Package ‘scuttle’

May 25, 2024

Type    Package
Version  1.15.0
Date     2024-03-05
License  GPL-3
Title    Single-Cell RNA-Seq Analysis Utilities
Description Provides basic utility functions for performing single-cell analyses, focusing on simple normalization, quality control and data transformations. Also provides some helper functions to assist development of other packages.
Depends  SingleCellExperiment
Imports  methods, utils, stats, Matrix, Rcpp, BiocGenerics, S4Vectors, BiocParallel, GenomicRanges, SummarizedExperiment, DelayedArray, DelayedMatrixStats, beachmat
Suggests BiocStyle, knitr, scRNAseq, rmarkdown, testthat, scran
VignetteBuilder knitr
biocViews ImmunoOncology, SingleCell, RNASeq, QualityControl, Preprocessing, Normalization, Transcriptomics, GeneExpression, Sequencing, Software, DataImport
LinkingTo Rcpp, beachmat
SystemRequirements C++11
RoxygenNote 7.2.3
NeedsCompilation yes
git_url  https://git.bioconductor.org/packages/scuttle
git_branch devel
git_last_commit fd7d1ad
git_last_commit_date 2024-04-30
Repository Bioconductor 3.20
Date/Publication 2024-05-24
Author  Aaron Lun [aut, cre],
        Davis McCarthy [aut]
Maintainer  Aaron Lun <infinite.monkeys.with.keyboards@gmail.com>
## Contents

- addPerCellQCMetrics ........................................ 3
- aggregateAcrossCells ........................................ 4
- aggregateAcrossFeatures ................................. 7
- calculateAverage ........................................... 8
- calculateCPM ............................................... 10
- calculateFPKM ............................................. 12
- calculateTPM ............................................... 13
- cleanSizeFactors ........................................... 14
- computePooledFactors ..................................... 16
- computeSpikeFactors ....................................... 21
- correctGroupSummary ...................................... 22
- downsampleBatches ......................................... 24
- downsampleMatrix ........................................... 26
- fitLinearModel .............................................. 28
- geometricSizeFactors ...................................... 29
- isOutlier ..................................................... 31
- librarySizeFactors ......................................... 34
- logNormCounts ............................................... 35
- makePerCellDF .............................................. 38
- makePerFeatureDF ........................................... 41
- medianSizeFactors .......................................... 42
- mockSCE ..................................................... 44
- normalizeCounts ............................................ 45
- numDetectedAcrossCells ................................... 49
- numDetectedAcrossFeatures .............................. 51
- perCellQCFilters ............................................ 52
- perCellQCMetrics ........................................... 54
- perFeatureQCMetrics ....................................... 58
- quickPerCellQC .............................................. 60
- readSparseCounts ........................................... 62
- reexports .................................................... 64
- scuttle-pkg ................................................... 64
- scuttle-utils ................................................ 64
- sumCountsAcrossCells ..................................... 65
- sumCountsAcrossFeatures ................................. 67
- summarizeAssayByGroup ................................... 69
- uniquifyFeatureNames ..................................... 71

| Index | 73 |
addPerCellQCMetrics

---

**addPerCellQCMetrics**  
Add QC metrics to a SummarizedExperiment

---

### Description

Convenient utilities to compute QC metrics and add them to a `SummarizedExperiment`'s row or column metadata.

### Usage

```r
addPerCellQCMetrics(x, ...)  
addPerFeatureQCMetrics(x, ...)  
addPerCellQC(x, ...)  
addPerFeatureQC(x, ...)```

### Arguments

- **x**
  A `SummarizedExperiment` object or one of its subclasses.
- **...**
  For `addPerCellQCMetrics`, further arguments to pass to `perCellQCMetrics`.
  For `addPerFeatureQCMetrics`, further arguments to pass to `perFeatureQCMetrics`.

### Details

These functions are simply wrappers around `perCellQCMetrics` and `perFeatureQCMetrics`, respectively. The computed QC metrics are automatically appended onto the existing `colData` or `rowData`. No protection is provided against duplicated column names.

`addPerCellQC` and `addPerFeatureQC` are exactly the same functions, *sans* the `Metrics` at the end of their names. They were added in the tempestuous youth of this package when naming was fast and loose. These can be considered to be soft-deprecated in favor of the longer forms.

### Value

`x` is returned with the QC metrics added to the row or column metadata.

### Author(s)

Aaron Lun

### See Also

`perCellQCMetrics` and `perFeatureQCMetrics`, which do the actual work.
aggregateAcrossCells

Aggregate data across groups of cells

Example

