels

Usage

```
rotate_x_labels(degrees = 45)
```

Arguments

degrees	Number of degrees to rotate by

sctransform_pearson *SCTransform Pearson Residuals*

Description

Calculate pearson residuals of a negative binomial sctransform model. Normalized values are calculated as `(X - mu) / sqrt(mu + mu^2/theta)`. mu is calculated as `cell_read_counts * gene_beta`.

Usage

```
sctransform_pearson(
  mat,
  gene_theta,
  gene_beta,
  cell_read_counts,
  min_var = -Inf,
  clip_range = c(-10, 10),
  columns_are_cells = TRUE,
  slow = FALSE
)
```

Arguments

mat	IterableMatrix (raw counts)
gene_theta	Vector of per-gene thetas (overdispersion values)
gene_beta	Vector of per-gene betas (expression level values)
cell_read_counts	Vector of total reads per (umi count for RNA)
min_var	Minimum value for clipping variance

clip_range	Length 2 vector of min and max clipping range
columns_are_cells	Whether the columns of the matrix correspond to cells (default) or genes
slow	If TRUE, use a 10x slower but more precise implementation (default FALSE)

Details

The parameterization used is somewhat simplified compared to the original SCTransform paper, in particular it uses a linear-scale rather than log-scale to represent the cell_read_counts and gene_beta variables. It also does not support the addition of arbitrary cell metadata (e.g. batch) to add to the negative binomial regression.

Value

IterableMatrix

---

select_cells *Subset, translate, or reorder cell IDs*

Description

Subset, translate, or reorder cell IDs

Usage

```
select_cells(fragments, cell_selection)
```

Arguments

fragments	Input fragments object
cell_selection	List of cell IDs (numeric), names (character), or logical mask.

Details

Numeric cell IDs will be re-assigned in the order of `cell_selection`. The output cell ID `n` will be taken from the input cell with ID/name `cell_selection[n]`.

Examples

```
## Prep data
frags <- tibble::tibble(
  chr = "chr1",
  start = seq(10, 260, 50),
  end = start + 30,
  cell_id = paste0("cell", c(rep(1, 2), rep(2,2), rep(3,2)))
) %>% convert_to_fragments()
frags
```

```
## Select cells by name
select_cells(frags, "cell1")

## Select cells by index
select_cells(frags, c(1,3))
```

`select_chromosomes` *Subset, translate, or reorder chromosome IDs*

Description

Subset, translate, or reorder chromosome IDs

Usage

```
select_chromosomes(fragments, chromosome_selection)
```

Arguments

`fragments`	Input fragments object
`chromosome_selection`	List of chromosme IDs (numeric), or names (character), or logical mask.

Details

Numeric chromosome IDs will be re-assigned in the order of `chromosome_selection`. The output chromosome ID `n` will be taken from the input chromosome with ID/name `chromosome_selection[n]`.

Examples

```
## Prep data
frags <- tibble::tibble(
  chr = c(rep("chr1", 2), rep("chrX", 2), rep("chr3", 2)),
  start = seq(10, 260, 50),
  end = start + 30,
  cell_id = paste0("cell1")
) %>% as("GRanges")
frags <- frags %>% convert_to_fragments()
frags

## Selecting by chromosome IDs
select_chromosomes(frags, c(1, 3))

## Selecting by name
select_chromosomes(frags, c("chrX"))
```

`select_regions` *Subset fragments by genomic region*

Description

Fragments can be subset based on overlapping (or not overlapping) a set of regions

Usage

```
select_regions(
  fragments,
  ranges,
  invert_selection = FALSE,
  zero_based_coords = !is(ranges, "GRanges")
)
```

Arguments

`fragments`	Input fragments object.
`ranges`	Peaks/ranges to overlap, given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position
`invert_selection`	If TRUE, select fragments *not* overlapping selected regions instead of only fragments overlapping the selected regions.
`zero_based_coords`	Whether to convert the ranges from a 1-based end-inclusive coordinate system to a 0-based end-exclusive coordinate system. Defaults to true for GRanges and false for other formats (see this archived UCSC blogpost)

Value

Fragments object filtered according to the selected regions

Examples

```
frags <- tibble::tibble(
  chr = "chr1",
  start = seq(10, 260, 50),
  end = start + seq(5, 30, 5),
  cell_id = "cell1"
)
frags
frags <- frags %>% convert_to_fragments()

region <- tibble::tibble(
  chr = "chr1",
```

```
  start = 60,
  end = 130
) %>% as("GRanges")

## Select ranges overlapping with region
select_regions(frags, region) %>% as("GRanges")

## Select ranges not overlapping with region
select_regions(frags, region, invert_selection = TRUE) %>% as("GRanges")
```

`set_trackplot_label` *Adjust trackplot properties*

Description

Adjust labels and heights on trackplots. Labels are set as facet labels in ggplot2, and heights are additional properties read by `trackplot_combine()` to determine relative height of input plots.

