Package ‘scater’

October 16, 2018

Type Package

Maintainer Davis McCarthy <davis@ebi.ac.uk>

Version 1.8.4

Date 2018-08-13

License GPL (>= 2)

Title Single-Cell Analysis Toolkit for Gene Expression Data in R

Description A collection of tools for doing various analyses of single-cell RNA-seq gene expression data, with a focus on quality control.

Depends R (>= 3.5), Biobase, ggplot2, SingleCellExperiment, SummarizedExperiment

Imports BiocGenerics, data.table, dplyr, edgeR, ggbeeswarm, grid, limma, Matrix, DelayedMatrixStats, methods, parallel, plyr, reshape2, rhdf5, rjson, S4Vectors, shiny, shinydashboard, stats, tximport, utils, viridis, Rcpp (>= 0.12.14), DelayedArray

Suggests BiocStyle, biomaRt, beachmat, cowplot, cluster, destiny, knitr, monocle, mvoutlier, rmarkdown, Rtsne, testthat, magrittr, pheatmap, DropletUtils, irlba

VignetteBuilder knitr

LazyData true

biocViews SingleCell, RNASeq, QualityControl, Preprocessing, Normalization, Visualization, DimensionReduction, Transcriptomics, GeneExpression, Sequencing, Software, DataImport, DataRepresentation, Infrastructure, Coverage

LinkingTo Rhdf5lib, Rcpp, beachmat

SystemRequirements C++11

RoxygenNote 6.0.1

NeedsCompilation yes

URL http://bioconductor.org/packages/scater/

BugReports https://support.bioconductor.org/

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

git_branch RELEASE_3_7

git_last_commit d560a9a

git_last_commit_date 2018-08-13

Date/Publication 2018-10-15

Author Davis McCarthy [aut, cre],
Kieran Campbell [aut],
Aaron Lun [aut, ctb],
Quin Wills [aut],
Vladimir Kiselev [ctb]

R topics documented:

`scater-package` *Single-cell analysis toolkit for expression in R*

Description

scater provides a class and numerous functions for the quality control, normalisation and visualisation of single-cell RNA-seq expression data.

Details

In particular, **scater** provides easy generation of quality control metrics and simple functions to visualise quality control metrics and their relationships.

`areSizeFactorsCentred` *Check if the size factors are centred at unity*

Description

Checks if each set of size factors is centred at unity, such that abundances can be reasonably compared between features normalized with different sets of size factors.

Usage

```
areSizeFactorsCentred(object, centre = 1, tol = 1e-06)
```

Arguments

`object`	A SingleCellExperiment object containing any number of (or zero) sets of size factors.
`centre`	a numeric scalar, the value around which all sets of size factors should be centred.
`tol`	a numeric scalar, the tolerance for testing equality of the mean of each size factor set to `centre`.

Value

A logical scalar indicating whether all sets of size factors are centered. If no size factors are available, `TRUE` is returned.

Author(s)

Aaron Lun

See Also

`centreSizeFactors`

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)

sizeFactors(example_sce) <- runif(ncol(example_sce))
areSizeFactorsCentred(example_sce)
example_sce <- normalize(example_sce, centre = TRUE)
areSizeFactorsCentred(example_sce)
```

---

arrange — *Arrange columns (cells) of a SingleCellExperiment object*

Description

The `SingleCellExperiment` returned will have cells ordered by the corresponding variable in `colData(object)`.

Usage

```
arrange(object, ...)

## S4 method for signature 'SingleCellExperiment'
arrange(object, ...)
```

Arguments

`object`	A `SingleCellExperiment` object.
`...`	Additional arguments to be passed to `dplyr::arrange` to act on `colData(object)`.

Value

An SingleCellExperiment object.

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts),
colData = sc_example_cell_info)
example_sce <- arrange(example_sce, Cell_Cycle)
```

`bootstraps` *Accessor and replacement for bootstrap results in a* `SingleCellExperiment` *object*

Description

`SingleCellExperiment` objects can contain bootstrap expression values (for example, as generated by the kallisto software for quantifying feature abundance). These functions conveniently access and replace the 'bootstrap' elements in the `assays` slot with the value supplied, which must be an matrix of the correct size, namely the same number of rows and columns as the `SingleCellExperiment` object as a whole.

Usage

```
bootstraps(object)

bootstraps(object) <- value

## S4 method for signature 'SingleCellExperiment'
bootstraps(object)

## S4 replacement method for signature 'SingleCellExperiment,array'
bootstraps(object) <- value
```

Arguments

`object`	a `SingleCellExperiment` object.
`value`	an array of class `"numeric"` containing bootstrap expression values

Value

If accessing bootstraps slot of an `SingleCellExperiment`, then an array with the bootstrap values, otherwise an `SingleCellExperiment` object containing new bootstrap values.

Author(s)

Davis McCarthy

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts), colData = sc_example_cell_info)
bootstraps(example_sce)
```

---

calcAverage — *Calculate average counts, adjusting for size factors or library size*

Description

Calculate average counts per feature, adjusting them as appropriate to take into account for size factors for normalization or library sizes (total counts).

Usage

```
calcAverage(object, exprs_values = "counts", use_size_factors = TRUE,
  size_factor_grouping = NULL, subset_row = NULL)
```

Arguments

`object`	A SingleCellExperiment object or count matrix.
`exprs_values`	A string specifying the assay of `object` containing the count matrix, if `object` is a SingleCellExperiment.
`use_size_factors`	a logical scalar specifying whetherthe size factors in `object` should be used to construct effective library sizes.
`size_factor_grouping`	A factor to be passed to `grouping=` in `centreSizeFactors`.
`subset_row`	A vector specifying whether the rows of `object` should be (effectively) subsetted before calcaulting feature averages.

Details

The size-adjusted average count is defined by dividing each count by the size factor and taking the average across cells. All sizes factors are scaled so that the mean is 1 across all cells, to ensure that the averages are interpretable on the scale of the raw counts.

Assuming that `object` is a SingleCellExperiment:

- If `use_size_factors=TRUE`, size factors are automatically extracted from the object. Note that different size factors may be used for features marked as spike-in controls. This is due to the presence of control-specific size factors in `object`, see `normalizeSCE` for more details.
- If `use_size_factors=FALSE`, all size factors in `object` are ignored. Size factors are instead computed from the library sizes, using `librarySizeFactors`.
- If `use_size_factors` is a numeric vector, it will override the any size factors for non-spike-in features in `object`. The spike-in size factors will still be used for the spike-in transcripts.

If no size factors are available, they will be computed from the library sizes using `librarySizeFactors`.

If `object` is a matrix or matrix-like object, size factors can be supplied by setting `use_size_factors` to a numeric vector. Otherwise, the sum of counts for each cell is used as the size factor through `librarySizeFactors`.

Value

Vector of average count values with same length as number of features, or the number of features in `subset_row` if supplied.

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
   list(counts = sc_example_counts),
   colData = sc_example_cell_info)

## calculate average counts
ave_counts <- calcAverage(example_sce)
```

`calcIsExprs` *Calculate which features are expressed in which cells using a threshold on observed counts, transcripts-per-million, counts-per-million, FPKM, or defined expression levels.*

Description

Calculate which features are expressed in which cells using a threshold on observed counts, transcripts-per-million, counts-per-million, FPKM, or defined expression levels.

Usage

```
calcIsExprs(object, detection_limit = 0, exprs_values = "counts")
```

Arguments

`object` a `SingleCellExperiment` object with expression and/or count data.

`detection_limit`
numeric scalar giving the minimum expression level for an expression observation in a cell for it to qualify as expressed.

`exprs_values` character scalar indicating whether the count data (`"counts"`), the log-transformed count data (`"logcounts"`), transcript-per-million (`"tpm"`), counts-per-million (`"cpm"`) or FPKM (`"fpkm"`) should be used to define if an observation is expressed or not. Defaults to the first available value of those options in the order shown.

Value

a logical matrix indicating whether or not a feature in a particular cell is expressed.

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts), colData = sc_example_cell_info)
assay(example_sce, "is_exprs") <- calcIsExprs(example_sce,
detection_limit = 1, exprs_values = "counts")
```

---

calculateCPM *Calculate counts per million (CPM)*

Description

Calculate count-per-million (CPM) values from the count data.

Usage

```
calculateCPM(object, exprs_values = "counts", use_size_factors = TRUE,
  size_factor_grouping = NULL, subset_row = NULL)
```

Arguments

`object`	A SingleCellExperiment object or count matrix.
`exprs_values`	A string specifying the assay of `object` containing the count matrix, if `object` is a SingleCellExperiment.
`use_size_factors`	A logical scalar indicating whether size factors in `object` should be used to compute effective library sizes. If not, all size factors are deleted and library size-based factors are used instead (see `librarySizeFactors`. Alternatively, a numeric vector containing a size factor for each cell, which is used in place of `sizeFactor(object)`.
`size_factor_grouping`	A factor to be passed to `grouping=` in `centreSizeFactors`.
`subset_row`	A vector specifying whether the rows of `object` should be (effectively) subsetted before calcaulting feature averages.

Details

If requested, size factors are used to define the effective library sizes. This is done by scaling all size factors such that the mean scaled size factor is equal to the mean sum of counts across all features. The effective library sizes are then used to in the denominator of the CPM calculation.

Assuming that `object` is a SingleCellExperiment:

- If `use_size_factors=TRUE`, size factors are automatically extracted from the object. Note that effective library sizes may be computed differently for features marked as spike-in controls. This is due to the presence of control-specific size factors in `object`, see `normalizeSCE` for more details.
- If `use_size_factors=FALSE`, all size factors in `object` are ignored. The total count for each cell will be used as the library size for all features (endogenous genes and spike-in controls).

- If `use_size_factors` is a numeric vector, it will override the any size factors for non-spike-in features in `object`. The spike-in size factors will still be used for the spike-in transcripts.

If no size factors are available, the library sizes will be used.

If `object` is a matrix or matrix-like object, size factors will only be used if `use_size_factors` is a numeric vector. Otherwise, the sum of counts for each cell is directly used as the library size.

Note that the rescaling is performed to the mean sum of counts for all features, regardless of whether `subset.row` is specified. This ensures that the output of the function with `subset.row` is equivalent (but more efficient) than subsetting the output of the function without `subset.row`.

Value

Matrix of CPM values.

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    list(counts = sc_example_counts),
    colData = sc_example_cell_info)

cpm(example_sce) <- calculateCPM(example_sce, use_size_factors = FALSE)
```

---

calculateFPKM — *Calculate fragments per kilobase of exon per million reads mapped (FPKM)*

Description

Calculate fragments per kilobase of exon per million reads mapped (FPKM) values for expression from counts for a set of features.

Usage

```
calculateFPKM(object, effective_length, ...)
```

Arguments

`object`	an `SingleCellExperiment` object
`effective_length`	vector of class `"numeric"` providing the effective length for each feature in the `SCESet` object
`...`	Further arguments to pass to `calculateCPM`.

Value

Matrix of FPKM values.

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts), colData = sc_example_cell_info)
effective_length <- rep(1000, 2000)
fpkm(example_sce) <- calculateFPKM(example_sce, effective_length,
use_size_factors = FALSE)
```

---

calculateQCMetrics *Calculate QC metrics*

Description

Compute quality control (QC) metrics for each feature and cell in a SingleCellExperiment object, accounting for specified control sets.

Usage

```
calculateQCMetrics(object, exprs_values = "counts", feature_controls = NULL,
  cell_controls = NULL, percent_top = c(50, 100, 200, 500),
  detection_limit = 0, use_spikes = TRUE, compact = FALSE)
```

Arguments

`object` A SingleCellExperiment object containing expression values, usually counts.

`exprs_values` A string indicating which assays in the `object` should be used to define expression.

`feature_controls` A named list containing one or more vectors (a character vector of feature names, a logical vector, or a numeric vector of indices), used to identify feature controls such as ERCC spike-in sets or mitochondrial genes.

`cell_controls` A named list containing one or more vectors (a character vector of cell (sample) names, a logical vector, or a numeric vector of indices), used to identify cell controls, e.g., blank wells or bulk controls.

`percent_top` An integer vector. Each element is treated as a number of top genes to compute the percentage of library size occupied by the most highly expressed genes in each cell. See `pct_X_top_Y_features` below for more details.

`detection_limit` A numeric scalar to be passed to `nexprs`, specifying the lower detection limit for expression.

`use_spikes` A logical scalar indicating whether existing spike-in sets in `object` should be automatically added to `feature_controls`, see `?isSpike`.

`compact` A logical scalar indicating whether the metrics should be returned in a compact format as a nested DataFrame.

Details

This function calculates useful quality control metrics to help with pre-processing of data and identification of potentially problematic features and cells.

Underscores in `assayNames(object)` and in `feature_controls` or `cell_controls` can cause theoretically cause ambiguities in the names of the output metrics. While problems are highly unlikely, users are advised to avoid underscores when naming their controls/assays.

Value

A SingleCellExperiment object containing QC metrics in the row and column metadata.

Cell-level QC metrics

Denote the value of `exprs_values` as `X`. Cell-level metrics are:

- **`total_X`:** Sum of expression values for each cell (i.e., the library size, when counts are the expression values).
- **`log10_total_X`:** Log10-transformed `total_X` after adding a pseudo-count of 1.
- **`total_features_by_X`:** The number of features that have expression values above the detection limit.
- **`log10_total_features_by_X`:** Log10-transformed `total_features_by_X` after adding a pseudo-count of 1.
- **`pct_X_in_top_Y_features`:** The percentage of the total that is contained within the top `Y` most highly expressed features in each cell. This is only reported when there are more than `Y` features. The top numbers are specified via `percent_top`.

If any controls are specified in `feature_controls`, the above metrics will be recomputed using only the features in each control set. The name of the set is appended to the name of the recomputed metric, e.g., `total_X_F`. A `pct_X_F` metric is also calculated for each set, representing the percentage of expression values assigned to features in `F`.

In addition to the user-specified control sets, two other sets are automatically generated when `feature_controls` is non-empty. The first is the `"feature_control"` set, containing a union of all feature control sets; and the second is an `"endogenous"` set, containing all genes not in any control set. Metrics are also computed for these sets in the same manner described above, suffixed with `_feature_control` and `_endogenous` instead of `_F`.

Finally, there is the `is_cell_control` field, which indicates whether each cell has been defined as a cell control by `cell_controls`. If multiple sets of cell controls are defined (e.g., blanks or bulk libraries), a metric `is_cell_control_C` is produced for each cell control set `C`. The union of all sets is stored in `is_cell_control`.

All of these cell-level QC metrics are added as columns to the `colData` slot of the SingleCellExperiment object. This allows them to be inspected by the user and makes them readily available for other functions to use.

Feature-level QC metrics

Denote the value of `exprs_values` as `X`. Feature-level metrics are:

- **`mean_X`:** Mean expression value for each gene across all cells.
- **`log10_mean_X`:** Log10-mean expression value for each gene across all cells.
- **`n_cells_by_X`:** Number of cells with expression values above the detection limit for each gene.

pct_dropout_by_X: Percentage of cells with expression values below the detection limit for each gene.

total_X: Sum of expression values for each gene across all cells.

log10_total_X: Log10-sum of expression values for each gene across all cells.

If any controls are specified in cell_controls, the above metrics will be recomputed using only the cells in each control set. The name of the set is appended to the name of the recomputed metric, e.g., total_X_C. A pct_X_C metric is also calculated for each set, representing the percentage of expression values assigned to cells in C.

In addition to the user-specified control sets, two other sets are automatically generated when cell_controls is non-empty. The first is the "cell_control" set, containing a union of all cell control sets; and the second is an "non_control" set, containing all genes not in any control set. Metrics are computed for these sets in the same manner described above, suffixed with _cell_control and _non_control instead of_C.

Finally, there is the is_feature_control field, which indicates whether each feature has been defined as a control by feature_controls. If multiple sets of feature controls are defined (e.g., ERCCs, mitochondrial genes), a metric is_feature_control_F is produced for each feature control set F. The union of all sets is stored in is_feature_control.

These feature-level QC metrics are added as columns to the rowData slot of the SingleCellExperiment object. They can be inspected by the user and are readily available for other functions to use.

Compacted output

If compact=TRUE, the QC metrics are stored in the "scater_qc" field of the colData and rowData as a nested DataFrame. This avoids cluttering the metadata with QC metrics, especially if many results are to be stored in a single SingleCellExperiment object.

Assume we have a feature control set F and a cell control set C. The nesting structure in scater_qc in the colData is:

```
scater_qc
|-- is_cell_control
|-- is_cell_control_C
|-- all
|   |-- total_counts
|   |-- total_features_by_counts
|   \-- ...
+-- endogenous
|   |-- total_counts
|   |-- total_features_by_counts
    |-- pct_counts
|   \-- ...
+-- feature_control
|   |-- total_counts
|   |-- total_features_by_counts
    |-- pct_counts
|   \-- ...
\-- feature_control_F
    |-- total_counts
    |-- total_features_by_counts
    |-- pct_counts
    \-- ...
```

The nesting in `scater_qc` in the `rowData` is:

```
scater_qc
|-- is_feature_control
|-- is_feature_control_F
|-- all
|   |-- total_counts
|   |-- total_features_by_counts
|   \-- ...
+-- non_control
|   |-- total_counts
|   |-- total_features_by_counts
    |-- pct_counts
|   \-- ...
+-- cell_control
|   |-- total_counts
|   |-- total_features_by_counts
    |-- pct_counts
|   \-- ...
\-- cell_control_C
    |-- total_counts
    |-- total_features_by_counts
    |-- pct_counts
    \-- ...
```

No suffixing of the metric names by the control names is performed here. This is not necessary when each control set has its own nested DataFrame.

Renamed metrics

Several metric names have been changed in **scater** 1.7.5:

- `total_features` was changed to `total_features_by_X` where `X` is the `exprs_values`. This avoids ambiguities if `calculateQCMetrics` is called multiple times with different `exprs_values`.
- `n_cells_X` was changed to `n_cells_by_X`, to provide a more sensible name for the metric.
- `pct_dropout_X` was changed to `pct_dropout_by_X`.
- `pct_X_top_Y_features` was changed to `pct_X_in_top_Y_features`.

All of the old metric names will be kept alongside the new metric names when `compact=FALSE`. Otherwise, only the new metric names will be stored. The old metric names may be removed in future releases of **scater**.

Author(s)

Davis McCarthy, with (many!) modifications by Aaron Lun

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
```

```
)
example_sce <- calculateQCMetrics(example_sce)

## with a set of feature controls defined
example_sce <- calculateQCMetrics(example_sce,
feature_controls = list(set1 = 1:40))

## with a named set of feature controls defined
example_sce <- calculateQCMetrics(example_sce,
     feature_controls = list(ERCC = 1:40))
```

`calculateTPM` *Calculate transcripts-per-million (TPM)*

Description

Calculate transcripts-per-million (TPM) values for expression from counts for a set of features.

Usage

```
calculateTPM(object, effective_length = NULL, calc_from = "counts")
```

Arguments

`object`	a `SingleCellExperiment` object
`effective_length`	vector of class `"numeric"` providing the effective length for each feature in the `SingleCellExperiment` object
`calc_from`	character string indicating whether to compute TPM from `"counts"`, `"normcounts"` or `"fpkm"`. Default is to use `"counts"`, in which case the `effective_length` argument must be supplied.

Value

Matrix of TPM values.

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts), colData = sc_example_cell_info)
tpm(example_sce) <- calculateTPM(example_sce, effective_length = 5e04,
    calc_from = "counts")

## calculate from FPKM
fpkm(example_sce) <- calculateFPKM(example_sce, effective_length = 5e04,
use_size_factors = FALSE)
tpm(example_sce) <- calculateTPM(example_sce, effective_length = 5e04,
                                    calc_from = "fpkm")
```

centreSizeFactors *Centre size factors at unity*

Description

Scales all size factors so that the average size factor across cells is equal to 1.

Usage

```
centreSizeFactors(object, centre = 1, grouping = NULL)
```

Arguments

object	A SingleCellExperiment object containing any number (or zero) sets of size factors.
centre	A numeric scalar, the value around which all sets of size factors should be centred.
grouping	A factor specifying the grouping of cells, where size factors are centred to unity within each group.

Details

Centering of size factors at unity ensures that division by size factors yields values on the same scale as the raw counts. This is important for the interpretation of the normalized values, as well as comaprisons between features normalized with different size factors (e.g., spike-ins).

Specification of `grouping` centres the size factors within each level of the provided factor. This is useful if different batches are sequenced at different depth, by preserving the scale of counts within each batch.

Value

A SingleCellExperiment with modified size factors that are centred at unity.

Author(s)

Aaron Lun

See Also

areSizeFactorsCentred

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)

sizeFactors(example_sce) <- runif(ncol(example_sce))
```

```
sizeFactors(example_sce, "ERCC") <- runif(ncol(example_sce))
example_sce <- centreSizeFactors(example_sce)

mean(sizeFactors(example_sce))
mean(sizeFactors(example_sce, "ERCC"))
```

---

downsampleCounts *Downsample a count matrix*

Description

Downsample a count matrix to a desired proportion.

Usage

```
downsampleCounts(x, prop)
```

Arguments

x	matrix of counts
prop	numeric scalar or vector of length `ncol(x)` in [0, 1] indicating the downsampling proportion

Details

This function calls `downsampleMatrix`. from the **DropletUtils** package. It is deprecated and will be removed in the next release.

Value

an integer matrix of downsampled counts

Examples

```
sce10x <- read10xResults(system.file("extdata", package="scater"))
downsampled <- downsampleCounts(counts(sce10x), prop = 0.5)
```

---

filter *Return* `SingleCellExperiment` *with cells matching conditions.*

Description

Subsets the columns (cells) of a `SingleCellExperiment` based on matching conditions in the rows of `colData(object)`.

Usage

```
filter(object, ...)

## S4 method for signature 'SingleCellExperiment'
filter(object, ...)
```

Arguments

`object`	A `SingleCellExperiment` object.
`...`	Additional arguments to be passed to `dplyr::filter` to act on `colData(object)`.

Value

An SingleCellExperiment object.

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts),
colData = sc_example_cell_info)
example_sce_treat1 <- filter(example_sce, Treatment == "treat1")
```

`findImportantPCs` *Find most important principal components for a given variable*

Description

Find most important principal components for a given variable

Usage

```
findImportantPCs(object, variable = "total_features",
  plot_type = "pcs-vs-vars", exprs_values = "logcounts", ntop = 500,
  feature_set = NULL, scale_features = TRUE, theme_size = 10)
```

Arguments

`object`	an SCESet object containing expression values and experimental information. Must have been appropriately prepared.
`variable`	character scalar providing a variable name (column from `colData(object)`) for which to determine the most important PCs.
`plot_type`	character string, indicating which type of plot to produce. Default, `"pairs-pcs"` produces a pairs plot for the top 5 PCs based on their R-squared with the variable of interest. A value of `"pcs-vs-vars"` produces plots of the top PCs against the variable of interest.
`exprs_values`	which slot of the `assayData` in the `object` should be used to define expression? Valid options are "counts", "tpm", "fpkm" and "logcounts" (default), or anything else in the object added manually by the user.

ntop	numeric scalar indicating the number of most variable features to use for the PCA. Default is 500, but any ntop argument is overrided if the feature_set argument is non-NULL.
feature_set	character, numeric or logical vector indicating a set of features to use for the PCA. If character, entries must all be in rownames(object). If numeric, values are taken to be indices for features. If logical, vector is used to index features and should have length equal to nrow(object).
scale_features	logical, should the expression values be standardised so that each feature has unit variance? Default is TRUE.
theme_size	numeric scalar providing base font size for ggplot theme.

Details

Plot the top 5 or 6 most important PCs (depending on the plot_type argument for a given variable. Importance here is defined as the R-squared value from a linear model regressing each PC onto the variable of interest.

Value

a ggplot plot object

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts), colData = sc_example_cell_info)
example_sce <- normalize(example_sce)
drop_genes <- apply(exprs(example_sce), 1, function(x) {var(x) == 0})
example_sce <- example_sce[!drop_genes, ]
example_sce <- calculateQCMetrics(example_sce)
findImportantPCs(example_sce, variable="total_features")
```

---

getBMFeatureAnnos *Get feature annotation information from Biomart*

Description

Use the **biomaRt** package to add feature annotation information to an SingleCellExperiment.

Usage

```
getBMFeatureAnnos(object, filters = "ensembl_transcript_id",
  attributes = c("ensembl_transcript_id", "ensembl_gene_id", feature_symbol,
 "chromosome_name", "transcript_biotype", "transcript_start", "transcript_end",
  "transcript_count"), feature_symbol = "mgi_symbol",
  feature_id = "ensembl_gene_id", biomart = "ENSEMBL_MART_ENSEMBL",
  dataset = "mmusculus_gene_ensembl", host = "www.ensembl.org")
```

Arguments

`object`	A `SingleCellExperiment` object.
`filters`	Character vector defining the filters to pass to the `getBM` function.
`attributes`	Character vector defining the attributes to pass to `getBM`.
`feature_symbol`	String specifying the attribute to be used to define the symbol to be used for each feature Default is `"mgi_symbol"`, using gene symbols for mouse - this should be changed if the organism is not *Mus musculus*.
`feature_id`	String specifying the attribute to be used to define the ID to be used for each feature. Default is `"ensembl_gene_id"`, using the Ensembl gene IDs.
`biomart`	String defining the biomaRt to be used, to be passed to `useMart`. Default is `"ENSEMBL_MART_ENSEMBL"`.
`dataset`	String defining the dataset to use, to be passed to `useMart`. Default is `"mmusculus_gene_ensembl"`, which should be changed if the organism is not mouse.
`host`	Character string argument which can be used to select a particular `"host"` to pass to `useMart`. Useful for accessing archived versions of biomaRt data. Default is `"www.ensembl.org"`, in which case the current version of the biomaRt (now hosted by Ensembl) is used.

Value

A SingleCellExperiment object containing feature annotation. The input `feature_symbol` appears as the `feature_symbol` field in the `rowData` of the output object.

Examples

```
## Not run:
object <- getBMFeatureAnnos(object)

## End(Not run)
```

---

isOutlier *Identify outlier values*

Description

Convenience function to determine which values in a numeric vector are outliers based on the median absolute deviation (MAD).

Usage

```
isOutlier(metric, nmads = 5, type = c("both", "lower", "higher"),
  log = FALSE, subset = NULL, batch = NULL, min_diff = NA)
```

Arguments

- `metric` — Numeric vector of values.
- `nmads` — A numeric scalar, specifying the minimum number of MADs away from median required for a value to be called an outlier.
- `type` — String indicating whether outliers should be looked for at both tails (`"both"`), only at the lower tail (`"lower"`) or the upper tail (`"higher"`).
- `log` — Logical scalar, should the values of the metric be transformed to the log10 scale before computing MADs?
- `subset` — Logical or integer vector, which subset of values should be used to calculate the median/MAD? If `NULL`, all values are used. Missing values will trigger a warning and will be automatically ignored.
- `batch` — Factor of length equal to `metric`, specifying the batch to which each observation belongs. A median/MAD is calculated for each batch, and outliers are then identified within each batch.
- `min_diff` — A numeric scalar indicating the minimum difference from the median to consider as an outlier. The outlier threshold is defined from the larger of `nmads` MADs and `min_diff`, to avoid calling many outliers when the MAD is very small. If `NA`, it is ignored.

Value

A logical vector of the same length as the `metric` argument, specifying the observations that are considered as outliers.

Author(s)

Aaron Lun

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)
example_sce <- calculateQCMetrics(example_sce)

## with a set of feature controls defined
example_sce <- calculateQCMetrics(example_sce,
feature_controls = list(set1 = 1:40))
isOutlier(example_sce$total_counts, nmads = 3)
```

kallisto-wrapper *kallisto wrapper functions*

Description

Run the abundance quantification tool kallisto on a set of FASTQ files. Requires kallisto (http://pachterlab.github.io/kallisto/) to be installed and a kallisto feature index must have been generated prior to using this function. See the kallisto website for installation and basic usage instructions.

Read kallisto results for a single sample into a list

After generating transcript/feature abundance results using kallisto for a batch of samples, read these abundance values into a SingleCellExperiment object.

Usage