```r
example_sce <- mockSCE()
example_sce <- addPerCellQCMetrics(example_sce)
colData(example_sce)

example_sce <- addPerFeatureQCMetrics(example_sce)
rowData(example_sce)
```

aggregateAcrossCells Aggregate data across groups of cells

Description

Sum counts or average expression values for each feature across groups of cells, while also aggregating values in the colData and other fields in a SummarizedExperiment.

Usage

```r
aggregateAcrossCells(x, ...)
```

## S4 method for signature 'SummarizedExperiment'
aggregateAcrossCells(
  x,
  ids,
  ..., 
  statistics = NULL,
  average = NULL,
  suffix = FALSE,
  subset.row = NULL,
  subset.col = NULL,
  store.number = "ncells",
  coldata.merge = NULL,
  use.assay.type = "counts",
  subset_row = NULL,
  subset_col = NULL,
  store_number = "ncells",
  coldata_merge = NULL,
  use_exprs_values = NULL
)

## S4 method for signature 'SingleCellExperiment'
aggregateAcrossCells(
  x,
  ids,
  ..., 
  subset.row = NULL,
  ...)
aggregateAcrossCells

subset.col = NULL,
use.altexps = FALSE,
use.dimred = TRUE,
dimred.stats = NULL,
suffix = FALSE,
subset.row = NULL,
subset.col = NULL,
use.altexps = NULL,
use_dimred = NULL
)

Arguments

x A SingleCellExperiment or SummarizedExperiment containing one or more matrices of expression values to be aggregated; possibly along with colData, reducedDims and altExps elements.

... For the generic, further arguments to be passed to specific methods.
For the SummarizedExperiment method, further arguments to be passed to summarizeAssayByGroup.
For the SingleCellExperiment method, arguments to be passed to the SummarizedExperiment method.

ids A factor (or vector coercible into a factor) specifying the group to which each cell in x belongs. Alternatively, a DataFrame of such vectors or factors, in which case each unique combination of levels defines a group.

statistics Character vector specifying the type of statistics to be computed, see ?summarizeAssayByGroup. If not specified, defaults to "sum".

average Deprecated, specifies whether to compute the average - use statistics="mean" instead. Only used if statistics=NULL.

suffix Logical scalar indicating whether to always suffix the assay name with the statistic type.

subset.row An integer, logical or character vector specifying the features to use. If NULL, defaults to all features.

subset.col An integer, logical or character vector specifying the cells to use. Defaults to all cells with non-NA entries of ids.

store.number String specifying the field of the output colData to store the number of cells in each group. If NULL, nothing is stored.

coldata.merge A named list of functions specifying how each column metadata field should be aggregated. Each function should be named according to the name of the column in colData to which it applies. Alternatively, a single function can be supplied, see below for more details.

use.assay.type A character or integer vector specifying the assay(s) of x containing count matrices.

subset.row, subset.col, store.number, use_exprs_values, use_altexps, use_dimred, coldata_merge Soft deprecated equivalents to the arguments described above.

use.altexps Deprecated, use applySCE instead.
use.dimred Logical scalar indicating whether aggregation should be performed for dimensionality reduction results. Alternatively, a character or integer vector specifying the dimensionality reduction results to be aggregated.

dimred.stats A character vector specifying how the reduced dimensions should be aggregated by group. This can be one or more of "mean" and "median".

Details

This function summarizes the assay values in x for each group in ids using `summarizeAssayByGroup` while also aggregating metadata across cells in a "sensible" manner. This makes it useful for obtaining an aggregated `SummarizedExperiment` during an analysis session; in contrast, `summarizeAssayByGroup` is more lightweight and is better for use inside other functions.

Aggregation of the `colData` is controlled using functions in `coldata.merge`. This can either be:

- A function that takes a subset of entries for any given column metadata field and returns a single value. This can be set to, e.g., `sum` or `median` for numeric covariates, or a function that takes the most abundant level for categorical factors.