Usage

```
set_trackplot_label(plot, labels)

set_trackplot_height(plot, height)

get_trackplot_height(plot)
```

Arguments

`plot`	ggplot object
`labels`	character vector of labels – must match existing number of facets in plot
`height`	New height. If numeric, adjusts relative height. If `ggplot2::unit` or `grid::unit` sets absolute height in specified units. `"null"` units are interpreted as relative height.

Value

set_trackplot_label: ggplot object with adjusted facet labels

set_trackplot_height: ggplot object with adjusted trackplot height

get_trackplot_height: `ggplot2::unit` object with height setting

shift_fragments *Shift start or end coordinates*

Description

Shifts start or end of fragments by a fixed amount, which can be useful to correct the Tn5 offset.

Usage

```
shift_fragments(fragments, shift_start = 0L, shift_end = 0L)
```

Arguments

fragments	Input fragments object
shift_start	How many basepairs to shift the start coords
shift_end	How many basepairs to shift the end coords

Details

The correct Tn5 offset is +/- 4bp since the Tn5 cut sites on opposite strands are offset by 9bp. However, +4/-5 bp is often applied to bed-format files, since the end coordinate in bed files is 1 past the last basepair of the sequenced DNA fragment. This results in a bed-like format except with inclusive end coordinates.

Value

Shifted fragments object

Examples

```
## Prep data
frags <- tibble::tibble(
  chr = "chr1",
  start = seq(10, 260, 50),
  end = start + 30,
  cell_id = paste0("cell1")
) %>% as("GRanges")
frags

frags <- frags %>% convert_to_fragments()


## Shift fragments
shift_fragments(frags, shift_start = 4, shift_end = -4) %>% as("GRanges")
```

subset_lengths *Subset fragments by length*

Description

Subset fragments by length

Usage

```
subset_lengths(fragments, min_len = 0L, max_len = NA_integer_)
```

Arguments

fragments	Input fragments object
min_len	Minimum bases in fragment (inclusive)
max_len	Maximum bases in fragment (inclusive)

Details

Fragment length is calculated as `end-start`

Value

Fragments object

Examples

```
## Prep data
frags <- tibble::tibble(
  chr = "chr1",
  start = seq(10, 260, 50),
  end = start + seq(5, 30, 5),
  cell_id = paste0("cell", c(rep(1, 2), rep(2,2), rep(3,2)))
)
frags
frags <- frags %>% convert_to_fragments()

## Subset lengths
subset_lengths(frags, min_len = 10, max_len = 20) %>% as("GRanges")
```

svds *Calculate svds*

Description

Use the C++ Spectra solver (same as RSpectra package), in order to compute the largest k values and corresponding singular vectors. Empirically, memory usage is much lower than using `irlba::irlba()`, likely due to avoiding R garbage creation while solving due to the pure-C++ solver. This documentation is a slightly-edited version of the `RSpectra::svds()` documentation.

Usage

```
svds(A, k, nu = k, nv = k, opts = list(), threads=0L, ...)
```

Arguments

`A`	The matrix whose truncated SVD is to be computed.
`k`	Number of singular values requested.
`nu`	Number of right singular vectors to be computed. This must be between 0 and 'k'. (Must be equal to 'k' for BPCells IterableMatrix)
`opts`	Control parameters related to computing algorithm. See *Details* below
`threads`	Control threads to use calculating mat-vec producs (BPCells specific)

Details

When RSpectra is installed, this function will just add a method to `RSpectra::svds()` for the `IterableMatrix` class.

The `opts` argument is a list that can supply any of the following parameters:

`ncv` Number of Lanzcos basis vectors to use. More vectors will result in faster convergence, but with greater memory use. `ncv` must be satisfy $k < ncv \leq p$ where `p = min(m, n)`. Default is `min(p, max(2*k+1, 20))`.

`tol` Precision parameter. Default is 1e-10.

`maxitr` Maximum number of iterations. Default is 1000.

`center` Either a logical value (`TRUE`/`FALSE`), or a numeric vector of length n. If a vector c is supplied, then SVD is computed on the matrix $A - 1c'$, in an implicit way without actually forming this matrix. `center = TRUE` has the same effect as `center = colMeans(A)`. Default is `FALSE`. Ignored in BPCells

`scale` Either a logical value (`TRUE`/`FALSE`), or a numeric vector of length n. If a vector s is supplied, then SVD is computed on the matrix $(A - 1c')S$, where c is the centering vector and $S = diag(1/s)$. If `scale = TRUE`, then the vector s is computed as the column norm of $A - 1c'$. Default is `FALSE`. Ignored in BPCells

Value

A list with the following components:

d	A vector of the computed singular values.
u	An m by nu matrix whose columns contain the left singular vectors. If nu == 0, NULL will be returned.
v	An n by nv matrix whose columns contain the right singular vectors. If nv == 0, NULL will be returned.
nconv	Number of converged singular values.
niter	Number of iterations used.
nops	Number of matrix-vector multiplications used.