```
runKallisto(targets_file, transcript_index, single_end = TRUE,
  output_prefix = "output", fragment_length = NULL,
  fragment_standard_deviation = NULL, n_cores = 2,
  n_bootstrap_samples = 0, bootstrap_seed = NULL, correct_bias = TRUE,
  plaintext = FALSE, kallisto_version = "current", verbose = TRUE,
  dry_run = FALSE, kallisto_cmd = "kallisto")

readKallistoResultsOneSample(directory, read_h5 = FALSE,
  kallisto_version = "current")

readKallistoResults(kallisto_log = NULL, samples = NULL,
  directories = NULL, read_h5 = FALSE, kallisto_version = "current",
  verbose = TRUE)
```

Arguments

targets_file
: character string giving the path to a tab-delimited text file with either 2 columns (single-end reads) or 3 columns (paired-end reads) that gives the sample names (first column) and FastQ file names (column 2 and if applicable 3). The file is assumed to have column headers, although these are not used.

transcript_index
: character string giving the path to the kallisto index to be used for the feature abundance quantification.

single_end
: logical, are single-end reads used, or paired-end reads?

output_prefix
: character string giving the prefix for the output folder that will contain the kallisto results. The default is "output" and the sample name (column 1 of targets_file) is appended (preceded by an underscore).

fragment_length
: scalar integer or numeric giving the estimated average fragment length. Required argument if single_end is TRUE, optional if FALSE (kallisto default for paired-end data is that the value is estimated from the input data).

fragment_standard_deviation
: scalar numeric giving the estimated standard deviation of read fragment length. Required argument if single_end is TRUE, optional if FALSE (kallisto default for paired-end data is that the value is estimated from the input data).

n_cores integer giving the number of cores (nodes/threads) to use for the kallisto jobs. The package parallel is used. Default is 2 cores.

n_bootstrap_samples
integer giving the number of bootstrap samples that kallisto should use (default is 0). With bootstrap samples, uncertainty in abundance can be quantified.

bootstrap_seed scalar integer or numeric giving the seed to use for the bootstrap sampling (default used by kallisto is 42). Optional argument.

correct_bias logical, should kallisto's option to model and correct abundances for sequence specific bias? Requires kallisto version 0.42.2 or higher.

plaintext logical, if TRUE then bootstrapping results are returned in a plain text file rather than an HDF5 https://www.hdfgroup.org/HDF5/ file.

kallisto_version
character string indicating whether or not the version of kallisto to be used is "pre-0.42.2" or "current". This is required because the kallisto developers changed the output file extensions and added features in version 0.42.2.

verbose logical, should timings for the run be printed?

dry_run logical, if TRUE then a list containing the kallisto commands that would be run and the output directories is returned. Can be used to read in results if kallisto is run outside an R session or to produce a script to run outside of an R session.

kallisto_cmd (optional) string giving full command to use to call kallisto, if simply typing "kallisto" at the command line does not give the required version of kallisto or does not work. Default is simply "kalliso". If used, this argument should give the full path to the desired kallisto binary.

directory character string giving the path to the directory containing the kallisto results for the sample.

read_h5 logical, if TRUE then read in bootstrap results from the HDF5 object produced by kallisto.

kallisto_log list, generated by runKallisto. If provided, then samples and directories arguments are ignored.

samples character vector providing a set of sample names to use for the abundance results.

directories character vector providing a set of directories containing kallisto abundance results to be read in.

Details

A kallisto transcript index can be built from a FASTA file: kallisto index [arguments] FASTA-file. See the kallisto documentation for further details.

The directory is expected to contain results for just a single sample. Putting more than one sample's results in the directory will result in unpredictable behaviour with this function. The function looks for the files (with the default names given by kallisto) 'abundance.txt', 'run_info.json' and (if read_h5=TRUE) 'abundance/h5'. If these files are missing, or if results files have different names, then this function will not find them.

This function expects to find only one set of kallisto abundance results per directory; multiple adundance results in a given directory will be problematic.

Value

A list containing three elements for each sample for which feature abundance has been quantified: (1) `kallisto_call`, the call used for kallisto, (2) `kallisto_log` the log generated by kallisto, and (3) `output_dir` the directory in which the kallisto results can be found.

A list with two elements: (1) a data.frame `abundance` with columns for 'target_id' (feature, transcript, gene etc), 'length' (feature length), 'eff_length' (effective feature length), 'est_counts' (estimated feature counts), 'tpm' (transcripts per million) and possibly many columns containing bootstrap estimated counts; and (2) a list `run_info` with details about the kallisto run that generated the results.

a SingleCellExperiment object

Examples

```
## Not run:
## If in kallisto's 'test' directory, then try these calls:
## Generate 'targets.txt' file:
write.table(data.frame(Sample="sample1", File1="reads_1.fastq.gz", File2="reads_1.fastq.gz"),
 file="targets.txt", quote=FALSE, row.names=FALSE, sep="\t")
kallisto_log <- runKallisto("targets.txt", "transcripts.idx", single_end=FALSE,
         output_prefix="output", verbose=TRUE, n_bootstrap_samples=10,
         dry_run = FALSE)

## End(Not run)
# If kallisto results are in the directory "output", then call:
# readKallistoResultsOneSample("output")
## Not run:
kallisto_log <- runKallisto("targets.txt", "transcripts.idx", single_end=FALSE,
         output_prefix="output", verbose=TRUE, n_bootstrap_samples=10)
sceset <- readKallistoResults(kallisto_log)

## End(Not run)
```

`librarySizeFactors` *Compute library size factors*

Description

Define size factors from the library sizes after centering. This ensures that the library size adjustment yields values comparable to those generated after normalization with other sets of size factors.

Usage

```
librarySizeFactors(object, exprs_values = "counts")
```

Arguments

`object`	A count matrix or SingleCellExperiment object containing counts.
`exprs_values`	A string indicating the assay of `object` containing the counts, if `object` is a SingleCellExperiment.

Value

A numeric vector of size factors.

Examples

```
data("sc_example_counts")
summary(librarySizeFactors(sc_example_counts))
```

`multiplot` *Multiple plot function for ggplot2 plots*

Description

Place multiple `ggplot` plots on one page.

Usage

```
multiplot(..., plotlist = NULL, cols = 1, layout = NULL)
```

Arguments

`...`	One or more ggplot objects.
`plotlist`	A list of ggplot objects, as an alternative to `...`.
`cols`	A numeric scalar giving the number of columns in the layout.
`layout`	A matrix specifying the layout. If present, `cols` is ignored.

Details

If the layout is something like `matrix(c(1,2,3,3), nrow=2, byrow=TRUE)`, then:

- plot 1 will go in the upper left;
- plot 2 will go in the upper right;
- and plot 3 will go all the way across the bottom.

There is no way to tweak the relative heights or widths of the plots with this simple function. It was adapted from `http://www.cookbook-r.com/Graphs/Multiple_graphs_on_one_page_(ggplot2)/`

Value

A ggplot object.

Examples

```
library(ggplot2)

## This example uses the ChickWeight dataset, which comes with ggplot2
## First plot
p1 <- ggplot(ChickWeight, aes(x = Time, y = weight, colour = Diet, group = Chick)) +
   geom_line() +
   ggtitle("Growth curve for individual chicks")
## Second plot
p2 <- ggplot(ChickWeight, aes(x = Time, y = weight, colour = Diet)) +
   geom_point(alpha = .3) +
   geom_smooth(alpha = .2, size = 1) +
   ggtitle("Fitted growth curve per diet")

## Third plot
p3 <- ggplot(subset(ChickWeight, Time == 21), aes(x = weight, colour = Diet)) +
   geom_density() +
   ggtitle("Final weight, by diet")
## Fourth plot
p4 <- ggplot(subset(ChickWeight, Time == 21), aes(x = weight, fill = Diet)) +
    geom_histogram(colour = "black", binwidth = 50) +
   facet_grid(Diet ~ .) +
   ggtitle("Final weight, by diet") +
   theme(legend.position = "none")        # No legend (redundant in this graph)

## Combine plots and display
multiplot(p1, p2, p3, p4, cols = 2)
```

---

mutate — *Add new variables to* `colData(object)`.

Description

Adds ne

Usage

```
mutate(object, ...)

## S4 method for signature 'SingleCellExperiment'
mutate(object, ...)
```

Arguments

object	a `SingleCellExperiment` object.
...	Additional arguments to be passed to `dplyr::mutate` to act on `colData(object)`.

Value

An SingleCellExperiment object.

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts),
colData = sc_example_cell_info)
example_sce <- mutate(example_sce, is_quiescent = Cell_Cycle == "G0")
```

---

nexprs *Count the number of expressed genes per cell*

Description

An efficient internal function that avoids the need to construct 'is_exprs_mat' by counting the number of expressed genes per cell on the fly.

Usage

```
nexprs(object, detection_limit = 0, exprs_values = "counts",
  byrow = FALSE, subset_row = NULL, subset_col = NULL)
```

Arguments

object	a SingleCellExperiment object or a numeric matrix of expression values.
detection_limit	numeric scalar providing the value above which observations are deemed to be expressed. Defaults to `object@detection_limit`.
exprs_values	character scalar indicating whether the count data (`"counts"`), the log-transformed count data (`"logcounts"`), transcript-per-million (`"tpm"`), counts-per-million (`"cpm"`) or FPKM (`"fpkm"`) should be used to define if an observation is expressed or not. Defaults to the first available value of those options in the order shown. However, if `is_exprs(object)` is present, it will be used directly; `exprs_values` and `detection_limit` are ignored.
byrow	logical scalar indicating if `TRUE` to count expressing cells per feature (i.e. gene) and if `FALSE` to count expressing features (i.e. genes) per cell.
subset_row	logical, integeror character vector indicating which rows (i.e. features/genes) to use.
subset_col	logical, integer or character vector indicating which columns (i.e., cells) to use.

Details

Setting `subset_row` or `subset_col` is equivalent to subsetting `object` before calling `nexprs`, but more efficient as a new copy of the matrix is not constructed.

Value

If `byrow=TRUE`, an integer vector containing the number of cells expressing each feature, of the same length as the number of features in `subset_row` (all features in `exprs_mat` if `subset_row=NULL`).

If `byrow=FALSE`, an integer vector containing the number of genes expressed in each cell, of the same length as the number of cells specified in `subset_col` (all cells in `exprs_mat` if `subset_col=NULL`).

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts), colData = sc_example_cell_info)
nexprs(example_sce)[1:10]
nexprs(example_sce, byrow = TRUE)[1:10]
```

---

normalize — *Normalise a SingleCellExperiment object using pre-computed size factors*

Description

Compute normalised expression values from count data in a SingleCellExperiment object, using the size factors stored in the object.

Usage

```
normalizeSCE(object, exprs_values = "counts", return_log = TRUE,
  log_exprs_offset = NULL, centre_size_factors = TRUE,
  size_factor_grouping = NULL)

## S4 method for signature 'SingleCellExperiment'
normalize(object, exprs_values = "counts",
  return_log = TRUE, log_exprs_offset = NULL, centre_size_factors = TRUE,
  size_factor_grouping = NULL)

normalise(...)
```

Arguments

`object`	A SingleCellExperiment object.
`exprs_values`	String indicating which assay contains the count data that should be used to compute log-transformed expression values.
`return_log`	Logical scalar, should normalized values be returned on the log2 scale?
`log_exprs_offset`	Numeric scalar specifying the offset to add when log-transforming expression values. If `NULL`, value is taken from `metadata(object)$log.exprs.offset` if defined, otherwise 1.
`centre_size_factors`	Logical scalar indicating whether size fators should be centred.
`size_factor_grouping`	Factor specifying groups of cells in which size factors should be centred, see `centreSizeFactors` for details.
`...`	Arguments passed to `normalize` when calling `normalise`.

Details

Normalized expression values are computed by dividing the counts for each cell by the size factor for that cell. This aims to remove cell-specific scaling biases, e.g., due to differences in sequencing coverage or capture efficiency. If `log=TRUE`, log-normalized values are calculated by adding `log_exprs_offset` to the normalized count and performing a log2 transformation.

Features marked as spike-in controls will be normalized with control-specific size factors, if these are available. This reflects the fact that spike-in controls are subject to different biases than those that are removed by gene-specific size factors (namely, total RNA content). If size factors for a particular spike-in set are not available, a warning will be raised.

Size factors will be centred to have a mean of unity if `centre_size_factors=TRUE`, prior to calculation of normalized expression values. This ensures that the computed `exprs` can be interpreted as being on the same scale as log-counts. It also standardizes the effect of the `log_exprs_offset` addition, and ensures that abundances are roughly comparable between features normalized with different sets of size factors.

If `size_factor_grouping` is specified and `centre_size_factors=TRUE`, this is equivalent to subsetting the SingleCellExperiment; centering the size factors within each subset; normalizing within each subset; and then merging the subsets back together for output. This enables convenient normalization of multiple batches separately.

Note that `normalize` is exactly the same as `normalise`.

Value

A SingleCellExperiment object containing normalized expression values in `"normcounts"` if `log=FALSE`, and log-normalized expression values in `"logcounts"` if `log=TRUE`. All size factors will also be centred in the output object if `centre_size_factors=TRUE`.

Warning about centred size factors

Generally speaking, centering does not affect relative comparisons between cells in the same `object`, as all size factors are scaled by the same amount. However, if two different `SingleCellExperiment` objects are run separately through `normalize`, the size factors in each object will be rescaled differently. This means that the size factors and log-expression values will *not* be comparable between objects.

This lack of comparability is not always obvious. For example, if we subsetted an existing SingleCellExperiment object, and ran `normalize` separately on each subset, the resulting expression values in each subsetted object would *not* be comparable to each other. This is despite the fact that all cells were originally derived from a single SingleCellExperiment object.

In general, it is advisable to only compare size factors and expression values between cells in one SingleCellExperiment object, from a single `normalize` call with `size_factor_grouping=NULL`. If objects are to be combined, new size factors should be computed using all cells in the combined object, followed by a single `normalize` call. If `size_factor_grouping` is specified, expression values should only be compared *within* each level of the specified factor.

Author(s)

Davis McCarthy and Aaron Lun

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
```

```
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)
keep_gene <- rowSums(counts(example_sce)) > 0
example_sce <- example_sce[keep_gene,]

## Apply TMM normalisation taking into account all genes
example_sce <- normaliseExprs(example_sce, method = "TMM")
## Scale counts relative to a set of control features (here the first 100 features)
example_sce <- normaliseExprs(example_sce, method = "none",
feature_set = 1:100)

## normalize the object using the saved size factors
example_sce <- normalize(example_sce)
```

`normalizeExprs` *Normalise expression levels for a SingleCellExperiment object*

Description

Compute normalised expression values from a SingleCellExperiment object and return the object with the normalised expression values added.

Usage

```
normalizeExprs(object, method = "none", design = NULL, feature_set = NULL,
  exprs_values = "counts", return_norm_as_exprs = TRUE, return_log = TRUE,
  ...)

normaliseExprs(...)
```

Arguments

`object`	A SingleCellExperiment object.
`method`	character string specified the method of calculating normalisation factors. Passed to `calcNormFactors`.
`design`	design matrix defining the linear model to be fitted to the normalised expression values. If not `NULL`, then the residuals of this linear model fit are used as the normalised expression values.
`feature_set`	character, numeric or logical vector indicating a set of features to use for calculating normalisation factors. If character, entries must all be in `featureNames(object)`. If numeric, values are taken to be indices for features. If logical, vector is used to index features and should have length equal to `nrow(object)`.
`exprs_values`	character string indicating which slot of the assayData from the `SingleCellExperiment` object should be used for the calculations. Valid options are `'counts'`, `'tpm'`, `'cpm'`, `'fpkm'` and `'exprs'`. Defaults to the first available value of these options in in order shown.

return_norm_as_exprs
: logical, should the normalised expression values be returned to the exprs slot of the object? Default is TRUE. If FALSE, values in the exprs slot will be left untouched. Regardless, normalised expression values will be returned to the norm_exprs slot of the object.

return_log
: logical(1), should normalized values be returned on the log scale? Default is TRUE. If TRUE and return_norm_as_exprs is TRUE then normalised output is stored as "logcounts" in the returned object; if TRUE and return_norm_as_exprs is FALSE then normalised output is stored as "norm_exprs"; if FALSE output is stored as "normcounts"

...
: arguments passed to normaliseExprs (in the case of normalizeExprs) or to calcNormFactors.

Details

This function allows the user to compute normalised expression values from an SingleCellExperiment object. The 'raw' values used can be the values in the 'counts' (default), or another specified assay slot of the SingleCellExperiment. Normalised expression values are computed through normalizeSCE and are on the log2-scale by default (if return_log is TRUE), with an offset defined by the metadata(object)$log.exprs.offset value in the SingleCellExperiment object. These are added to the 'norm_exprs' slot of the returned object. If 'exprs_values' argument is 'counts' and return_log is FALSE a 'normcounts' slot is added, containing normalised counts-per-million values.

If the raw values are counts, this function will compute size factors using methods in calcNormFactors. Library sizes are multiplied by size factors to obtain an "effective library size" before calculation of the aforementioned normalized expression values. If feature_set is specified, only the specified features will be used to calculate the size factors.