- A named list of such functions, where each function is applied to the column metadata field after which it is named. Any field that does not have an entry in `coldata.merge` is "unspecified" and handled as described below. A list element can also be set to `FALSE`, in which case no aggregation is performed for the corresponding field.
- `NULL`, in which case all fields are considered to be unspecified.
- `FALSE`, in which case no aggregation of column metadata is performed.

For any unspecified field, we check if all cells of a group have the same value. If so, that value is reported, otherwise a `NA` is reported for the offending group.

By default, each matrix values is returned with the same name as the original per-cell matrix from which it was derived. If `statistics` is of length greater than 1 or `suffix=TRUE`, the names of all aggregated matrices are suffixed with their type of aggregate statistic.

If `ids` is a `DataFrame`, the combination of levels corresponding to each column is also reported in the column metadata. Otherwise, the level corresponding to each column is reported in the `ids` column metadata field as well as in the column names.

If x is a `SingleCellExperiment`, entries of `reducedDims` specified by `use.dimred` are averaged across cells. This assumes that the average of low-dimensional coordinates has some meaning for a group of cells but the sum does not. We can explicitly specify computation of the "mean" or "median" (or both) with `dimred.stats`. If `dimred.stats` is of length greater than 1 or `suffix=TRUE`, the name of each matrix in the output `reducedDims` is suffixed with the type of average.

If x is a `SingleCellExperiment`, any alternative Experiments are removed from the output object. Users should call `applySCE` to repeat the same aggregation on the alternative Experiments if this is desired - see Examples.

Value

A `SummarizedExperiment` of the same class of x is returned containing summed/averaged matrices generated by `summarizeAssayByGroup` on all assays in `use.assay.type`. Column metadata are also aggregated according to the rules in `coldata.merge`, see Details.
aggregateAcrossFeatures

Aggregate feature sets in a SummarizedExperiment

Description

Sum together expression values (by default, counts) for each feature set in each cell of a SummarizedExperiment object.

Usage

aggregateAcrossFeatures(
  x,
  ids,
  ...
)

Author(s)

Aaron Lun

See Also

summarizeAssayByGroup, which does the heavy lifting at the assay level.

Examples

element_sce <- mockSCE()
ids <- sample(LETTERS[1:5], ncol(element_sce), replace=TRUE)
out <- aggregateAcrossCells(element_sce, ids)
out

batches <- sample(1:3, ncol(element_sce), replace=TRUE)
out2 <- aggregateAcrossCells(element_sce,
  DataFrame(label=ids, batch=batches))
out2

# Using another column metadata merge strategy.
example_sce$stuff <- runif(ncol(example_sce))
out3 <- aggregateAcrossCells(element_sce, ids,
  coldata_merge=list(stuff=sum))

# Aggregating across the alternative Experiments as well.
out4 <- applySCE(element_sce, aggregateAcrossCells, ids=ids)
assay(altExp(out4))[1:10,]
calculateAverage

Arguments

- **x**
  
  A `SummarizedExperiment` containing an expression matrix.

- **ids**
  
  A factor of length `nrow(x)`, specifying the set to which each feature in `x` belongs. Alternatively, a list of integer or character vectors, where each vector specifies the indices or names of features in a set. Logical vectors are also supported.

- **use.assay.type**
  
  A character or integer vector specifying the assay(s) of `x` containing expression matrices.

- **use_exprs_values**
  
  Soft-deprecated equivalent of `use.assay.type`.

Value

A `SummarizedExperiment` of the same class as `x` is returned, containing summed matrices generated by `sumCountsAcrossFeatures` on all assays in `use.assay.type`.

If `ids` is a factor, row metadata is retained for the first instance of a feature from each set in `ids`. This behavior assumes that `ids` specifies duplicates of the same gene, such that the first instance is a reasonable choice.

If `ids` is a list, row metadata is simply discarded. This behavior assumes that `ids` specifies gene sets such that any existing gene-level metadata is meaningless.

Author(s)

Aaron Lun

See Also

`sumCountsAcrossFeatures`, which does the heavy lifting.

Examples