References

Qiu Y, Mei J (2022). *RSpectra: Solvers for Large-Scale Eigenvalue and SVD Problems*. R package version 0.16-1, https://CRAN.R-project.org/package=RSpectra.

Examples

```
mat <- matrix(rnorm(500), nrow = 50, ncol = 10)
rownames(mat) <- paste0("gene", seq_len(50))
colnames(mat) <- paste0("cell", seq_len(10))
mat <- mat %>% as("dgCMatrix") %>% as("IterableMatrix")

svd_res <- svds(mat, k = 5)

names(svd_res)

svd_res$d

dim(svd_res$u)

dim(svd_res$v)
# Can also pass in values directly into RSpectra::svds
svd_res <- svds(mat, k = 5, opts=c(maxitr = 500))
```

---

tile_matrix *Calculate ranges x cells tile overlap matrix*

Description

Calculate ranges x cells tile overlap matrix

Usage

```
tile_matrix(
  fragments,
  ranges,
  mode = c("insertions", "fragments"),
  zero_based_coords = !is(ranges, "GRanges"),
  explicit_tile_names = FALSE
)
```

Arguments

`fragments`	Input fragments object
`ranges`	Tiled regions given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position • `tile_width`: Size of each tile in this region in basepairs Must be non-overlapping and sorted by (chr, start), with chromosomes ordered according to the chromosome names of `fragments`
`mode`	Mode for counting tile overlaps. (See "value" section for more detail)
`zero_based_coords`	Whether to convert the ranges from a 1-based end-inclusive coordinate system to a 0-based end-exclusive coordinate system. Defaults to true for GRanges and false for other formats (see this archived UCSC blogpost)
`explicit_tile_names`	Boolean for whether to add rownames to the output matrix in format e.g chr1:500-1000, where start and end coords are given in a 0-based coordinate system. For whole-genome Tile matrices the names will take ~5 seconds to generate and take up 400MB of memory. Note that either way, tile names will be written when the matrix is saved.

Value

Iterable matrix object with dimension ranges x cells. When saved, the column names will be in the format chr1:500-1000, where start and end coords are given in a 0-based coordinate system.

`mode` **options**

- `"insertions"`: Start and end coordinates are separately overlapped with each tile
- `"fragments"`: Like `"insertions"`, but each fragment can contribute at most 1 count to each tile, even if both the start and end coordinates overlap

Note

When calculating the matrix directly from a fragments tsv, it's necessary to first call `select_chromosomes()` in order to provide the ordering of chromosomes to expect while reading the tsv.

Examples

```
## Prep demo data
frags <- get_demo_frags(subset = FALSE)
chrom_sizes <- read_ucsc_chrom_sizes(file.path(tempdir(), "references"), genome="hg38")
blacklist <- read_encode_blacklist(file.path(tempdir(), "references"), genome="hg38")
frags_filter_blacklist <- frags %>% select_regions(blacklist, invert_selection = TRUE)
ranges <- tibble::tibble(
  chr = "chr4",
  start = 0,
  end = "190214555",
  tile_width = 200
)


## Get tile matrix
tile_matrix(frags_filter_blacklist, ranges)
```

---

trackplot_combine *Combine track plots*

Description

Combines multiple track plots of the same region into a single grid. Uses the `patchwork` package to perform the alignment.

Usage

```
trackplot_combine(
  tracks,
  side_plot = NULL,
  title = NULL,
  side_plot_width = 0.3
)
```

Arguments

`tracks`	List of tracks in order from top to bottom, generally ggplots as output from the other `trackplot_*()` functions.
`side_plot`	Optional plot to align to the right (e.g. RNA expression per cluster). Will be aligned to the first `trackplot_coverage()` output if present, or else the first generic ggplot in the alignment. Should be in horizontal orientation and in the same cluster ordering as the coverage plots.
`title`	Text for overarching title of the plot
`side_plot_width`	Fraction of width that should be used for the side plot relative to the main track area

Value

A plot object with aligned genome plots. Each aligned row has the text label, y-axis, and plot body. The relative height of each row is given by heights. A shared title and x-axis are put at the top.