If the user wishes to remove the effects of certain explanatory variables, then the 'design' argument can be defined. The design argument must be a valid design matrix, for example as produced by model.matrix, with the relevant variables. A linear model is then fitted using lmFit on expression values after any size-factor and library size normalisation as descrived above. The returned values in 'norm_exprs' are the residuals from the linear model fit.

After normalisation, normalised expression values can be accessed with the norm_exprs function (with corresponding accessor functions for counts, tpm, fpkm, cpm). These functions can also be used to assign normalised expression values produced with external tools to a SingleCellExperiment object.

normalizeExprs is exactly the same as normaliseExprs, provided for those who prefer North American spelling.

Value

an SingleCellExperiment object

Author(s)

Davis McCarthy

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
```

```
assays = list(counts = sc_example_counts), colData = sc_example_cell_info)
keep_gene <- rowSums(counts(example_sce)) > 0
example_sce <- example_sce[keep_gene,]

## Apply TMM normalisation taking into account all genes
example_sce <- normaliseExprs(example_sce, method = "TMM")
## Scale counts relative to a set of control features (here the first 100 features)
example_sce <- normaliseExprs(example_sce, method = "none",
feature_set = 1:100)
```

`norm_exprs` *Additional accessors for the typical elements of a SingleCellExperiment object.*

Description

Convenience functions to access commonly-used assays of the `SingleCellExperiment` object.

Usage

```
norm_exprs(object)

norm_exprs(object) <- value

stand_exprs(object)

stand_exprs(object) <- value

fpkm(object)

fpkm(object) <- value
```

Arguments

`object`	`SingleCellExperiment` class object from which to access or to which to assign assay values. Namely: "exprs", norm_exprs", "stand_exprs", "fpkm". The following are imported from `SingleCellExperiment`: "counts", "normcounts", "logcounts", "cpm", "tpm".
`value`	a numeric matrix (e.g. for `exprs`)

Value

a matrix of normalised expression data

a matrix of standardised expressiond data

a matrix of FPKM values

A matrix of numeric, integer or logical values.

Author(s)

Davis McCarthy

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts), colData = sc_example_cell_info)

example_sce <- normalize(example_sce)
head(logcounts(example_sce)[,1:10])
head(exprs(example_sce)[,1:10]) # identical to logcounts()

example_sce <- SingleCellExperiment(
assays = list(norm_counts = sc_example_counts), colData = sc_example_cell_info)

counts(example_sce) <- sc_example_counts
norm_exprs(example_sce) <- log2(calculateCPM(example_sce, use_size_factors = FALSE) + 1)

stand_exprs(example_sce) <- log2(calculateCPM(example_sce, use_size_factors = FALSE) + 1)

tpm(example_sce) <- calculateTPM(example_sce, effective_length = 5e4)

cpm(example_sce) <- calculateCPM(example_sce, use_size_factors = FALSE)

fpkm(example_sce)
```

---

plotColData *Plot column metadata*

Description

Plot column-level (i.e., cell) metadata in an SingleCellExperiment object.

Usage

```
plotColData(object, y, x = NULL, colour_by = NULL, shape_by = NULL,
  size_by = NULL, by_exprs_values = "logcounts", by_show_single = FALSE,
  ...)

plotPhenoData(...)

plotCellData(...)
```

Arguments

object	A SingleCellExperiment object containing expression values and experimental information.
y	Specification of the column-level metadata to show on the y-axis, see ?"scater-vis-var" for possible values. Note that only metadata fields will be searched, assays will not be used.
x	Specification of the column-level metadata to show on the x-axis, see ?"scater-vis-var" for possible values. Again, only metadata fields will be searched, assays will not be used.

`colour_by` Specification of a column metadata field or a feature to colour by, see ?"scater-vis-var" for possible values.

`shape_by` Specification of a column metadata field or a feature to shape by, see ?"scater-vis-var" for possible values.

`size_by` Specification of a column metadata field or a feature to size by, see ?"scater-vis-var" for possible values.

`by_exprs_values` A string or integer scalar specifying which assay to obtain expression values from, for use in point aesthetics - see ?"scater-vis-var" for details.

`by_show_single` Logical scalar specifying whether single-level factors should be used for point aesthetics, see ?"scater-vis-var" for details.

`...` Additional arguments for visualization, see ?"scater-plot-args" for details.

Details

If `y` is continuous and `x=NULL`, a violin plot is generated. If `x` is categorical, a grouped violin plot will be generated, with one violin for each level of `x`. If `x` is continuous, a scatter plot will be generated.

If `y` is categorical and `x` is continuous, horizontal violin plots will be generated. If `x` is missing or categorical, rectangule plots will be generated where the area of a rectangle is proportional to the number of points for a combination of factors.

Note that `plotPhenoData` and `plotCellData` are synonyms for `plotColData`. These are artifacts of the transition from the old SCESet class, and will be deprecated in future releases.

Value

A ggplot object.

Author(s)

Davis McCarthy, with modifications by Aaron Lun

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)
example_sce <- calculateQCMetrics(example_sce)
example_sce <- normalize(example_sce)

plotColData(example_sce, y = "total_features_by_counts",
   x = "log10_total_counts", colour_by = "Mutation_Status")

plotColData(example_sce, y = "total_features_by_counts",
   x = "log10_total_counts", colour_by = "Mutation_Status",
   size_by = "Gene_0001", shape_by = "Treatment")

plotColData(example_sce, y = "Treatment",
   x = "log10_total_counts", colour_by = "Mutation_Status")
```

```
plotColData(example_sce, y = "total_features_by_counts",
   x = "Cell_Cycle", colour_by = "Mutation_Status")
```

---

plotExplanatoryVariables   *Plot explanatory variables ordered by percentage of phenotypic variance explained*

Description

Plot explanatory variables ordered by percentage of phenotypic variance explained

Usage

```
plotExplanatoryVariables(object, method = "density",
  exprs_values = "logcounts", nvars_to_plot = 10, min_marginal_r2 = 0,
  variables = NULL, return_object = FALSE, theme_size = 10, ...)
```

Arguments

object	an SingleCellExperiment object containing expression values and experimental information. Must have been appropriately prepared.
method	character scalar indicating the type of plot to produce. If "density", the function produces a density plot of R-squared values for each variable when fitted as the only explanatory variable in a linear model. If "pairs", then the function produces a pairs plot of the explanatory variables ordered by the percentage of feature expression variance (as measured by R-squared in a marginal linear model) explained.
exprs_values	which slot of the `assayData` in the `object` should be used to define expression? Valid options are "logcounts" (default), "tpm", "fpkm", "cpm", and "counts".
nvars_to_plot	integer, the number of variables to plot in the pairs plot. Default value is 10.
min_marginal_r2	numeric scalar giving the minimal value required for median marginal R-squared for a variable to be plotted. Only variables with a median marginal R-squared strictly larger than this value will be plotted.
variables	optional character vector giving the variables to be plotted. Default is `NULL`, in which case all variables in `colData(object)` are considered and the `nvars_to_plot` variables with the highest median marginal R-squared are plotted.
return_object	logical, should an `SingleCellExperiment` object with median marginal R-squared values added to `varMetadata(object)` be returned?
theme_size	numeric scalar giving font size to use for the plotting theme
...	parameters to be passed to `pairs`.

Details

If the `method` argument is "pairs", then the function produces a pairs plot of the explanatory variables ordered by the percentage of feature expression variance (as measured by R-squared in a marginal linear model) explained by variable. Median percentage R-squared is reported on the plot for each variable. Discrete variables are coerced to a factor and plotted as integers with jittering. Variables with only one unique value are quietly ignored.

Value

A ggplot object

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts), colData = sc_example_cell_info)
example_sce <- normalize(example_sce)
drop_genes <- apply(exprs(example_sce), 1, function(x) {var(x) == 0})
example_sce <- example_sce[!drop_genes, ]
example_sce <- calculateQCMetrics(example_sce)
vars <- names(colData(example_sce))[c(2:3, 5:14)]
plotExplanatoryVariables(example_sce, variables=vars)
```

---

plotExpression *Plot expression values for all cells*

Description

Plot expression values for a set of features (e.g. genes or transcripts) in a SingleExperiment object, against a continuous or categorical covariate for all cells.

Usage

```
plotExpression(object, features, x = NULL, exprs_values = "logcounts",
  log2_values = FALSE, colour_by = NULL, shape_by = NULL,
  size_by = NULL, by_exprs_values = exprs_values, by_show_single = FALSE,
  xlab = NULL, feature_colours = TRUE, one_facet = TRUE, ncol = 2,
  scales = "fixed", ...)
```

Arguments

object	A SingleCellExperiment object containing expression values and other metadata.
features	A character vector (of feature names), a logical vector or numeric vector (of indices) specifying the features to plot.
x	Specification of a column metadata field or a feature to show on the x-axis, see ?"scater-vis-var" for possible values.
exprs_values	A string or integer scalar specifying which assay in assays(object) to obtain expression values from.
log2_values	Logical scalar, specifying whether the expression values be transformed to the log2-scale for plotting (with an offset of 1 to avoid logging zeroes).
colour_by	Specification of a column metadata field or a feature to colour by, see ?"scater-vis-var" for possible values.
shape_by	Specification of a column metadata field or a feature to shape by, see ?"scater-vis-var" for possible values.

size_by
: Specification of a column metadata field or a feature to size by, see ?"scater-vis-var" for possible values.

by_exprs_values
: A string or integer scalar specifying which assay to obtain expression values from, for use in point aesthetics - see ?"scater-vis-var" for details.

by_show_single
: Logical scalar specifying whether single-level factors should be used for point aesthetics, see ?"scater-vis-var" for details.

xlab
: String specifying the label for x-axis. If NULL (default), x will be used as the x-axis label.

feature_colours
: Logical scalar indicating whether violins should be coloured by feature when x and colour_by are not specified and one_facet=TRUE.

one_facet
: Logical scalar indicating whether grouped violin plots for multiple features should be put onto one facet. Only relevant when x=NULL.

ncol
: Integer scalar, specifying the number of columns to be used for the panels of a multi-facet plot.

scales
: String indicating whether should multi-facet scales be fixed ("fixed"), free ("free"), or free in one dimension ("free_x", "free_y"). Passed to the scales argument in the facet_wrap when multiple facets are generated.

...
: Additional arguments for visualization, see ?"scater-plot-args" for details.

Details

This function plots expression values for one or more features. If x is not specified, a violin plot will be generated of expression values. If x is categorical, a grouped violin plot will be generated, with one violin for each level of x. If x is continuous, a scatter plot will be generated.

If multiple features are requested and x is not specified and one_facet=TRUE, a grouped violin plot will be generated with one violin per feature. This will be coloured by feature if colour_by=NULL and feature_colours=TRUE, to yield a more aesthetically pleasing plot. Otherwise, if x is specified or one_facet=FALSE, a multi-panel plot will be generated where each panel corresponds to a feature. Each panel will be a scatter plot or (grouped) violin plot, depending on the nature of x.

Note that this assumes that the expression values are numeric. If not, and x is continuous, horizontal violin plots will be generated. If x is missing or categorical, rectangule plots will be generated where the area of a rectangle is proportional to the number of points for a combination of factors.

Value

A ggplot object.

Author(s)

Davis McCarthy, with modifications by Aaron Lun

Examples

```
## prepare data
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
```

```
)
example_sce <- calculateQCMetrics(example_sce)
sizeFactors(example_sce) <- colSums(counts(example_sce))
example_sce <- normalize(example_sce)

## default plot
plotExpression(example_sce, 1:15)

## plot expression against an x-axis value
plotExpression(example_sce, c("Gene_0001", "Gene_0004"), x="Mutation_Status")
plotExpression(example_sce, c("Gene_0001", "Gene_0004"), x="Gene_0002")

## add visual options
plotExpression(example_sce, 1:6, colour_by = "Mutation_Status")
plotExpression(example_sce, 1:6, colour_by = "Mutation_Status",
     shape_by = "Treatment", size_by = "Gene_0010")

## plot expression against expression values for Gene_0004
plotExpression(example_sce, 1:4, "Gene_0004", show_smooth = TRUE)
```

`plotExprsFreqVsMean` *Plot frequency against mean for each feature*

Description

Plot the frequency of expression (i.e., percentage of expressing cells) against the mean expression level for each feature in a SingleCellExperiment object.

Usage

```
plotExprsFreqVsMean(object, freq_exprs, mean_exprs, controls,
  by_show_single = FALSE, show_smooth = TRUE, show_se = TRUE, ...)
```

Arguments

`object`	A SingleCellExperiment object.
`freq_exprs`	Specification of the row-level metadata field containing the number of expressing cells per feature, see ?"scater-vis-var" for possible values. Note that only metadata fields will be searched, assays will not be used. If not supplied or NULL, this defaults to "n_cells_by_counts" or equivalent for compacted data.
`mean_exprs`	Specification of the row-level metadata field containing the mean expression of each feature, see ?"scater-vis-var" for possible values. Again, only metadata fields will be searched, assays will not be used. If not supplied or NULL, this defaults to "mean_counts" or equivalent for compacted data.
`controls`	Specification of the row-level metadata column indicating whether a feature is a control, see ?"scater-vis-var" for possible values. Only metadata fields will be searched, assays will not be used. If not supplied, this defaults to "is_feature_control" or equivalent for compacted data.
`by_show_single`	Logical scalar specifying whether a single-level factor for controls should be used for colouring, see ?"scater-vis-var" for details.

show_smooth	Logical scalar, should a smoothed fit (through feature controls if available; all features otherwise) be shown on the plot? See geom_smooth for details.
show_se	Logical scalar, should the standard error be shown for a smoothed fit?
...	Further arguments passed to plotRowData.

Details

This function plots gene expression frequency versus mean expression level, which can be useful to assess the effects of technical dropout in the dataset. We fit a non-linear least squares curve for the relationship between expression frequency and mean expression. We use this curve to define the number of genes above high technical dropout and the numbers of genes that are expressed in at least 50% and at least 25% of cells.

The plot will attempt to colour the points based on whether the corresponding features are labelled as feature controls in `object`. This can be turned off by setting `controls=NULL`.

Value

A ggplot object.