```r
example_sce <- mockSCE()
ids <- sample(LETTERS, nrow(example_sce), replace=TRUE)
aggr <- aggregateAcrossFeatures(example_sce, ids)
aggr
```

---

**calculateAverage**  
*Calculate per-feature average counts*

Description

Calculate the average count for each feature after normalizing observations using per-cell size factors.
Usage

calculateAverage(x, ...)

## S4 method for signature 'ANY'
calculateAverage(
  x,
  size.factors = NULL,
  subset.row = NULL,
  BPPARAM = SerialParam(),
  size_factors = NULL,
  subset_row = NULL
)

## S4 method for signature 'SummarizedExperiment'
calculateAverage(x, ..., assay.type = "counts", exprs_values = NULL)

## S4 method for signature 'SingleCellExperiment'
calculateAverage(x, size.factors = NULL, ...)

Arguments

x            A numeric matrix of counts where features are rows and columns are cells. Alternatively, a SummarizedExperiment or a SingleCellExperiment containing such counts.
...
size.factors A numeric vector containing size factors. If NULL, these are calculated or extracted from x.
subset.row   A vector specifying the subset of rows of object for which to return a result.
BPPARAM      A BiocParallelParam object specifying whether the calculations should be parallelized. Only relevant for parallelized rowSums(x), e.g., for DelayedMatrix inputs.
assay.type   A string specifying the assay of x containing the count matrix.

Details

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

If no size factors are supplied, they are determined automatically:
For count matrices and `SummarizedExperiment` inputs, the sum of counts for each cell is used to compute a size factor via the `librarySizeFactors` function.

For `SingleCellExperiment` instances, the function searches for `sizeFactors` from `x`. If none are available, it defaults to library size-derived size factors.

If `size_factors` are supplied, they will override any size factors present in `x`.

**Value**

A numeric vector of average count values with same length as number of features (or the number of features in `subset_row` if supplied).

**Author(s)**

Aaron Lun

**See Also**

`librarySizeFactors`, for the default calculation of size factors.

`logNormCounts`, for the calculation of normalized expression values.

**Examples**

```r
example_sce <- mockSCE()
ave_counts <- calculateAverage(example_sce)
summary(ave_counts)
```

---

**calculateCPM**

**Calculate CPMs**

**Description**

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

**Usage**

```r
calculateCPM(x, ...)
```

## S4 method for signature 'ANY'

```r
calculateCPM(
  x,
  size.factors = NULL,
  subset.row = NULL,
  size_factors = NULL,
  subset_row = NULL
)
```

## S4 method for signature 'SummarizedExperiment'
calculateCPM

calculateCPM(x, ..., assay.type = "counts", exprs_values = NULL)

## S4 method for signature 'SingleCellExperiment'
calculateCPM(x, size.factors = NULL, ...)

Arguments

x       A numeric matrix of counts where features are rows and cells are columns. Alternatively, a SummarizedExperiment or a SingleCellExperiment containing such counts.

...     For the generic, arguments to pass to specific methods. For the SummarizedExperiment method, further arguments to pass to the ANY method. For the SingleCellExperiment method, further arguments to pass to the SummarizedExperiment method.

size.factors A numeric vector containing size factors to adjust the library sizes. If NULL, the library sizes are used directly.

subset.row A vector specifying the subset of rows of x for which to return a result.

Details

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

Value

A numeric matrix of CPM values with the same dimensions as x (unless subset.row is specified).

Author(s)

Aaron Lun

See Also

normalizeCounts, on which this function is based.

Examples