See Also

`trackplot_coverage()`, `trackplot_gene()`, `trackplot_loop()`, `trackplot_scalebar()`

Examples

```
## Prep data
frags <- get_demo_frags()

## Use genes and blacklist to determine proper number of reads per cell
genes <- read_gencode_transcripts(
  file.path(tempdir(), "references"), release = "42",
  annotation_set = "basic",
  features = "transcript"
)
blacklist <- read_encode_blacklist(file.path(tempdir(), "references"), genome="hg38")
read_counts <- qc_scATAC(frags, genes, blacklist)$nFrags
region <- "chr4:3034877-4034877"
cell_types <- paste("Group", rep(1:3, length.out = length(cellNames(frags))))
transcripts <- read_gencode_transcripts(
  file.path(tempdir(), "references"), release = "42",
  annotation_set = "basic"
)
region <- "chr4:3034877-4034877"


## Get all trackplots and scalebars to combine
plot_scalebar <- trackplot_scalebar(region)
plot_gene <- trackplot_gene(transcripts, region)
plot_coverage <- trackplot_coverage(frags, region, groups = cell_types, cell_read_counts = read_counts)


## Combine trackplots and render
## Also remove colors from gene track
plot <- trackplot_combine(
    list(plot_scalebar, plot_coverage, plot_gene + ggplot2::guides(color = "none"))
)
BPCells:::render_plot_from_storage(plot, width = 6, height = 4)
```

---

trackplot_coverage *Pseudobulk coverage trackplot*

Description

Plot a pseudobulk genome track, showing the number of fragment insertions across a region for each cell type or group.

Usage

```
trackplot_coverage(
  fragments,
  region,
  groups,
  cell_read_counts,
  group_order = NULL,
  bins = 500,
  clip_quantile = 0.999,
  colors = discrete_palette("stallion"),
  legend_label = NULL,
  zero_based_coords = !is(region, "GRanges"),
  return_data = FALSE
)
```

Arguments

fragments	Fragments object
region	Region to plot, e.g. output from gene_region(). String of format "chr1:100-200", or list/data.frame/GRanges of length 1 specifying chr, start, end. See help("genomic-ranges-like") for details
groups	Vector with one entry per cell, specifying the cell's group
cell_read_counts	Numeric vector of read counts for each cell (used for normalization)
group_order	Optional vector listing ordering of groups
bins	Number of bins to plot across the region
clip_quantile	(optional) Quantile of values for clipping y-axis limits. Default of 0.999 will crop out just the most extreme outliers across the region. NULL to disable clipping
colors	Character vector of color values (optionally named by group)
legend_label	**[Deprecated]** Custom label to put on the legend (no longer used as color legend is not shown anymore)
zero_based_coords	Whether to convert the ranges from a 1-based end-inclusive coordinate system to a 0-based end-exclusive coordinate system. Defaults to true for GRanges and false for other formats (see this archived UCSC blogpost)
return_data	If true, return data from just before plotting rather than a plot.
scale_bar	Whether to include a scale bar in the top track (TRUE or FALSE)

Value

Returns a combined plot of pseudobulk genome tracks. For compatability with draw_trackplot_grid(), the extra attribute $patches$labels will be added to specify the labels for each track. If return_data or return_plot_list is TRUE, the return value will be modified accordingly.

See Also

trackplot_combine(), trackplot_gene(), trackplot_loop(), trackplot_scalebar()

Examples

```
frags <- get_demo_frags()

## Use genes and blacklist to determine proper number of reads per cell
genes <- read_gencode_transcripts(
  file.path(tempdir(), "references"), release = "42",
  annotation_set = "basic",
  features = "transcript"
)
blacklist <- read_encode_blacklist(file.path(tempdir(), "references"), genome="hg38")
read_counts <- qc_scATAC(frags, genes, blacklist)$nFrags
region <- "chr4:3034877-4034877"
cell_types <- paste("Group", rep(1:3, length.out = length(cellNames(frags))))


BPCells:::render_plot_from_storage(
  trackplot_coverage(frags, region, groups = cell_types, cell_read_counts = read_counts),
  width = 6, height = 3
)
```

---

trackplot_gene *Plot transcript models*

Description

Plot transcript models

Usage

```
trackplot_gene(
  transcripts,
  region,
  exon_size = 2.5,
  gene_size = 0.5,
  label_size = 11 * 0.8/ggplot2::.pt,
  track_label = "Genes",
  return_data = FALSE
)
```

Arguments

transcripts	Transcipt features given as GRanges, data.frame, or list. See help("genomic-ranges-like") for details on format and coordinate systems. Required attributes: • chr, start, end: genomic position

- `strand`: +/- or TRUE/FALSE for positive or negative strand
- `feature`: Only entries marked as `"transcript"` or `"exon"` will be considered
- `gene_name`: Symbol or gene ID to display
- `transcript_id`: Transcritp identifier to link transcripts and exons

Usually given as the output from `read_gencode_transcripts()`

`region` Region to plot, e.g. output from `gene_region()`. String of format "chr1:100-200", or list/data.frame/GRanges of length 1 specifying chr, start, end. See `help("genomic-ranges-like")` for details

`exon_size` size for exon lines in units of mm

`gene_size` size for intron/gene lines in units of mm

`label_size` size for transcript labels in units of mm

`return_data` If true, return data from just before plotting rather than a plot.