See Also

plotRowData

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)
example_sce <- normalize(example_sce)

example_sce <- calculateQCMetrics(example_sce,
    feature_controls = list(set1 = 1:500))
plotExprsFreqVsMean(example_sce)

plotExprsFreqVsMean(example_sce, size_by = "is_feature_control")
```

---

plotExprsVsTxLength *Plot expression against transcript length*

Description

Plot mean expression values for all features in a SingleCellExperiment object against transcript length values.

Usage

```
plotExprsVsTxLength(object, tx_length = "median_feat_eff_len",
  length_is_assay = FALSE, exprs_values = "logcounts",
  log2_values = FALSE, colour_by = NULL, shape_by = NULL,
  size_by = NULL, by_exprs_values = exprs_values, by_show_single = FALSE,
  xlab = "Median transcript length", show_exprs_sd = FALSE, ...)
```

Arguments

object	A SingleCellExperiment object.
tx_length	Transcript lengths for all features, to plot on the x-axis. If `length_is_assay=FALSE`, this can take any of the values described in `?"scater-vis-var"` for feature-level metadata; data in `assays(object)` will *not* be searched. Otherwise, if `length_is_assay=TRUE`, `tx_length` should be the name or index of an assay in `object`.
length_is_assay	Logical scalar indicating whether `tx_length` refers to an assay of `object` containing transcript lengths for all features in all cells.
exprs_values	A string or integer scalar specifying which assay in `assays(object)` to obtain expression values from.
log2_values	Logical scalar, specifying whether the expression values be transformed to the log2-scale for plotting (with an offset of 1 to avoid logging zeroes).
colour_by	Specification of a column metadata field or a feature to colour by, see `?"scater-vis-var"` for possible values.
shape_by	Specification of a column metadata field or a feature to shape by, see `?"scater-vis-var"` for possible values.
size_by	Specification of a column metadata field or a feature to size by, see `?"scater-vis-var"` for possible values.
by_exprs_values	A string or integer scalar specifying which assay to obtain expression values from, for use in point aesthetics - see `?"scater-vis-var"` for details.
by_show_single	Logical scalar specifying whether single-level factors should be used for point aesthetics, see `?"scater-vis-var"` for details.
xlab	String specifying the label for x-axis.
show_exprs_sd	Logical scalar indicating whether the standard deviation of expression values for each feature should be plotted.
...	Additional arguments for visualization, see `?"scater-plot-args"` for details.

Details

If `length_is_assay=TRUE`, the median transcript length of each feature across all cells is used. This may be necessary if the effective transcript length differs across cells, e.g., as observed in the results from pseudo-aligners.

Value

A ggplot object.

Author(s)

Davis McCarthy, with modifications by Aaron Lun

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
rd <- DataFrame(gene_id = rownames(sc_example_counts),
    feature_id = paste("feature", rep(1:500, each = 4), sep = "_"),
    median_tx_length = rnorm(2000, mean = 5000, sd = 500),
    other = sample(LETTERS, 2000, replace = TRUE)
)
rownames(rd) <- rownames(sc_example_counts)
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info, rowData = rd
)
example_sce <- normalize(example_sce)

plotExprsVsTxLength(example_sce, "median_tx_length")
plotExprsVsTxLength(example_sce, "median_tx_length", show_smooth = TRUE)
plotExprsVsTxLength(example_sce, "median_tx_length", show_smooth = TRUE,
    colour_by = "other", show_exprs_sd = TRUE)

## using matrix of tx length values in assays(object)
mat <- matrix(rnorm(ncol(example_sce) * nrow(example_sce), mean = 5000,
    sd = 500), nrow = nrow(example_sce))
dimnames(mat) <- dimnames(example_sce)
assay(example_sce, "tx_len") <- mat

plotExprsVsTxLength(example_sce, "tx_len", show_smooth = TRUE,
    length_is_assay = TRUE, show_exprs_sd = TRUE)

## using a vector of tx length values
plotExprsVsTxLength(example_sce,
    data.frame(rnorm(2000, mean = 5000, sd = 500)))
```

---

plotHeatmap *Plot heatmap of gene expression values*

Description

Create a heatmap of expression values for each cell and specified features in a SingleCellExperiment object.

Usage

```
plotHeatmap(object, features, columns = NULL, exprs_values = "logcounts",
  center = FALSE, zlim = NULL, symmetric = FALSE, color = NULL,
  colour_columns_by = NULL, by_exprs_values = exprs_values,
  by_show_single = FALSE, ...)
```

Arguments

`object`	A SingleCellExperiment object.
`features`	A character vector of row names, a logical vector of integer vector of indices specifying rows of `object` to show in the heatmap.
`columns`	A vector specifying the subset of columns in `object` to show as columns in the heatmp. By default, all columns are used in their original order.
`exprs_values`	A string or integer scalar indicating which assay of `object` should be used as expression values for colouring in the heatmap.
`center`	A logical scalar indicating whether each row should have its mean expression centered at zero prior to plotting.
`zlim`	A numeric vector of length 2, specifying the upper and lower bounds for the expression values. This winsorizes the expression matrix prior to plotting (but after centering, if `center=TRUE`). If `NULL`, it defaults to the range of the expression matrix.
`symmetric`	A logical scalar specifying whether the default `zlim` should be symmetric around zero. If `TRUE`, the maximum absolute value of `zlim` will be computed and multiplied by `c(-1, 1)` to redefine `zlim`.
`color`	A vector of colours specifying the palette to use for mapping expression values to colours. This defaults to the default setting in `pheatmap`.
`colour_columns_by`	A list of values specifying how the columns should be annotated with colours. Each entry of the list can be of the form described by `?"scater-vis-var"`. A character vector can also be supplied and will be treated as a list of strings.
`by_exprs_values`	A string or integer scalar specifying which assay to obtain expression values from, for colouring of column-level data - see `?"scater-vis-var"` for details.
`by_show_single`	Logical scalar specifying whether single-level factors should be used for column-level colouring, see `?"scater-vis-var"` for details.
`...`	Additional arguments to pass to `pheatmap`.

Details

Setting `center=TRUE` is useful for examining log-fold changes of each cell's expression profile from the average across all cells. This avoids issues with the entire row appearing a certain colour because the gene is highly/lowly expressed across all cells.

Setting `zlim` preserves the dynamic range of colours in the presence of outliers. Otherwise, the plot may be dominated by a few genes, which will "flatten" the observed colours for the rest of the heatmap.

Value

A heatmap is produced on the current graphics device. The output of `pheatmap` is invisibly returned.

Author(s)

Aaron Lun

See Also

`pheatmap`

Examples

```
example(normalizeSCE) # borrowing the example objects in here.
plotHeatmap(example_sce, features=rownames(example_sce)[1:10])
plotHeatmap(example_sce, features=rownames(example_sce)[1:10],
    center=TRUE, symmetric=TRUE)

plotHeatmap(example_sce, features=rownames(example_sce)[1:10],
    colour_columns_by=c("Mutation_Status", "Cell_Cycle"))
```

`plotHighestExprs` *Plot the highest expressing features*

Description

Plot the features with the highest average expression across all cells, along with their expression in each individual cell.

Usage

```
plotHighestExprs(object, n = 50, controls, colour_cells_by,
  drop_features = NULL, exprs_values = "counts",
  by_exprs_values = exprs_values, by_show_single = TRUE,
  feature_names_to_plot = NULL, as_percentage = TRUE)
```

Arguments

`object`	A SingleCellExperiment object.
`n`	A numeric scalar specifying the number of the most expressed features to show.
`controls`	Specification of the row-level metadata column indicating whether a feature is a control, see `?"scater-vis-var"` for possible values. Only metadata fields will be searched, assays will not be used. If not supplied, this defaults to `"is_feature_control"` or equivalent for compacted data.
`colour_cells_by`	Specification of a column metadata field or a feature to colour by, see `?"scater-vis-var"` for possible values. If not supplied, this defaults to `"total_features_by_counts"` or equivalent for compacted data.
`drop_features`	A character, logical or numeric vector indicating which features (e.g. genes, transcripts) to drop when producing the plot. For example, spike-in transcripts might be dropped to examine the contribution from endogenous genes.
`exprs_values`	A integer scalar or string specifying the assay to obtain expression values from.
`by_exprs_values`	A string or integer scalar specifying which assay to obtain expression values from, for use in colouring - see `?"scater-vis-var"` for details.
`by_show_single`	Logical scalar specifying whether single-level factors should be used for colouring, see `?"scater-vis-var"` for details. Default is `NULL`, in which case `rownames(object)` are used.
`feature_names_to_plot`	Specification of which row-level metadata column contains the feature names, see `?"scater-vis-var"` for possible values.
`as_percentage`	logical scalar indicating whether percentages should be plotted. If `FALSE`, the raw `exprs_values` are shown instead.

Details

This function will plot the percentage of counts accounted for by the top `n` most highly expressed features across the dataset. Each feature corresponds to a row on the plot, sorted by average expression (denoted by the point).

The plot will attempt to colour the points based on whether the corresponding feature is labelled as a control in `object`. This can be turned off by setting `controls=NULL`.

The distribution of expression across all cells is shown as tick marks for each feature. These ticks can be coloured according to cell-level metadata, as specified by `colour_cells_by`. Setting `colour_cells_by=NULL` will disable all tick colouring.

Value

A ggplot object.

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)
example_sce <- calculateQCMetrics(example_sce,
    feature_controls = list(set1 = 1:500)
)

plotHighestExprs(example_sce, colour_cells_by ="total_features")
plotHighestExprs(example_sce, controls = NULL)
plotHighestExprs(example_sce, colour_cells_by="Mutation_Status")
```

`plotPlatePosition` *Plot cells in plate positions*

Description

Plots cells in their position on a plate, coloured by metadata variables or feature expression values from a SingleCellExperiment object.

Usage

```
plotPlatePosition(object, plate_position = NULL, colour_by = NULL,
  size_by = NULL, shape_by = NULL, by_exprs_values = "logcounts",
  by_show_single = FALSE, legend = TRUE, theme_size = 24, alpha = 0.6,
  size = 24)
```

Arguments

`object`	A SingleCellExperiment object.

`plate_position` A character vector specifying the plate position for each cell (e.g., A01, B12, and so on, where letter indicates row and number indicates column). If `NULL`, the function will attempt to extract this from `object$plate_position`. Alternatively, a list of two factors (`"row"` and `"column"`) can be supplied, specifying the row and column for each cell in `object`.

`colour_by` Specification of a column metadata field or a feature to colour by, see ?`"scater-vis-var"` for possible values.

`size_by` Specification of a column metadata field or a feature to size by, see ?`"scater-vis-var"` for possible values.

`shape_by` Specification of a column metadata field or a feature to shape by, see ?`"scater-vis-var"` for possible values.

`by_exprs_values` A string or integer scalar specifying which assay to obtain expression values from, for use in point aesthetics - see ?`"scater-vis-var"` for details.

`by_show_single` Logical scalar specifying whether single-level factors should be used for point aesthetics, see ?`"scater-vis-var"` for details.

`legend` Logical scalar specifying whether a legend should be shown.

`theme_size` Numeric scalar, see ?`"scater-plot-args"` for details.

`alpha` Numeric scalar specifying the transparency of the points, see ?`"scater-plot-args"` for details.

`size` Numeric scalar specifying the size of the points, see ?`"scater-plot-args"` for details.

Details

This function expects plate positions to be given in a charcter format where a letter indicates the row on the plate and a numeric value indicates the column. Each cell has a plate position such as "A01", "B12", "K24" and so on. From these plate positions, the row is extracted as the letter, and the column as the numeric part. Alternatively, the row and column identities can be directly supplied by setting `plate_position` as a list of two factors.

Value

A ggplot object.

Author(s)

Davis McCarthy, with modifications by Aaron Lun

Examples

```
## prepare data
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)
example_sce <- normalize(example_sce)
example_sce <- calculateQCMetrics(example_sce)

## define plate positions
```

```
example_sce$plate_position <- paste0(
    rep(LETTERS[1:5], each = 8),
    rep(formatC(1:8, width = 2, flag = "0"), 5)
)

## plot plate positions
plotPlatePosition(example_sce, colour_by = "Mutation_Status")

plotPlatePosition(example_sce, shape_by = "Treatment", colour_by = "Gene_0004")

plotPlatePosition(example_sce, shape_by = "Treatment", size_by = "Gene_0001",
    colour_by = "Cell_Cycle")
```

`plotQC` *Produce QC diagnostic plots*

Description

Produce QC diagnostic plots

Usage

```
plotQC(object, type = "highest-expression", ...)
```

Arguments

`object`	an SingleCellExperiment object containing expression values and experimental information. Must have been appropriately prepared.
`type`	character scalar providing type of QC plot to compute: "highest-expression" (showing features with highest expression), "find-pcs" (showing the most important principal components for a given variable), "explanatory-variables" (showing a set of explanatory variables plotted against each other, ordered by marginal variance explained), or "exprs-mean-vs-freq" (plotting the mean expression levels against the frequency of expression for a set of features).
`...`	arguments passed to `plotHighestExprs`, `findImportantPCs`, `plotExplanatoryVariables` and `{plotExprsMeanVsFreq}` as appropriate.

Details

Display useful quality control plots to help with pre-processing of data and identification of potentially problematic features and cells.

Value

a ggplot plot object

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts),
colData = sc_example_cell_info)
example_sce <- normalize(example_sce)

drop_genes <- apply(exprs(example_sce), 1, function(x) {var(x) == 0})
example_sce <- example_sce[!drop_genes, ]
example_sce <- calculateQCMetrics(example_sce)
plotQC(example_sce, type="high", colour_cells_by="Mutation_Status")
plotQC(example_sce, type="find", variable="total_features")
vars <- names(colData(example_sce))[c(2:3, 5:14)]
plotQC(example_sce, type="expl", variables=vars)
```

`plotReducedDim` *Plot reduced dimensions*

Description

Plot cell-level reduced dimension results stored in a SingleCellExperiment object.

Usage

```
plotReducedDim(object, use_dimred, ncomponents = 2, percentVar = NULL,
  colour_by = NULL, shape_by = NULL, size_by = NULL,
  by_exprs_values = "logcounts", by_show_single = FALSE, ...,
  add_ticks = TRUE)
```

Arguments

`object`	A SingleCellExperiment object.
`use_dimred`	A string or integer scalar indicating the reduced dimension result in `reducedDims(object)` to plot.
`ncomponents`	A numeric scalar indicating the number of dimensions to plot, starting from the first dimension. Alternatively, a numeric vector specifying the dimensions to be plotted.
`percentVar`	A numeric vector giving the proportion of variance in expression explained by each reduced dimension. Only expected to be used in PCA settings, e.g., in the `plotPCA` function.
`colour_by`	Specification of a column metadata field or a feature to colour by, see ?"`scater-vis-var`" for possible values.
`shape_by`	Specification of a column metadata field or a feature to shape by, see ?"`scater-vis-var`" for possible values.
`size_by`	Specification of a column metadata field or a feature to size by, see ?"`scater-vis-var`" for possible values.
`by_exprs_values`	A string or integer scalar specifying which assay to obtain expression values from, for use in point aesthetics - see ?"`scater-vis-var`" for details.

by_show_single	Logical scalar specifying whether single-level factors should be used for point aesthetics, see ?"scater-vis-var" for details.
...	Additional arguments for visualization, see ?"scater-plot-args" for details.
add_ticks	Logical scalar indicating whether ticks should be drawn on the axes corresponding to the location of each point.

Details

If ncomponents is a scalar and equal to 2, a scatterplot of the first two dimensions is produced. If ncomponents is greater than 2, a pairs plots for the top dimensions is produced.

Alternatively, if ncomponents is a vector of length 2, a scatterplot of the two specified dimensions is produced. If it is of length greater than 2, a pairs plot is produced containing all pairwise plots between the specified dimensions.