```r
example_sce <- mockSCE()
cpm(example_sce) <- calculateCPM(example_sce)
str(cpm(example_sce))
```
calculateFPKM  \hspace{1cm} \textit{Calculate FPKMs}

\textbf{Description}

Calculate fragments per kilobase of exon per million reads mapped (FPKM) values from the feature-level counts.

\textbf{Usage}

\begin{verbatim}
calculateFPKM(x, lengths, ..., subset.row = NULL, subset_row = NULL)
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
\item \textbf{x} \hspace{1cm} A numeric matrix of counts where features are rows and cells are columns. Alternatively, a \texttt{SummarizedExperiment} or a \texttt{SingleCellExperiment} containing such counts.
\item \textbf{lengths} \hspace{1cm} Numeric vector providing the effective length for each feature in \texttt{x}.
\item \textbf{...} \hspace{1cm} Further arguments to pass to \texttt{calculateCPM}.
\item \textbf{subset.row} \hspace{1cm} A vector specifying the subset of rows of \texttt{x} for which to return a result.
\item \textbf{subset_row} \hspace{1cm} Soft-deprecated equivalent to the argument above.
\end{itemize}

\textbf{Details}

FPKMs are computed by dividing the CPMs by the effective length of each gene in kilobases. For RNA-seq datasets, the effective length is best set to the sum of lengths of all exons; for nucleus sequencing datasets, the effective length may instead be the entire width of the gene body.

\textbf{Value}

A numeric matrix of FPKM values with the same dimensions as \texttt{x} (unless \texttt{subset.row} is specified).

\textbf{Author(s)}

Aaron Lun, based on code by Davis McCarthy

\textbf{See Also}

\texttt{calculateCPM}, for the initial calculation of CPM values.

\textbf{Examples}

\begin{verbatim}
example_sce <- mockSCE()
eff_len <- runif(nrow(example_sce), 500, 2000)
fout <- calculateFPKM(example_sce, eff_len)
str(fout)
\end{verbatim}
**calculateTPM**  
*Calculate TPMs*

**Description**

Calculate transcripts-per-million (TPM) values for expression from feature-level counts.

**Usage**

```r
calculateTPM(x, ...)  
## S4 method for signature 'ANY'
calculateTPM(x, lengths = NULL, ...)

## S4 method for signature 'SummarizedExperiment'
calculateTPM(x, ..., assay.type = "counts", exprs_values = NULL)

## S4 method for signature 'SingleCellExperiment'
calculateTPM(x, lengths = NULL, size.factors = NULL, ...)
```

**Arguments**

- `x`  
  A numeric matrix of counts where features are rows and cells are columns. Alternatively, a `SummarizedExperiment` or a `SingleCellExperiment` containing such counts.

- `...`  
  For the generic, arguments to pass to specific methods.

  For the ANY method, further arguments to pass to `calculateCPM`.

  For the SummarizedExperiment method, further arguments to pass to the ANY method.

  For the SingleCellExperiment method, further arguments to pass to the SummarizedExperiment method.

- `lengths`  
  Numeric vector providing the effective length for each feature in `x`. Alternatively `NULL`, see Details.

- `assay.type`  
  A string specifying the assay of `x` containing the count matrix.

- `exprs_values`  
  Soft-deprecated equivalents to the arguments above.

- `size.factors`  
  A numeric vector containing size factors to adjust the library sizes. If `NULL`, the library sizes are used directly.

**Details**

For read count data, this function assumes uniform coverage along the (effective) length of the transcript. Thus, the number of transcripts for a gene is proportional to the read count divided by the transcript length. Here, the division is done before calculation of the library size to compute permillion values, where `calculateFPKM` will only divide by the length after library size normalization.

For UMI count data, this function should be run with `lengths=NULL`, i.e., no division by the effective length. This is because the number of UMIs is a direct (albeit biased) estimate of the number of transcripts.
**Value**

A numeric matrix of TPM values with the same dimensions as `x` (unless `subset.row` is specified).

**Author(s)**

Aaron Lun, based on code by Davis McCarthy

**See Also**

`calculateCPM`, on which this function is based.

**Examples**

```r
example_sce <- mockSCE()
eff_len <- runif(nrow(example_sce), 500, 2000)
tout <- calculateTPM(example_sce, lengths = eff_len)
str(tout)
```

---

**cleanSizeFactors**

Clean out non-positive size factors

**Description**

Coerce non-positive size factors (occasionally generated by `pooledSizeFactors`) to positive values based on the number of detected features.

**Usage**