`labels` Character vector with labels for each item in transcripts. NA for items that should not be labeled

`transcript_size` size for transcript lines in units of mm

Value

Plot of gene locations

See Also

`trackplot_combine()`, `trackplot_coverage()`, `trackplot_loop()`, `trackplot_scalebar()`

Examples

```
## Prep data
transcripts <- read_gencode_transcripts(
  file.path(tempdir(), "references"), release = "42",
  annotation_set = "basic", features = "transcript"
)
region <- "chr4:3034877-4034877"


## Plot gene trackplot
plot <- trackplot_gene(transcripts, region)
BPCells:::render_plot_from_storage(plot, width = 6, height = 1)
```

trackplot_genome_annotation *Plot range-based annotation tracks (e.g. peaks)*

Description

Plot range-based annotation tracks (e.g. peaks)

Usage

```
trackplot_genome_annotation(
  loci,
  region,
  color_by = NULL,
  colors = NULL,
  label_by = NULL,
  label_size = 11 * 0.8/ggplot2::.pt,
  show_strand = FALSE,
  annotation_size = 2.5,
  track_label = "Peaks",
  return_data = FALSE
)
```

Arguments

loci	Genomic loci given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position
region	Region to plot, e.g. output from `gene_region()`. String of format "chr1:100-200", or list/data.frame/GRanges of length 1 specifying chr, start, end. See `help("genomic-ranges-like")` for details
color_by	Name of a metadata column in `loci` to use for coloring, or a data vector with same length as loci. Column must be numeric or convertible to a factor.
colors	Vector of hex color codes to use for the color scale. For numeric `color_by` data, this is passed to `ggplot2::scale_color_gradientn()`, otherwise it is interpreted as a discrete color palette in `ggplot2::scale_color_manual()`
label_by	Name of a metadata column in `loci` to use for labeling, or a data vector with same length as loci. Column must hold string data.
label_size	size for labels in units of mm
show_strand	If TRUE, show strand direction as arrows
annotation_size	size for annotation lines in mm
return_data	If true, return data from just before plotting rather than a plot.

Value

Plot of genomic loci if return_data is FALSE, otherwise returns the data frame used to generate the plot

See Also

trackplot_combine(), trackplot_coverage(), trackplot_loop(), trackplot_scalebar(), trackplot_gene()

Examples

```
## Prep data
## Peaks generated from demo frags, as input into `call_peaks_tile()`
peaks <- tibble::tibble(
  chr = factor(rep("chr4", 16)),
  start = c(3041400, 3041733, 3037400, 3041933, 3040466, 3041200,
            3038200, 3038000, 3040266, 3037733, 3040800, 3042133,
            3038466, 3037200, 3043333, 3040066),
  end = c(3041600, 3041933, 3037600, 3042133, 3040666, 3041400,
          3038400, 3038200, 3040466, 3037933, 3041000, 3042333,
          3038666, 3037400, 3043533, 3040266),
  enrichment = c(46.4, 43.5, 28.4, 27.3, 17.3, 11.7,
                 10.5, 7.95, 7.22, 6.86, 6.32, 6.14,
                 5.96, 5.06, 4.51, 3.43)
)
region <- "chr4:3034877-3044877"

## Plot peaks
BPCells:::render_plot_from_storage(
  trackplot_genome_annotation(peaks, region, color_by = "enrichment"),
  width = 6, height = 1
)
```

---

trackplot_loop *Plot loops*

Description

Plot loops

Usage

```
trackplot_loop(
  loops,
  region,
  color_by = NULL,
  colors = NULL,
  allow_truncated = TRUE,
  curvature = 0.75,
```

```
  track_label = "Links",
  return_data = FALSE
)
```

Arguments

loops	Genomic regions given as GRanges, data.frame, or list. See `help("genomic-ranges-like")` for details on format and coordinate systems. Required attributes: • `chr`, `start`, `end`: genomic position
region	Region to plot, e.g. output from `gene_region()`. String of format "chr1:100-200", or list/data.frame/GRanges of length 1 specifying chr, start, end. See `help("genomic-ranges-like")` for details
color_by	Name of a metadata column in `loops` to use for coloring, or a data vector with same length as loci. Column must be numeric or convertible to a factor.
colors	Vector of hex color codes to use for the color scale. For numeric `color_by` data, this is passed to `ggplot2::scale_color_gradientn()`, otherwise it is interpreted as a discrete color palette in `ggplot2::scale_color_manual()`
allow_truncated	If FALSE, remove any loops that are not fully contained within `region`
curvature	Curvature value between 0 and 1. 1 is a 180-degree arc, and 0 is flat lines.
return_data	If true, return data from just before plotting rather than a plot.