Value

A ggplot object

Author(s)

Davis McCarthy, with modifications by Aaron Lun

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)
example_sce <- normalize(example_sce)

example_sce <- runPCA(example_sce, ncomponents=5)
plotReducedDim(example_sce, "PCA")
plotReducedDim(example_sce, "PCA", colour_by="Cell_Cycle")
plotReducedDim(example_sce, "PCA", colour_by="Gene_0001")

plotReducedDim(example_sce, "PCA", ncomponents=5)
plotReducedDim(example_sce, "PCA", ncomponents=5, colour_by="Cell_Cycle",
    shape_by="Treatment")
```

---

plotRLE *Plot a relative log expression (RLE) plot*

Description

Produce a relative log expression (RLE) plot of one or more transformations of cell expression values.

Usage

```
plotRLE(object, exprs_mats = list(logcounts = "logcounts"),
  exprs_logged = c(TRUE), colour_by = NULL, style = "minimal",
  legend = "auto", order_by_colour = TRUE, ncol = 1, ...)
```

Arguments

`object`	an `SingleCellExperiment` object
`exprs_mats`	named list of expression matrices. Entries can either be a character string, in which case the corresponding expression matrix will be extracted from the SingleCellExperiment `object`, or a matrix of expression values.
`exprs_logged`	logical vector of same length as `exprs_mats` indicating whether the corresponding entry in `exprs_mats` contains logged expression values (`TRUE`) or not (`FALSE`).
`colour_by`	character string defining the column of `colData(object)` to be used as a factor by which to colour the points in the plot. Alternatively, a data frame with one column, containing values to map to colours for all cells.
`style`	character(1), either `"minimal"` (default) or `"full"`, defining the boxplot style to use. `"minimal"` uses Tufte-style boxplots and is fast for large numbers of cells. `"full"` uses the usual `ggplot2` and is more detailed and flexible, but can take a long time to plot for large datasets.
`legend`	character, specifying how the legend(s) be shown? Default is `"auto"`, which hides legends that have only one level and shows others. Alternative is "none" (hide all legends).
`order_by_colour`	logical, should cells be ordered (grouped) by the `colour_by` variable? Default is `TRUE`. Useful for visualising differences between batches or experimental conditions.
`ncol`	integer, number of columns for the facetting of the plot. Default is 1.
`...`	further arguments passed to `geom_boxplot`.

Details

Unwanted variation can be highly problematic and so its detection is often crucial. Relative log expression (RLE) plots are a powerful tool for visualising such variation in high dimensional data. RLE plots are particularly useful for assessing whether a procedure aimed at removing unwanted variation, i.e. a normalisation procedure, has been successful. These plots, while originally devised for gene expression data from microarrays, can also be used to reveal unwanted variation in single-cell expression data, where such variation can be problematic.

If style is "full", as usual with boxplots, the box shows the inter-quartile range and whiskers extend no more than 1.5 * IQR from the hinge (the 25th or 75th percentile). Data beyond the whiskers are called outliers and are plotted individually. The median (50th percentile) is shown with a white bar.

If style is "minimal", then median is shown with a circle, the IQR in a grey line, and "whiskers" (as defined above) for the plots are shown with coloured lines. No outliers are shown for this plot style.

Value

a ggplot plot object

Author(s)

Davis McCarthy

References

Gandolfo LC, Speed TP. RLE Plots: Visualising Unwanted Variation in High Dimensional Data. arXiv [stat.ME]. 2017. Available: http://arxiv.org/abs/1704.03590

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
 example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts),
colData = sc_example_cell_info)
example_sce <- normalize(example_sce)
drop_genes <- apply(logcounts(example_sce), 1, function(x) {var(x) == 0})
example_sce <- example_sce[!drop_genes, ]

plotRLE(example_sce, list(logcounts= "logcounts", counts = "counts"), c(TRUE, FALSE),
       colour_by = "Mutation_Status", style = "minimal")

plotRLE(example_sce, list(logcounts = "logcounts", counts = "counts"), c(TRUE, FALSE),
       colour_by = "Mutation_Status", style = "full",
       outlier.alpha = 0.1, outlier.shape = 3, outlier.size = 0)
```

`plotRowData` *Plot row metadata*

Description

Plot row-level (i.e., gene) metadata from a SingleCellExperiment object.

Usage

```
plotRowData(object, y, x = NULL, colour_by = NULL, shape_by = NULL,
  size_by = NULL, by_exprs_values = "logcounts", by_show_single = FALSE,
  ...)

plotFeatureData(...)
```

Arguments

`object`	A SingleCellExperiment object containing expression values and experimental information.
`y`	Specification of the row-level metadata to show on the y-axis, see ?"`scater-vis-var`" for possible values. Note that only metadata fields will be searched, assays will not be used.
`x`	Specification of the row-level metadata to show on the x-axis, see ?"`scater-vis-var`" for possible values. Again, only metadata fields will be searched, assays will not be used.
`colour_by`	Specification of a row metadata field or a cell to colour by, see ?"`scater-vis-var`" for possible values.

shape_by
: Specification of a row metadata field or a cell to shape by, see ?"scater-vis-var" for possible values.

size_by
: Specification of a row metadata field or a cell to size by, see ?"scater-vis-var" for possible values.

by_exprs_values
: A string or integer scalar specifying which assay to obtain expression values from, for use in point aesthetics - see ?"scater-vis-var" for details.

by_show_single
: Logical scalar specifying whether single-level factors should be used for point aesthetics, see ?"scater-vis-var" for details.

...
: Additional arguments for visualization, see ?"scater-plot-args" for details.

Details

If y is continuous and x=NULL, a violin plot is generated. If x is categorical, a grouped violin plot will be generated, with one violin for each level of x. If x is continuous, a scatter plot will be generated.

If y is categorical and x is continuous, horizontal violin plots will be generated. If x is missing or categorical, rectangule plots will be generated where the area of a rectangle is proportional to the number of points for a combination of factors.

Note that plotFeatureData is a synonym for plotRowData. This is an artifact of the transition from the old SCESet class, and will be deprecated in future releases.

Value

A ggplot object.

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)
example_sce <- calculateQCMetrics(example_sce,
    feature_controls = list(ERCC=1:40))
example_sce <- normalize(example_sce)

plotRowData(example_sce, y="n_cells_by_counts", x="log10_total_counts")
plotRowData(example_sce, y="n_cells_by_counts",
   size_by ="log10_total_counts",
   colour_by = "is_feature_control")
```

---

plotScater — *Plot an overview of expression for each cell*

Description

Plot the relative proportion of the library size that is accounted for by the most highly expressed features for each cell in a SingleCellExperiment object.

Usage

```
plotScater(x, nfeatures = 500, exprs_values = "counts", colour_by = NULL,
  by_exprs_values = exprs_values, by_show_single = FALSE, block1 = NULL,
  block2 = NULL, ncol = 3, line_width = 1.5, theme_size = 10)
```

Arguments

`x`	A SingleCellExperiment object.
`nfeatures`	Numeric scalar indicating the number of top-expressed features to show n the plot.
`exprs_values`	String or integer scalar indicating which assay of `object` should be used to obtain the expression values for this plot.
`colour_by`	Specification of a column metadata field or a feature to colour by, see ?"`scater-vis-var`" for possible values. The curve for each cell will be coloured according to this specification.
`by_exprs_values`	A string or integer scalar specifying which assay to obtain expression values from, for use in line colouring - see ?"`scater-vis-var`" for details.
`by_show_single`	Logical scalar specifying whether single-level factors should be used for line colouring, see ?"`scater-vis-var`" for details.
`block1`	Specification of a factor by which to separate the cells into blocks (separate panels) in the plot. This can be any type of value described in ?"`scater-vis-var`" for column-level metadata. Default is `NULL`, in which case there is no blocking.
`block2`	Same as `block1`, providing another level of blocking.
`ncol`	Number of columns to use for `facet_wrap` if only one block is defined.
`line_width`	Numeric scalar specifying the line width.
`theme_size`	Numeric scalar specifying the font size to use for the plotting theme.

Details

For each cell, the features are ordered from most-expressed to least-expressed. The cumulative proportion of the total expression for the cell is computed across the top `nfeatures` features. These plots can flag cells with a very high proportion of the library coming from a small number of features; such cells are likely to be problematic for downstream analyses.

Using the colour and blocking arguments can flag overall differences in cells under different experimental conditions or affected by different batch and other variables. If only one of `block1` and `block2` are specified, each panel corresponds to a separate level of the specified blocking factor. If both are specified, each panel corresponds to a combination of levels.

Value

a ggplot plot object

Author(s)

Davis McCarthy, with modifications by Aaron Lun

Examples

```
## Set up an example SingleCellExperiment
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)

plotScater(example_sce)
plotScater(example_sce, exprs_values = "counts", colour_by = "Cell_Cycle")
plotScater(example_sce, block1 = "Treatment", colour_by = "Cell_Cycle")

cpm(example_sce) <- calculateCPM(example_sce, use_size_factors = FALSE)
plotScater(example_sce, exprs_values = "cpm", block1 = "Treatment",
    block2 = "Mutation_Status", colour_by = "Cell_Cycle")
```

---

read10xResults *Load in data from 10x experiment*

Description

Creates a full or sparse matrix from a sparse data matrix provided by 10X genomics.

Usage

```
read10xResults(data_dir, min_total_cell_counts = NULL,
  min_mean_gene_counts = NULL)

read10XResults(...)
```

Arguments

`data_dir`	Directory containing the matrix.mtx, genes.tsv, and barcodes.tsv files provided by 10x. A vector or named vector can be given in order to load several data directories. If a named vector is given, the cell barcode names will be prefixed with the name.
`min_total_cell_counts`	integer(1) threshold such that cells (barcodes) with total counts below the threshold are filtered out
`min_mean_gene_counts`	numeric(1) threshold such that genes with mean counts below the threshold are filtered out.
`...`	passed arguments

Details

This function calls `read10xCounts` from the **DropletUtils** package. It is deprecated and will be removed in the next release.

Value

Returns an SingleCellExperiment object with counts data stored as a sparse matrix. Rows are named with the gene name and columns are named with the cell barcode (if data_dir contains one element; otherwise the columns are unnamed to avoid problems with non-unique barcodes).

Examples

```
sce10x <- read10xResults(system.file("extdata", package="scater"))
```

---

readTxResults — *Read transcript quantification data with tximport package*

Description

After generating transcript/feature abundance results using kallisto, Salmon, Sailfish or RSEM for a batch of samples, read these abundance values into an SCESet object.

Usage

```
readTxResults(samples = NULL, files = NULL, log = NULL,
  type = "kallisto", txOut = TRUE, logExprsOffset = 1, verbose = TRUE,
  ...)
```

Arguments

samples	character vector providing a set of sample names to use for the abundance results.
files	character vector providing a set of filenames containing kallisto abundance results to be read in.
log	list (optional), generated by runKallisto. If provided, then samples and files arguments are ignored.
type	character, the type of software used to generate the abundances. Options are "kallisto", "salmon", "sailfish", "rsem". This argument is passed to tximport.
txOut	logical, whether the function should just output transcript-level (default TRUE)
logExprsOffset	numeric scalar, providing the offset used when doing log2-transformations of expression data to avoid trying to take logs of zero. Default offset value is 1.
verbose	logical, should function provide output about progress?
...	optional parameters passed to tximport. See documentation for tximport for options and details.

Details

Note: tximport does not import bootstrap estimates from kallisto, Salmon, or Sailfish. If you want bootstrap estimates use the readKallistoResults or readSalmonResults functions.

Value

an SCESet object containing the abundance, count and feature length data from the supplied samples.

References

Soneson C, Love MI, Robinson MD. Differential analyses for RNA-seq: transcript-level estimates improve gene-level inferences. F1000Res. 2015;4: 1521.

Examples

```
## Not run:
## this example requires installation of the tximportData package from
## Bioconductor
library(tximportData)
dir <- system.file("extdata", package = "tximportData")
list.files(dir)
samples <- read.table(file.path(dir, "samples.txt"), header = TRUE)
samples
directories <- file.path(dir, "kallisto", samples$run)
names(directories) <- paste0("sample", 1:6)
files <- file.path(directories, "abundance.tsv")
sce_example <- readTxResults(samples = names(directories),
files = files, type = "kallisto")

## for faster reading of results use the read_tsv function from the readr pkg
library(readr)
sce_example <- readTxResults(samples = names(directories),
files = files, type = "kallisto", reader = read_tsv)

## End(Not run)
```

`Reduced dimension plots` *Plot specific reduced dimensions*

Description

Wrapper functions to create plots for specific types of reduced dimension results in a SingleCellExperiment object, or, if they are not already present, to calculate those results and then plot them.

Usage

```
plotPCASCE(object, ..., return_SCE = FALSE, draw_plot = TRUE,
  rerun = FALSE, ncomponents = 2, run_args = list())

plotTSNE(object, ..., return_SCE = FALSE, draw_plot = TRUE, rerun = FALSE,
  ncomponents = 2, run_args = list())

plotDiffusionMap(object, ..., return_SCE = FALSE, draw_plot = TRUE,
  rerun = FALSE, ncomponents = 2, run_args = list())

plotMDS(object, ..., ncomponents = 2, return_SCE = FALSE, rerun = FALSE,
  draw_plot = TRUE, run_args = list())

## S4 method for signature 'SingleCellExperiment'
plotPCA(object, ..., return_SCE = FALSE,
  draw_plot = TRUE, rerun = FALSE, ncomponents = 2, run_args = list())
```

Arguments

object	A SingleCellExperiment object.
...	Additional arguments to pass to plotReducedDim.
return_SCE	Logical, should the function return a SingleCellExperiment object with reduced dimension results in the reducedDim slot? Default is FALSE, in which case a ggplot object is returned. This will be deprecated in the next release in favour of directly calling the underlying run* functions to compute the results.
draw_plot	Logical, should the plot be drawn on the current graphics device? Only used if return_SCE is TRUE, otherwise the plot is always produced.
rerun	Logical, should the reduced dimensions be recomputed even if object contains an appropriately named set of results in the reducedDims slot?
ncomponents	Numeric scalar indicating the number of dimensions components to (calculate and) plot. This can also be a numeric vector, see ?plotReducedDim for details.
run_args	Arguments to pass to runPCA.

Details

Each function will search the reducedDims slot for an appropriately named set of results and pass those coordinates onto plotReducedDim. If the results are not present or rerun=TRUE, they will be computed using the relevant run* function. The result name and run* function for each plot* function are:

- "PCA" and runPCA for plotPCA
- "TSNE" and runTSNE for plotTSNE
- "DiffusionMap" and runDiffusionMap for plotDiffusionMap
- "MDS" and runMDS for "plotMDS"

Users can specify arguments to the run* functions via run_args.

If ncomponents is a numeric vector, the maximum value will be used to determine the required number of dimensions to compute in the run* functions. However, only the specified dimensions in ncomponents will be plotted.

Value

A ggplot object or an SingleCellExperiment object, depending on return_SCE.

Author(s)

Davis McCarthy, with modifications by Aaron Lun

See Also

runPCA, runDiffusionMap, runTSNE, runMDS, plotReducedDim

Examples

```
## Set up an example SingleCellExperiment
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
```

```
    colData = sc_example_cell_info
)
example_sce <- normalize(example_sce)

## Examples plotting PC1 and PC2
plotPCA(example_sce)
plotPCA(example_sce, colour_by = "Cell_Cycle")
plotPCA(example_sce, colour_by = "Cell_Cycle", shape_by = "Treatment")
plotPCA(example_sce, colour_by = "Cell_Cycle", shape_by = "Treatment",
    size_by = "Mutation_Status")

## Force legend to appear for shape:
example_subset <- example_sce[, example_sce$Treatment == "treat1"]
plotPCA(example_subset, colour_by = "Cell_Cycle", shape_by = "Treatment",
    by_show_single = TRUE)

## Examples plotting more than 2 PCs
plotPCA(example_sce, ncomponents = 4, colour_by = "Treatment",
    shape_by = "Mutation_Status")

## Same for TSNE:
plotTSNE(example_sce, perplexity = 10)

## Same for DiffusionMaps:
plotDiffusionMap(example_sce)

## Same for MDS plots:
plotMDS(example_sce)
```

---

rename *Rename variables of* `colData(object)`.

Description

Rename variables of `colData(object)`.

Usage

```
rename(object, ...)