```r
cleanSizeFactors(
  size.factors,
  num.detected,
  control = nls.control(warnOnly = TRUE),
  iterations = 3,
  nmads = 3,
  ...
)
```

**Arguments**

- `size.factors`: A numeric vector containing size factors for all libraries.
- `num.detected`: A numeric vector of the same length as `size.factors`, containing the number of features detected in each library.
- `control`: Argument passed to `nls` to control the fitting, see `?nls.control` for details.
- `iterations`: Integer scalar specifying the number of robustness iterations.
- `nmads`: Numeric scalar specifying the multiple of MADs to use for the tricube bandwidth in robustness iterations.
- `...`: Further arguments to pass to `nls`. 
Details

This function will first fit a non-linear curve of the form

\[ y = \frac{ax}{1 + bx} \]

where \( y \) is `num.detected` and \( x \) is `size.factors` for all positive size factors. This is a purely empirical expression, chosen because it is passes through the origin, is linear near zero and asymptotes at large \( x \). The fitting is done robustly with iterations of tricube weighting to eliminate outliers.

We then consider the number of detected features for all samples with non-positive size factors. This is treated as \( y \) and used to solve for \( x \) based on the curve fitted above. The result is the “cleaned” size factor, which must always be positive for \( y < a/b \). For \( y > a/b \), there is no solution so the cleaned size factor is defined as the largest positive value in `size.factors`.

Negative size factors can occasionally be generated by `pooledSizeFactors`, see the documentation there for more details. By coercing them to positive values, we can proceed to normalization and downstream analyses. Here, we use the number of detected features as this is more robust to differential expression that would cause biases in the library size. Of course, it is not theoretically guaranteed to yield the correct size factor, but a rough guess is better than a negative value.

Value

A numeric vector identical to `size.factors` but with all non-positive size factors replaced with fitted values from the curve.

Author(s)

Aaron Lun

See Also

`pooledSizeFactors`, which can occasionally generate negative size factors.

`nls`, which performs the curve fitting.

Examples

```r
set.seed(100)
counts <- matrix(rpois(20000, lambda=1), ncol=100)

library(scuttle)
sf <- librarySizeFactors(counts)
ngenues <- colSums(counts > 0)