Value

Plot of loops connecting genomic coordinates

See Also

`trackplot_combine()`, `trackplot_coverage()`, `trackplot_gene()`, `trackplot_scalebar()`, `trackplot_genome_annotation()`

Examples

```
peaks <- c(3054877, 3334877, 3534877, 3634877, 3734877)
loops <- tibble::tibble(
  chr = "chr4",
  start = peaks[c(1,1,2,3)],
  end = peaks[c(2,3,4,5)],
  score = c(4,1,3,2)
)
region <- "chr4:3034877-4034877"

## Plot loops
plot <- trackplot_loop(loops, region, color_by = "score")
BPCells:::render_plot_from_storage(plot, width = 6, height = 1.5)
```

trackplot_scalebar *Plot scale bar*

Description

Plots a human-readable scale bar and coordinates of the region being plotted

Usage

```
trackplot_scalebar(region, font_pt = 11)
```

Arguments

region	Region to plot, e.g. output from gene_region(). String of format "chr1:100-200", or list/data.frame/GRanges of length 1 specifying chr, start, end. See help("genomic-ranges-like") for details
font_pt	Font size for scale bar labels in units of pt.

Value

Plot with coordinates and scalebar for plotted genomic region

See Also

trackplot_combine(), trackplot_coverage(), trackplot_gene(), trackplot_loop()

Examples

```
region <- "chr4:3034877-3044877"
BPCells:::render_plot_from_storage(
  trackplot_scalebar(region), width = 6, height = 1
)
```

transpose_storage_order *Transpose the storage order for a matrix*

Description

Transpose the storage order for a matrix

Usage

```
transpose_storage_order(
  matrix,
  outdir = tempfile("transpose"),
  tmpdir = tempdir(),
  load_bytes = 4194304L,
  sort_bytes = 1073741824L
)
```

Arguments

matrix	Input matrix
outdir	Directory to store the output
tmpdir	Temporary directory to use for intermediate storage
load_bytes	The minimum contiguous load size during the merge sort passes
sort_bytes	The amount of memory to allocate for re-sorting chunks of entries

Details

This re-sorts the entries of a matrix to change the storage order from row-major to col-major. For large matrices, this can be slow – around 2 minutes to transpose a 500k cell RNA-seq matrix The default load_bytes (4MiB) and sort_bytes (1GiB) parameters allow ~85GB of data to be sorted with two passes through the data, and ~7.3TB of data to be sorted in three passes through the data.

Value

MatrixDir object with a copy of the input matrix, but the storage order flipped

Examples

```
mat <- matrix(rnorm(50), nrow = 10, ncol = 5)
rownames(mat) <- paste0("gene", seq_len(10))
colnames(mat) <- paste0("cell", seq_len(5))
mat <- mat %>% as("dgCMatrix") %>% as("IterableMatrix")
mat

## A regular transpose operation switches a user's rows and cols
t(mat)

## Running `transpose_storage_order()` instead changes whether the storage is in row-major or col-major,
## but does not switch the rows and cols
transpose_storage_order(mat)
```

write_fragments_memory *Read/write BPCells fragment objects*

Description

BPCells fragments can be read/written in compressed (bitpacked) or uncompressed form in a variety of storage locations: in memory (as an R object), in an hdf5 file, or in a directory on disk (containing binary files).

Usage

```
write_fragments_memory(fragments, compress = TRUE)

write_fragments_dir(
  fragments,
  dir,
  compress = TRUE,
  buffer_size = 1024L,
  overwrite = FALSE
)

open_fragments_dir(dir, buffer_size = 1024L)

write_fragments_hdf5(
  fragments,
  path,
  group = "fragments",
  compress = TRUE,
  buffer_size = 8192L,
  chunk_size = 1024L,
  overwrite = FALSE,
  gzip_level = 0L
)

open_fragments_hdf5(path, group = "fragments", buffer_size = 16384L)
```

Arguments

fragments	Input fragments object
compress	Whether or not to compress the data. With compression, storage size is be about half the size of a gzip-compressed 10x fragments file.
dir	Directory to read/write the data from
buffer_size	For performance tuning only. The number of items to be bufferred in memory before calling writes to disk.

overwrite	If TRUE, write to a temp dir then overwrite existing data. Alternatively, pass a temp path as a string to customize the temp dir location.
path	Path to the hdf5 file on disk
group	The group within the hdf5 file to write the data to. If writing to an existing hdf5 file this group must not already be in use
chunk_size	For performance tuning only. The chunk size used for the HDF5 array storage.
gzip_level	Gzip compression level. Default is 0 (no compression). This is recommended when both compression and compatibility with outside programs is required. Otherwise, using compress=TRUE is recommended as it is >10x faster with often similar compression levels.