## S4 method for signature 'SingleCellExperiment'
rename(object, ...)
```

Arguments

object	A `SingleCellExperiment` object.
...	Additional arguments to be passed to `dplyr::rename` to act on `colData(object)`.

Value

An SingleCellExperiment object.

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts),
colData = sc_example_cell_info)
example_sce <- rename(example_sce, Cell_Phase = Cell_Cycle)
```

---

runDiffusionMap *Create a diffusion map from cell-level data*

Description

Produce a diffusion map for the cells, based on the data in a SingleCellExperiment object.

Usage

```
runDiffusionMap(object, ncomponents = 2, ntop = 500, feature_set = NULL,
  exprs_values = "logcounts", scale_features = TRUE, use_dimred = NULL,
  n_dimred = NULL, rand_seed = NULL, ...)
```

Arguments

`object`	A SingleCellExperiment object
`ncomponents`	Numeric scalar indicating the number of diffusion components to obtain.
`ntop`	Numeric scalar specifying the number of most variable features to use for constructing the diffusion map.
`feature_set`	Character vector of row names, a logical vector or a numeric vector of indices indicating a set of features to use to construct the diffusion map. This will override any `ntop` argument if specified.
`exprs_values`	Integer scalar or string indicating which assay of `object` should be used to obtain the expression values for the calculations.
`scale_features`	Logical scalar, should the expression values be standardised so that each feature has unit variance?
`use_dimred`	String or integer scalar specifying the entry of `reducedDims(object)` to use as input to `DiffusionMap`. Default is to not use existing reduced dimension results.
`n_dimred`	Integer scalar, number of dimensions of the reduced dimension slot to use when `use_dimred` is supplied. Defaults to all available dimensions.
`rand_seed`	Numeric scalar that can be passed to `set.seed` to make the results reproducible.
`...`	Additional arguments to pass to `DiffusionMap`.

Details

The function `DiffusionMap` is used internally to compute the diffusion map.

Setting `use_dimred` allows users to easily construct a diffusion map from low-rank approximations of the original expression matrix (e.g., after PCA). In such cases, arguments such as `ntop`, `feature_set`, `exprs_values` and `scale_features` will be ignored.

Value

A SingleCellExperiment object containing the coordinates of the first `ncomponent` diffusion map components for each cell. This is stored in the `"DiffusionMap"` entry of the `reducedDims` slot.

Author(s)

Aaron Lun, based on code by Davis McCarthy

References

Haghverdi L, Buettner F, Theis FJ. Diffusion maps for high-dimensional single-cell analysis of differentiation data. Bioinformatics. 2015; doi:10.1093/bioinformatics/btv325

See Also

`destiny`, `plotDiffusionMap`

Examples

```
## Set up an example SingleCellExperiment
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)
example_sce <- normalize(example_sce)

example_sce <- runDiffusionMap(example_sce)
reducedDimNames(example_sce)
head(reducedDim(example_sce))
```

---

runMDS *Perform MDS on cell-level data*

Description

Perform multi-dimensional scaling (MDS) on cells, based on the data in a SingleCellExperiment object.

Usage

```
runMDS(object, ncomponents = 2, ntop = 500, feature_set = NULL,
  exprs_values = "logcounts", scale_features = TRUE, use_dimred = NULL,
  n_dimred = NULL, method = "euclidean")
```

Arguments

`object`	A SingleCellExperiment object.
`ncomponents`	Numeric scalar indicating the number of MDS dimensions to obtain.
`ntop`	Numeric scalar specifying the number of most variable features to use for MDS.
`feature_set`	Character vector of row names, a logical vector or a numeric vector of indices indicating a set of features to use for MDS. This will override any `ntop` argument if specified.
`exprs_values`	Integer scalar or string indicating which assay of `object` should be used to obtain the expression values for the calculations.
`scale_features`	Logical scalar, should the expression values be standardised so that each feature has unit variance?
`use_dimred`	String or integer scalar specifying the entry of `reducedDims(object)` to use as input to `cmdscale`. Default is to not use existing reduced dimension results.
`n_dimred`	Integer scalar, number of dimensions of the reduced dimension slot to use when `use_dimred` is supplied. Defaults to all available dimensions.
`method`	String specifying the type of distance to be computed between cells.

Details

The function `cmdscale` is used internally to compute the multidimensional scaling components to plot.

Setting `use_dimred` allows users to easily perform MDS on low-rank approximations of the original expression matrix (e.g., after PCA). In such cases, arguments such as `ntop`, `feature_set`, `exprs_values` and `scale_features` will be ignored.

Value

A SingleCellExperiment object containing the coordinates of the first `ncomponent` MDS dimensions for each cell. This is stored in the `"MDS"` entry of the `reducedDims` slot.

Author(s)

Aaron Lun, based on code by Davis McCarthy

See Also

`cmdscale`, `plotMDS`

Examples

```
## Set up an example SingleCellExperiment
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)
example_sce <- normalize(example_sce)

example_sce <- runMDS(example_sce)
reducedDimNames(example_sce)
head(reducedDim(example_sce))
```

runPCA *Perform PCA on cell-level data*

Description

Perform a principal components analysis (PCA) on cells, based on the data in a SingleCellExperiment object.

Usage

```
runPCA(object, ncomponents = 2, method = c("prcomp", "irlba"), ntop = 500,
  exprs_values = "logcounts", feature_set = NULL, scale_features = TRUE,
  use_coldata = FALSE, selected_variables = NULL, detect_outliers = FALSE,
  rand_seed = NULL, ...)
```

Arguments

`object`	A SingleCellExperiment object.
`ncomponents`	Numeric scalar indicating the number of principal components to obtain.
`method`	String specifying how the PCA should be performed.
`ntop`	Numeric scalar specifying the number of most variable features to use for PCA.
`exprs_values`	Integer scalar or string indicating which assay of `object` should be used to obtain the expression values for the calculations.
`feature_set`	Character vector of row names, a logical vector or a numeric vector of indices indicating a set of features to use for PCA. This will override any `ntop` argument if specified.
`scale_features`	Logical scalar, should the expression values be standardised so that each feature has unit variance?
`use_coldata`	Logical scalar specifying whether the column data should be used instead of expression values to perform PCA.
`selected_variables`	List of strings or a character vector indicating which variables in `colData(object)` to use for PCA when `use_coldata=TRUE`. If a list, each entry can take the form described in `?"scater-vis-var"`.
`detect_outliers`	Logical scalar, should outliers be detected based on PCA coordinates generated from column-level metadata?
`rand_seed`	Numeric scalar specifying the random seed when using `method="irlba"`.
`...`	Additional arguments to pass to `prcomp_irlba` when `method="irlba"`.

Details

The function `prcomp` is used internally to do the PCA when `method="prcomp"`. Alternatively, the **irlba** package can be used, which performs a fast approximation of PCA through the `prcomp_irlba` function. This is especially useful for large, sparse matrices.

If `use_coldata=TRUE`, PCA will be performed on column-level metadata. The `selected_variables` defaults to a vector containing:

- "pct_counts_top_100_features"
- "total_features"
- "pct_counts_feature_control"
- "total_features_feature_control"
- "log10_total_counts_endogenous"
- "log10_total_counts_feature_control"

This can be useful for identifying outliers cells based on QC metrics, especially when combined with detect_outliers=TRUE. If outlier identification is enabled, the outlier field of the output colData will contain the identified outliers.

Value

A SingleCellExperiment object containing the first ncomponent principal coordinates for each cell. If use_coldata=FALSE, this is stored in the "PCA" entry of the reducedDims slot. Otherwise, it is stored in the "PCA_coldata" entry.

The proportion of variance explained by each PC is stored as a numeric vector in the "percentVar" attribute of the reduced dimension matrix. Note that this will only be of length equal to ncomponents when method is not "prcomp". This is because approximate PCA methods do not compute singular values for all components.

Author(s)

Aaron Lun, based on code by Davis McCarthy

See Also

prcomp, plotPCA

Examples

```
## Set up an example SingleCellExperiment
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)
example_sce <- normalize(example_sce)

example_sce <- runPCA(example_sce)
reducedDimNames(example_sce)
head(reducedDim(example_sce))
```

runTSNE *Perform t-SNE on cell-level data*

Description

Perform t-stochastic neighbour embedding (t-SNE) for the cells, based on the data in a SingleCell-Experiment object.

Usage

```
runTSNE(object, ncomponents = 2, ntop = 500, feature_set = NULL,
  exprs_values = "logcounts", scale_features = TRUE, use_dimred = NULL,
  n_dimred = NULL, rand_seed = NULL, perplexity = min(50,
  floor(ncol(object)/5)), pca = TRUE, initial_dims = 50, ...)
```

Arguments

object	A SingleCellExperiment object.
ncomponents	Numeric scalar indicating the number of t-SNE dimensions to obtain.
ntop	Numeric scalar specifying the number of most variable features to use for t-SNE.
feature_set	Character vector of row names, a logical vector or a numeric vector of indices indicating a set of features to use for t-SNE. This will override any `ntop` argument if specified.
exprs_values	Integer scalar or string indicating which assay of `object` should be used to obtain the expression values for the calculations.
scale_features	Logical scalar, should the expression values be standardised so that each feature has unit variance?
use_dimred	String or integer scalar specifying the entry of `reducedDims(object)` to use as input to `Rtsne`. Default is to not use existing reduced dimension results.
n_dimred	Integer scalar, number of dimensions of the reduced dimension slot to use when `use_dimred` is supplied. Defaults to all available dimensions.
rand_seed	Numeric scalar that can be passed to `set.seed` to make the results reproducible.
perplexity	Numeric scalar defining the perplexity parameter, see ?`Rtsne` for more details.
pca	Logical scalar passed to `Rtsne`, indicating whether an initial PCA step should be performed. This is ignored if `use_dimred` is specified.
initial_dims	Integer scalar passed to `Rtsne`, specifying the number of principal components to be retained if `pca=TRUE`.
...	Additional arguments to pass to `Rtsne`.

Details

The function `Rtsne` is used internally to compute the t-SNE. Note that the algorithm is not deterministic, so different runs of the function will produce differing results. Users are advised to test multiple random seed, and then use `rand_seed` to set a random seed for replicable results.

The value of the `perplexity` parameter can have a large effect on the results. By default, the function will try to provide a reasonable setting, by scaling the perplexity with the number of cells until it reaches a maximum of 50. However, it is often worthwhile to manually try multiple values to ensure that the conclusions are robust.

Setting use_dimred allows users to easily perform t-SNE on low-rank approximations of the original expression matrix (e.g., after PCA). In such cases, arguments such as ntop, feature_set, exprs_values and scale_features will be ignored.

Value

A SingleCellExperiment object containing the coordinates of the first ncomponent t-SNE dimensions for each cell. This is stored in the "TSNE" entry of the reducedDims slot.

Author(s)

Aaron Lun, based on code by Davis McCarthy

References

L.J.P. van der Maaten. Barnes-Hut-SNE. In Proceedings of the International Conference on Learning Representations, 2013.

See Also

Rtsne, plotTSNE

Examples

```
## Set up an example SingleCellExperiment
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
    assays = list(counts = sc_example_counts),
    colData = sc_example_cell_info
)
example_sce <- normalize(example_sce)

example_sce <- runTSNE(example_sce)
reducedDimNames(example_sce)
head(reducedDim(example_sce))
```

---

salmon-wrapper *Salmon wrapper functions*

Description

Salmon wrapper functions

After generating transcript/feature abundance results using Salmon for a batch of samples, read these abundance values into a SingleCellExperiment object.

Run the abundance quantification tool Salmon on a set of FASTQ files. Requires Salmon (https://combine-lab.github.io/salmon/) to be installed and a Salmon transcript index must have been generated prior to using this function. See the Salmon website for installation and basic usage instructions.

Usage

```
readSalmonResultsOneSample(directory)

readSalmonResults(Salmon_log = NULL, samples = NULL, directories = NULL,
  logExprsOffset = 1, verbose = TRUE)

runSalmon(targets_file, transcript_index, single_end = FALSE,
  output_prefix = "output", lib_type = "A", n_processes = 2,
  n_thread_per_process = 4, n_bootstrap_samples = 0, seqBias = TRUE,
  gcBias = TRUE, posBias = FALSE, allowOrphans = FALSE,
  advanced_opts = NULL, verbose = TRUE, dry_run = FALSE,
  salmon_cmd = "salmon")
```

Arguments

- `directory` character string giving the path to the directory containing the Salmon results for the sample.
- `Salmon_log` list, generated by `runSalmon`. If provided, then `samples` and `directories` arguments are ignored.
- `samples` character vector providing a set of sample names to use for the abundance results.
- `directories` character vector providing a set of directories containing Salmon abundance results to be read in.
- `logExprsOffset` numeric scalar, providing the offset used when doing log2-transformations of expression data to avoid trying to take logs of zero. Default offset value is `1`.
- `verbose` logical, should function provide output about progress?
- `targets_file` character string giving the path to a tab-delimited text file with either 2 columns (single-end reads) or 3 columns (paired-end reads) that gives the sample names (first column) and FastQ file names (column 2 and if applicable 3). The file is assumed to have column headers, although these are not used.
- `transcript_index` character string giving the path to the Salmon index to be used for the feature abundance quantification.
- `single_end` logical, are single-end reads used, or paired-end reads?
- `output_prefix` character string giving the prefix for the output folder that will contain the Salmon results. The default is `"output"` and the sample name (column 1 of `targets_file`) is appended (preceded by an underscore).
- `lib_type` scalar, indicating RNA-seq library type. See Salmon documentation for details. Default is "A", for automatic detection.
- `n_processes` integer giving the number of processes to use for parallel Salmon jobs across samples. The package `parallel` is used. Default is 2 concurrent processes.
- `n_thread_per_process` integer giving the number of threads for Salmon to use per process (to parallelize Salmon for a given sample). Default is 4.
- `n_bootstrap_samples` integer giving the number of bootstrap samples that Salmon should use (default is 0). With bootstrap samples, uncertainty in abundance can be quantified.
- `seqBias` logical, should Salmon's option be used to model and correct abundances for sequence specific bias? Default is `TRUE`.