# Adding negative size factor values to be cleaned.
out <- cleanSizeFactors(c(-1, -1, sf), c(100, 50, ngenes))
head(out)
```
computePooledFactors  

Normalization by deconvolution

Description
Scaling normalization of single-cell RNA-seq data by deconvolving size factors from cell pools.

Usage

pooledSizeFactors(x, ...)

## S4 method for signature 'ANY'
pooledSizeFactors(
  x,
  sizes = seq(21, 101, 5),
  clusters = NULL,
  ref.clust = NULL,
  max.cluster.size = 3000,
  positive = TRUE,
  scaling = NULL,
  min.mean = NULL,
  subset.row = NULL,
  BPPARAM = SerializeParam()
)

## S4 method for signature 'SummarizedExperiment'
pooledSizeFactors(x, ..., assay.type = "counts")

computePooledFactors(x, ..., assay.type = "counts")

Arguments

x
For pooledSizeFactors, a numeric matrix-like object of counts, where rows are genes and columns are cells. Alternatively, a SummarizedExperiment object containing such a matrix.
For computePooledFactors, a SingleCellExperiment object containing a count matrix.

... For the pooledSizeFactors generic, additional arguments to pass to each method.
For the SummarizedExperiment method, additional methods to pass to the ANY method.
For the computePooledFactors function, additional arguments to pass to pooledSizeFactors.

sizes A numeric vector of pool sizes, i.e., number of cells per pool.

clusters An optional factor specifying which cells belong to which cluster, for deconvolution within clusters.

ref.clust A level of clusters to be used as the reference cluster for inter-cluster normalization.
computePooledFactors

max.cluster.size
An integer scalar specifying the maximum number of cells in each cluster.

positive
A logical scalar indicating whether linear inverse models should be used to enforce positive estimates.

scaling
A numeric scalar containing scaling factors to adjust the counts prior to computing size factors.

min.mean
A numeric scalar specifying the minimum (library size-adjusted) average count of genes to be used for normalization.

subset.row
An integer, logical or character vector specifying the features to use.

BPPARAM
A BiocParallelParam object specifying whether and how clusters should be processed in parallel.

assay.type
A string specifying which assay values to use when x is a SummarizedExperiment or SingleCellExperiment.

Value
For pooledSizeFactors, a numeric vector of size factors for all cells in x is returned.
For computePooledFactors, an object of class x is returned containing the vector of size factors in sizeFactors(x).

Overview of the deconvolution method

The pooledSizeFactors function implements the deconvolution strategy of Lun et al. (2016) for scaling normalization of sparse count data. Briefly, a pool of cells is selected and the expression profiles for those cells are summed together. The pooled expression profile is normalized against an average reference pseudo-cell, constructed by averaging the counts across all cells. This defines a size factor for the pool as the median ratio between the count sums and the average across all genes. The scaling bias for the pool is equal to the sum of the biases for the constituent cells. The same applies for the size factors, as these are effectively estimates of the bias for each cell. This means that the size factor for the pool can be written as a linear equation of the size factors for the cells. Repeating this process for multiple pools will yield a linear system that can be solved to obtain the size factors for the individual cells.

In this manner, pool-based factors are deconvolved to yield the relevant cell-based factors. The advantage is that the pool-based estimates are more accurate, as summation reduces the number of stochastic zeroes and the associated bias of the size factor estimate. This accuracy feeds back into the deconvolution process, thus improving the accuracy of the cell-based size factors.

Pooling with a sliding window

Within each cluster (if not specified, all cells are put into a single cluster), cells are sorted by increasing library size and a sliding window is applied to this ordering. Each location of the window defines a pool of cells with similar library sizes. This avoids inflated estimation errors for very small cells when they are pooled with very large cells. Sliding the window will construct an overdetermined linear system that can be solved by least-squares methods to obtain cell-specific size factors.

Window sliding is repeated with different window sizes to construct the linear system, as specified by sizes. By default, the number of cells in each window ranges from 21 to 101. Using a range of
window sizes improves the precision of the estimates, at the cost of increased computational work. The defaults were chosen to provide a reasonable compromise between these two considerations. The default set of sizes also avoids rare cases of linear dependencies and unstable estimates when all pool sizes are not co-prime with the number of cells.

The smallest window should be large enough so that the pool-based size factors are, on average, non-zero. We recommend window sizes no lower than 20 for UMI data, though smaller windows may be possible for read count data. The total number of cells should also be at least 100 for effective pooling. (If cluster is specified, we would want at least 100 cells per cluster.)

If there are fewer cells than the smallest window size, the function will naturally degrade to performing library size normalization. This yields results that are the same as `librarySizeFactors`.

**Prescaling of the counts**

The simplest approach to pooling is to simply add the counts together for all cells in each pool. However, this is suboptimal as any errors in the estimation of the pooled size factor will propagate to all component cell-specific size factors upon solving the linear system. If the error is distributed evenly across all cell-specific size factors, the small size factors will have larger relative errors compared to the large size factors.

To avoid this, we perform “prescaling” where we divide the counts by a cell-specific factor prior to pooling. Ideally, the prescaling factor should be close to the true size factor for each cell. Solving the linear system constructed with prescaled values should yield estimates that are more-or-less equal across all cells. Thus, given similar absolute errors, the relative errors for all cells will also be similar.

Obviously, the true size factor is unknown (otherwise why bother running this function?) so we use the library size for each cell as a proxy instead. This may perform poorly in pathological scenarios involving extreme differential expression and strong composition biases. In cases where a more appropriate initial estimate is available, this can be used as the prescaling factor by setting the scaling argument.

One potential approach is to run `computePooledFactors` twice to improve accuracy. The first run is done as usual and will yield an initial estimate of the size factor for each cell. In the second run, we supply our initial estimates in the scaling argument to serve as better prescaling factors. Obviously, this involves twice as much computational work so we would only recommend attempting this in extreme circumstances.