Details

Saving in a directory on disk is a good default for local analysis, as it provides the best I/O performance and lowest memory usage. The HDF5 format allows saving within existing hdf5 files to group data together, and the in memory format provides the fastest performance in the event memory usage is unimportant.

Value

Fragment object

Examples

```
## Create temporary directory to keep demo fragments
data_dir <- file.path(tempdir(), "frags")
dir.create(data_dir, recursive = TRUE, showWarnings = FALSE)
## Get demo frags loaded from disk
frags <- get_demo_frags()
frags

#########################################################################
## write_fragments_memory() example
#########################################################################
frags_memory <- write_fragments_memory(frags)
frags_memory


#########################################################################
## write_fragments_dir() example
#########################################################################
frags <- write_fragments_dir(
 frags_memory,
 file.path(data_dir, "demo_frags"),
 overwrite = TRUE
)
frags


#########################################################################
```

```
## open_fragments_dir() example
################################################################################
frags <- open_fragments_dir(file.path(data_dir, "demo_frags"))
frags


################################################################################
## write_fragments_hdf5() example
################################################################################
frags_hdf5 <- write_fragments_hdf5(
 frags,
 file.path(data_dir, "demo_frags.h5"),
 overwrite = TRUE
)
frags_hdf5


################################################################################
## open_fragments_hdf5() example
################################################################################
frags_hdf5 <- open_fragments_hdf5(file.path(data_dir, "demo_frags.h5"))
frags_hdf5
```

`write_insertion_bedgraph`

Write insertion counts to bed/bedgraph file

Description

Write insertion counts data for one or more pseudobulks to bed/bedgraph format. Beds only hold chrom, start, and end data, while bedGraphs also provide a score column. This reports the total number of insertions at each basepair for each group listed in `cell_groups`.

Usage

```
write_insertion_bedgraph(
  fragments,
  path,
  cell_groups = rlang::rep_along(cellNames(fragments), "all"),
  insertion_mode = c("both", "start_only", "end_only"),
  tile_width = 1,
  normalization_method = c("none", "cpm", "n_cells"),
  chrom_sizes = NULL
)

write_insertion_bed(
  fragments,
```

```
  path,
  cell_groups = rlang::rep_along(cellNames(fragments), "all"),
  insertion_mode = c("both", "start_only", "end_only"),
  verbose = FALSE,
  threads = 1
)
```

Arguments

fragments	IterableFragments object
path	(character vector) Path(s) to save bed/bedgraphs to, optionally ending in ".gz" to add gzip compression. If cell_groups is provided, path must be a named character vector, with one name for each level in cell_groups
cell_groups	Character or factor assigning a group to each cell, in order of cellNames(fragments)
insertion_mode	(string) Which fragment ends to use for coverage calculation. One of "both", "start_only", or "end_only"
tile_width	(integer) Width of tiles to use for binning insertions. All insertions in a single bin are summed. If tile_width is 1, then this is functionally equivalent to write_insertion_bedgraph().
normalization_method	(character) Normalization method to use. One of: • none: No normalization • cpm: Normalize by total number of fragments in each group, scaling to 1 million fragments (i.e. CPM). • n_cells: Normalize by total number of cells in each group.
chrom_sizes	(GRanges, data.frame, list, numeric, or NULL) Chromosome sizes to clip tiles when at the end of a chromosome. If NULL, then tile_width is required to be 1. If a data.frame or list, must contain columns chr and end (See help("genomic-ranges-like")). If a numeric vector, then it is assumed to be the chromosome sizes in the order of chrNames(fragments).
verbose	(bool) Whether to provide verbose progress output to console.
threads	(int) Number of threads to use.