`gcBias`	logical, should Salmon's option be used to model and correct abundances for GC content bias? Requires Salmon version 0.7.2 or higher. Default is `TRUE`.
`posBias`	logical, should Salmon's option be used to model and correct abundances for positional biases? Requires Salmon version 0.7.3 or higher. Default is `FALSE`.
`allowOrphans`	logical, Consider orphaned reads as valid hits when performing lightweight-alignment. This option will increase sensitivity (allow more reads to map and more transcripts to be detected), but may decrease specificity as orphaned alignments are more likely to be spurious. For more details see Salmon documentation.
`advanced_opts`	character scalar supplying list of advanced option arguments to apply to each Salmon call. For details see Salmon documentation or type `salmon quant --help-reads` at the command line.
`dry_run`	logical, if `TRUE` then a list containing the Salmon commands that would be run and the output directories is returned. Can be used to read in results if Salmon is run outside an R session or to produce a script to run outside of an R session.
`salmon_cmd`	(optional) string giving full command to use to call Salmon, if simply typing "salmon" at the command line does not give the required version of Salmon or does not work. Default is simply "salmon". If used, this argument should give the full path to the desired Salmon binary.

Details

The directory is expected to contain results for just a single sample. Putting more than one sample's results in the directory will result in unpredictable behaviour with this function. The function looks for the files (with the default names given by Salmon) 'quant.sf', 'stats.tsv', 'libFormatCounts.txt' and the sub-directories 'logs' (which contains a log file) and 'libParams' (which contains a file detailing the fragment length distribution). If these files are missing, or if results files have different names, then this function will not find them.

This function will work for Salmon v0.7.x and greater, as the name of one of the default output directories was changed from "aux" to "aux_info" in Salmon v0.7.

This function expects to find only one set of Salmon abundance results per directory; multiple adundance results in a given directory will be problematic.

A Salmon transcript index can be built from a FASTA file: `salmon index [arguments] FASTA-file`. See the Salmon documentation for further details. This simple wrapper does not give access to all nuances of Salmon usage. For finer-grained usage of Salmon please run it at the command line - results can still be read into R with `readSalmonResults`.

Value

A list with two elements: (1) a data.frame `abundance` with columns for 'target_id' (feature, transcript, gene etc), 'length' (feature length), 'est_counts' (estimated feature counts), 'tpm' (transcripts per million); (2) a list, `run_info`, with metadata about the Salmon run that generated the results, including number of reads processed, mapping percentage, the library type used for the RNA-sequencing, including details about number of reads that did not match the given or inferred library type, details about the Salmon command used to generate the results, and so on.

an SingleCellExperiment object

A list containing three elements for each sample for which feature abundance has been quantified: (1) `salmon_call`, the call used for Salmon, (2) `salmon_log` the log generated by Salmon, and (3) `output_dir` the directory in which the Salmon results can be found.

Examples

```
## Not run:
# If Salmon results are in the directory "output", then call:
readSalmonResultsOneSample("output")

## End(Not run)
## Not run:
## Define output directories in a vector called here "Salmon_dirs"
## and sample names as "Salmon_samples"
sceset <- readSalmonResults(samples = Salmon_samples,
directories = Salmon_dirs)

## End(Not run)

## Not run:
## If in Salmon's 'test' directory, then try these calls:
## Generate 'targets.txt' file:
write.table(data.frame(Sample="sample1", File1="reads_1.fastq.gz", File2="reads_1.fastq.gz"),
 file="targets.txt", quote=FALSE, row.names=FALSE, sep="\t")
Salmon_log <- runSalmon("targets.txt", "transcripts.idx", single_end=FALSE,
         output_prefix="output", verbose=TRUE, n_bootstrap_samples=10,
         dry_run = FALSE)

## End(Not run)
```

---

scater-plot-args *General visualization parameters*

Description

scater functions that plot points share a number of visualization parameters, which are described on this page.

Aesthetic parameters

`legend`**:** Logical scalar, specifying whether a legend should be shown. Defaults to TRUE.

`theme_size`**:** Integer scalar, specifying the font size. Defaults to 10.

`alpha`**:** Numeric scalar in [0, 1], specifying the transparency. Defaults to 0.6.

`size`**:** Numeric scalar, specifying the size of the points. Defaults to `NULL`.

`jitter`**:** String to define whether points are to be jittered (`"jitter"`) or presented in a “beeswarm” style (if `"swarm"`, default). The latter usually looks more attractive, but for datasets with a large number of cells, or for dense plots, the jitter option may work better.

Distributional calculations

`show_median`**:** Logical, should the median of the distribution be shown for violin plots? Defaults to `FALSE`.

`show_violin`**:** Logical, should the outline of a violin plot be shown? Defaults to `TRUE`.

`show_smooth`**:** Logical, should a smoother be fitted to a scatter plot? Defaults to `FALSE`.

`show_se`**:** Logical, should standard errors for the fitted line be shown on a scatter plot when `show_smooth=TRUE`? Defaults to `TRUE`.

See Also

plotColData, plotRowData, plotReducedDim, plotExpression, plotPlatePosition, and most other plotting functions.

---

scater-vis-var *Variable selection for visualization*

Description

A number of **scater** functions accept a SingleCellExperiment object and extract (meta)data from it for use in a plot. These values are then used on the x- or y-axes (e.g., plotColData) or for tuning visual parameters, e.g., colour_by, shape_by, size_by. This page describes how the selection of these values can be controlled by the user, by passing appropriate values to the arguments of the desired plotting function.

When plotting by cells

Here, we assume that each visual feature of interest (e.g., point or line) corresponds to a cell in the SingleCellExperiment object sce. We will also assume that the user wants to change the colour of each feature according to the cell (meta)data. To do so, the user can pass to colour_by:

- An unnamed character string. This is initially assumed to be the name of a column-level metadata field. The function will first search the column names of colData(sce), and extract metadata for all cells if a matching field is found. If no match is found, the function will assume that the string represents a gene name. It will search rownames(sce) and extract gene expression values for any matching row across all cells. Otherwise, an error is raised.
- A named character string, where the name is either "exprs" or "metadata". This forces the function to only search for the string in rownames(sce) or colnames(colData(sce)), respectively. Adding an explicit name is useful when the same field exists in both the row names and column metadata names.
- A character vector of length greater than 1. This will search for nested fields in colData(sce). For example, supplying a character vector c("A", "B", "C") will retrieve colData(sce)AB$C, where both A and B contain nested DataFrames. See calculateQCMetrics with compact=TRUE for an example of how these can be constructed. The concatenated name "A:B:C" will be used in the legend.
- A data frame with one column and number of rows equal to the number of cells. This should contain values to use for visualization (in this case, for colouring by). In this manner, the user can use new information without manually adding it to the SingleCellExperiment object. The column name of the data frame will be used in the legend.

The same logic applies for other visualization parameters such as shape_by and size_by. Other arguments may also use the same scheme, but this depends on the context; see the documentation for each function for details. In particular, if an argument explicitly refers to a metadata field, any names for the character string will be ignored. Similarly, a character vector of length > 1 is not allowed for an argument that explicitly refers to expression values.

When plotting by features

Here, we assume that each visual feature of interest (e.g., point or line) corresponds to a feature in the SingleCellExperiment object `sce`. The scheme is mostly the same as described above, with a few differences:

- `rowData` is searched instead of `colData`, as we are extracting metadata for each feature.
- When extracting expression values, the name of a single cell must be specified. Visualization will then use the expression profile for all features in that cell. (This tends to be a rather unusual choice for colouring.)
- Character strings named with `"exprs"` will search for the string in `colnames(sce)`.
- A data frame input should have number of rows equal to the number of features.

Miscellaneous details

Most functions will have a `by_exprs_values` parameter. This defines the assay of the SingleCellExperiment object from which expression values are extracted for use in colouring, shaping or sizing the points. The setting of `by_exprs_values` will usually default to `"logcounts"`, or to the value of `exprs_values` in functions such as `plotExpression`. However, it can be specified separately from `exprs_values`, which is useful for visualizing two different types of expression values on the same plot.

Most functions will also have a `by_show_single` parameter. If `FALSE`, variables with only one level are not used for visualization, i.e., the visual aspect (colour or shape or size) is set to the default for all points. No guide is created for this aspect, avoiding clutter in the legend when that aspect provides no information. If `TRUE`, all supplied variables are used for visualization, regardless of how many levels they have.

See Also

`plotColData`, `plotRowData`, `plotReducedDim`, `plotExpression`, `plotPlatePosition`, and most other plotting functions.

`scater_gui` *scater GUI function*

Description

scater shiny app GUI for workflow for less programmatically inclined users or those who would like a quick and easy way to view multiple plots.

Usage

```
scater_gui(object)
```

Arguments

`object`	SinglCellExperiment object after running `calculateQCMetrics` on it

Value

Opens a browser window with an interactive shiny app and visualize all possible plots included in the scater

Author(s)

Davis McCarthy and Vladimir Kiselev

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts), colData = sc_example_cell_info)
example_sce <- normalize(example_sce)
drop_genes <- apply(exprs(example_sce), 1, function(x) {var(x) == 0})
example_sce <- example_sce[!drop_genes, ]
example_sce <- calculateQCMetrics(example_sce,
feature_controls = list(set1 = 1:40))
## Not run:
scater_gui(example_sce)

## End(Not run)
```

`SCESet` *The "Single Cell Expression Set" (SCESet) class*

Description

S4 class and the main class used by scater to hold single cell expression data. SCESet extends the basic Bioconductor ExpressionSet class.

Details

This class is initialized from a matrix of expression values.

Methods that operate on SCESet objects constitute the basic scater workflow.

Slots

`logExprsOffset`: Scalar of class `"numeric"`, providing an offset applied to expression data in the ‘exprs‘ slot when undergoing log2-transformation to avoid trying to take logs of zero.

`lowerDetectionLimit`: Scalar of class `"numeric"`, giving the lower limit for an expression value to be classified as "expressed".

`cellPairwiseDistances`: Matrix of class `"numeric"`, containing pairwise distances between cells.

`featurePairwiseDistances`: Matrix of class `"numeric"`, containing pairwise distances between features.

`reducedDimension`: Matrix of class `"numeric"`, containing reduced-dimension coordinates for cells (generated, for example, by PCA).

`bootstraps`: Array of class `"numeric"` that can contain bootstrap estimates of the expression or count values.

`sc3`: List containing results from consensus clustering from the SC3 package.

`featureControlInfo`: Data frame of class `"AnnotatedDataFrame"` that can contain information/metadata about sets of control features defined for the `SCESet` object. bootstrap estimates of the expression or count values.

References

Thanks to the Monocle package (github.com/cole-trapnell-lab/monocle-release/) for their CellDataSet class, which provided the inspiration and template for SCESet.

---

sc_example_cell_info — *Cell information for the small example single-cell counts dataset to demonstrate capabilities of scater*

Description

This data.frame contains cell metadata information for the 40 cells included in the example `counts` dataset included in the package.

Usage

```
sc_example_cell_info
```

Format

a data.frame instance, 1 row per cell.

Value

NULL, but makes aavailable a data frame with cell metadata

Author(s)

Davis McCarthy, 2015-03-05

Source

Wellcome Trust Centre for Human Genetics, Oxford

---

sc_example_counts — *A small example of single-cell counts dataset to demonstrate capabilities of scater*

Description

This data set contains counts for 2000 genes for 40 cells. They are from a real experiment, but details have been anonymised.

Usage

```
sc_example_counts
```

Format

a matrix instance, 1 row per gene.

Value

NULL, but makes aavailable a matrix of count data

Author(s)

Davis McCarthy, 2015-03-05

Source

Wellcome Trust Centre for Human Genetics, Oxford

`summariseExprsAcrossFeatures` *Summarise expression values across feature*

Description

Create a new `SingleCellExperiment` with counts summarised at a different feature level. A typical use would be to summarise transcript-level counts at gene level.

Usage

```
summariseExprsAcrossFeatures(object, exprs_values = "tpm",
  summarise_by = "feature_id", scaled_tpm_counts = TRUE, lib_size = NULL)
```

Arguments

`object`	an `SingleCellExperiment` object.
`exprs_values`	character string indicating which slot of the assayData from the `SingleCellExperiment` object should be used as expression values. Valid options are `'counts'` the counts slot, `'tpm'` the transcripts-per-million slot or `'fpkm'` the FPKM slot.
`summarise_by`	character string giving the column of `rowData(object)` that will be used as the features for which summarised expression levels are to be produced. Default is `'feature_id'`.
`scaled_tpm_counts`	logical, should feature-summarised counts be computed from summed TPM values scaled by total library size? This approach is recommended (see https://f1000research.com/articles/4-1521/v2), so the default is `TRUE` and it is applied if TPM values are available in the object.
`lib_size`	optional vector of numeric values of same length as the number of columns in the `SingleCellExperiment` object providing the total library size (e.g. "count of mapped reads") for each cell/sample.

Details

Only transcripts-per-million (TPM) and fragments per kilobase of exon per million reads mapped (FPKM) expression values should be aggregated across features. Since counts are not scaled by the length of the feature, expression in counts units are not comparable within a sample without adjusting for feature length. Thus, we cannot sum counts over a set of features to get the expression of that set (for example, we cannot sum counts over transcripts to get accurate expression estimates for a gene). See the following link for a discussion of RNA-seq expression units by Harold Pimentel: https://haroldpimentel.wordpress.com/2014/05/08/what-the-fpkm-a-review-rna-seq-expression-units For more details about the effects of summarising transcript expression values at the gene level see Sonesen et al, 2016 (https://f1000research.com/articles/4-1521/v2).

Value

an SingleCellExperiment object

Examples

```
data("sc_example_counts")
data("sc_example_cell_info")
example_sce <- SingleCellExperiment(
assays = list(counts = sc_example_counts), colData = sc_example_cell_info)
rd <- data.frame(gene_id = rownames(example_sce),
feature_id = paste("feature", rep(1:500, each = 4), sep = "_"))
rownames(rd) <- rownames(example_sce)
rowData(example_sce) <- rd
effective_length <- rep(c(1000, 2000), times = 1000)
tpm(example_sce) <- calculateTPM(example_sce, effective_length, calc_from = "counts")

example_sceset_summarised <-
summariseExprsAcrossFeatures(example_sce, exprs_values = "tpm")
example_sceset_summarised <-
summariseExprsAcrossFeatures(example_sce, exprs_values = "counts")
```

---

uniquifyFeatureNames *Make feature names unique*

Description

Combine a user-interpretable feature name (e.g., gene symbol) with a standard identifier that is guaranteed to be unique (e.g., Ensembl) for use as row names.

Usage

```
uniquifyFeatureNames(ID, names)
```

Arguments

ID	A character vector of unique identifiers.
names	A character vector of feature names.

Details

This function will attempt to use names if it is unique. If not, it will append the _ID to any non-unique value of names. Missing names will be replaced entirely by ID.

The output is guaranteed to be unique, assuming that ID is also unique. This can be directly used as the row names of a SingleCellExperiment object.

Value

A character vector of unique-ified feature names.

Author(s)

Aaron Lun

Examples

```
uniquifyFeatureNames(
  ID=paste0("ENSG0000000", 1:5),
  names=c("A", NA, "B", "C", "A")
)
```

---

updateSCESet — *Convert an SCESet object to a SingleCellExperiment object*

Description

Convert an SCESet object produced with an older version of the package to a SingleCellExperiment object compatible with the current version.

Usage

```
updateSCESet(object)

toSingleCellExperiment(object)
```

Arguments

object	an SCESet object to be updated

Value

a SingleCellExperiment object

Examples

```
## Not run:
updateSCESet(example_sceset)

## End(Not run)
## Not run:
toSingleCellExperiment(example_sceset)

## End(Not run)
```

Index