Value

NULL

Examples

```
## Prep data
frags <- get_demo_frags()
bedgraph_outputs <- file.path(tempdir(), "bedgraph_outputs")

#####################################################
## `write_insertion_bedgraph()` examples
#####################################################
## Write insertions
```

```
write_insertion_bedgraph(frags, file.path(bedgraph_outputs, "all.tar.gz"))
list.files(bedgraph_outputs)

# With tiling
chrom_sizes <- read_ucsc_chrom_sizes("./reference", genome="hg38") %>%
  dplyr::filter(chr %in% c("chr4", "chr11"))
write_insertion_bedgraph(frags, file.path(bedgraph_outputs, "all_tiled.bedGraph"),
  chrom_sizes = chrom_sizes, normalization_method = "cpm", tile_width = 100)
reads <- readr::read_tsv(file.path(bedgraph_outputs, "all_tiled.bedGraph"),
  col_names = c("chr", "start", "end", "score"),
  show_col_types = FALSE)
head(reads)


####################################################
## `write_insertion_bed()` examples
####################################################

# We utilize two groups this time
bed_outputs <- file.path(tempdir(), "bed_outputs")
cell_groups <- rep(c("A", "B"), length.out = length(cellNames(frags)))
bed_paths <- c(file.path(bed_outputs, "A.bed"), file.path(bed_outputs, "B.bed"))
names(bed_paths) <- c("A", "B")
write_insertion_bed(
  frags, path = bed_paths, cell_groups = cell_groups,
  verbose = TRUE
)
list.files(bed_outputs)
head(readr::read_tsv(
  file.path(bed_outputs, "A.bed"),
  col_names = c("chr", "start", "end"), show_col_types = FALSE)
)
```

`write_matrix_memory` *Read/write sparse matrices*

Description

BPCells matrices are stored in sparse format, meaning only the non-zero entries are stored. Matrices can store integer counts data or decimal numbers (float or double). See details for more information.

Usage

```
write_matrix_memory(mat, compress = TRUE)

write_matrix_dir(
  mat,
  dir,
  compress = TRUE,
  buffer_size = 8192L,
```

```
  overwrite = FALSE
)

open_matrix_dir(dir, buffer_size = 8192L)

write_matrix_hdf5(
  mat,
  path,
  group,
  compress = TRUE,
  buffer_size = 8192L,
  chunk_size = 1024L,
  overwrite = FALSE,
  gzip_level = 0L
)

open_matrix_hdf5(path, group, buffer_size = 16384L)
```

Arguments

compress	Whether or not to compress the data.
dir	Directory to save the data into
buffer_size	For performance tuning only. The number of items to be buffered in memory before calling writes to disk.
overwrite	If TRUE, write to a temp dir then overwrite existing data. Alternatively, pass a temp path as a string to customize the temp dir location.
path	Path to the hdf5 file on disk
group	The group within the hdf5 file to write the data to. If writing to an existing hdf5 file this group must not already be in use
chunk_size	For performance tuning only. The chunk size used for the HDF5 array storage.
gzip_level	Gzip compression level. Default is 0 (no compression). This is recommended when both compression and compatibility with outside programs is required. Otherwise, using compress=TRUE is recommended as it is >10x faster with often similar compression levels.
matrix	Input matrix, either IterableMatrix or dgCMatrix

Details

Storage locations:

Matrices can be stored in a directory on disk, in memory, or in an HDF5 file. Saving in a directory on disk is a good default for local analysis, as it provides the best I/O performance and lowest memory usage. The HDF5 format allows saving within existing hdf5 files to group data together, and the in memory format provides the fastest performance in the event memory usage is unimportant.

Bitpacking Compression:

For typical RNA counts matrices holding integer counts, this bitpacking compression will result in 6-8x less space than an R dgCMatrix, and 4-6x smaller than a scipy csc_matrix. The compression will be more effective when the count values in the matrix are small, and when the rows of the matrix are sorted by rowMeans. In tests on RNA-seq data optimal ordering could save up to 40% of storage space. On non-integer data only the row indices are compressed, not the values themselves so space savings will be smaller.

For non-integer data matrices, bitpacking compression is much less effective, as it can only be applied to the indexes of each entry but not the values. There will still be some space savings, but far less than for counts matrices.

Value

BPCells matrix object

Examples

```
## Create temporary directory to keep demo matrix
data_dir <- file.path(tempdir(), "mat")
if (dir.exists(data_dir)) unlink(data_dir, recursive = TRUE)
dir.create(data_dir, recursive = TRUE, showWarnings = FALSE)

mat <- get_demo_mat()
mat

########################################################################
## write_matrix_memory() example
########################################################################
mat_memory <- write_matrix_memory(mat)
mat_memory


########################################################################
## write_matrix_dir() example
########################################################################
mat %>% write_matrix_dir(
 file.path(data_dir, "demo_mat"),
 overwrite = TRUE
)


########################################################################
## open_matrix_dir() example
########################################################################
mat <- open_matrix_dir(
 file.path(data_dir, "demo_mat")
)
mat


########################################################################
## write_matrix_hdf5() example
########################################################################
```

```
mat %>% write_matrix_hdf5(path = file.path(data_dir, "demo_mat.h5"), group = "mat")


########################################################################
## open_matrix_hdf5() example
########################################################################
mat_hdf5 <- open_matrix_hdf5(
 file.path(data_dir, "demo_mat.h5"),
 group = 'mat'
)
mat_hdf5
```

Index