**Solving the linear system**

The linear system is solved using the sparse QR decomposition from the `Matrix` package. However, this has known problems when the linear system becomes too large (see [https://stat.ethz.ch/pipermail/r-help/2011-August/285855.html](https://stat.ethz.ch/pipermail/r-help/2011-August/285855.html)). In such cases, we set clusters to break up the linear system into smaller, more manageable components that can be solved separately. The default `max.cluster.size` will arbitrarily break up the cell population (within each cluster, if specified) so that we never pool more than 3000 cells. Note that this involves appending a suffix like "-1" to the end of each cluster’s name; this may appear on occasion in warnings or error messages.

**Normalization within and between clusters**

In general, it is more appropriate to pool more similar cells to avoid violating the assumption of a non-DE majority of genes. This can be done by specifying the clusters argument where cells in
each cluster have similar expression profiles. Deconvolution is subsequently applied on the cells within each cluster, where there should be fewer DE genes between cells. Any clustering can be used, and only a rough clustering is required; computePooledFactors is robust to a moderate level of DE within each cluster. The quickCluster function from the scran package is particularly convenient for this purpose.

Size factors computed within each cluster must be rescaled for comparison between clusters. To do so, we choose one cluster as a “reference” to which all others are normalized. Ideally, the reference cluster should have a stable expression profile and not be extremely different from all other clusters. The assumption here is that there is a non-DE majority between the reference and each other cluster (which is still a weaker assumption than that required without clustering). The rescaling factor is then defined by computing the ratios in averaged expression between each cluster’s pseudo-cell and that of the reference, and taking the median of these ratios across all genes.

By default, the cluster with the most non-zero counts is used as the reference. This reduces the risk of obtaining undefined rescaling factors for the other clusters, while improving the precision (and also accuracy) of the median-based factor estimate. Alternatively, the reference can be manually specified using ref.clust if there is prior knowledge about which cluster is most suitable, e.g., from PCA or t-SNE plots.

Each cluster should ideally be large enough to contain a sufficient number of cells for pooling. Otherwise, computePooledFactors will fall back to library size normalization for small clusters.

If the estimated rescaling factor is not positive, a warning is emitted and the function falls back to the ratio of sums between pseudo-cells (in effect, library size normalization). This can occasionally happen when a cluster’s cells expresses a small subset of genes - this is not problematic for within-cluster normalization, as non-expressed genes are simply ignored, but violates the assumption of a non-DE majority when performing inter-cluster comparisons.

Dealing with non-positive size factors

It is possible for the deconvolution algorithm to yield negative or zero estimates for the size factors. These values are obviously nonsensical and computePooledFactors will raise a warning if they are encountered. Negative estimates are mostly commonly generated from low quality cells with few expressed features, such that most genes still have zero counts even after pooling. They may also occur if insufficient filtering of low-abundance genes was performed.

To avoid these problematic size factors, the best solution is to increase the stringency of the filtering.

• If only a few negative/zero size factors are present, they are likely to correspond to a few low-quality cells with few expressed features. Such cells are difficult to normalize reliably under any approach, and can be removed by increasing the stringency of the quality control.

• If many negative/zero size factors are present, it is probably due to insufficient filtering of low-abundance genes. This results in many zero counts and pooled size factors of zero, and can be fixed by filtering out more genes with a higher min.mean - see “Gene selection” below.

Another approach is to increase in the number of sizes to improve the precision of the estimates. This reduces the chance of obtaining negative/zero size factors due to estimation error, for cells where the true size factors are very small.

As a last resort, positive=TRUE is set by default, which uses cleanSizeFactors to coerce any non-positive estimates to positive values. This ensures that, at the very least, downstream analysis is possible even if the size factors for affected cells are not accurate. Users can skip this step by setting positive=FALSE to perform their own diagnostics or coercions.
Gene selection

If too many genes have consistently low counts across all cells, even the pool-based size factors will be zero. This results in zero or negative size factor estimates for many cells. We avoid this by filtering out low-abundance genes using the min.mean argument. This represents a minimum threshold min.mean on the library size-adjusted average counts from calculateAverage.

By default, we set min.mean to 1 for read count data and 0.1 for UMI data. The exact values of these defaults are more-or-less arbitrary and are retained for historical reasons. The lower threshold for UMIs is motivated by (i) their lower count sizes, which would result in the removal of too many genes with a higher threshold; and (ii) the lower variability of UMI counts, which results in a lower frequency of zeroes compared to read count data at the same mean. We use the median library size to detect whether the counts are those of reads (above 100,000) or UMIs (below 50,000) to automatically set min.mean. Mean library sizes in between these two limits will trigger a warning and revert to using min.mean=0.1.

If clusters is specified, filtering by min.mean is performed on the per-cluster average during within-cluster normalization, and then on the (library size-adjusted) average of the per-cluster averages during between-cluster normalization.

Performance can generally be improved by removing genes that are known to be strongly DE between cells. This weakens the assumption of a non-DE majority and avoids the error associated with DE genes. For example, we might remove viral genes when our population contains both infected and non-infected cells. Of course, computePooledFactors is robust to some level of DE genes - that is, after all, its raison d’etre - so one should only explicitly remove DE genes if it is convenient to do so.

Obtaining standard errors

Previous versions of computePooledFactors would return the standard error for each size factor when errors=TRUE. This argument is no longer available as we have realized that standard error estimation from the linear model is not reliable. Errors are likely underestimated due to correlations between pool-based size factors when they are computed from a shared set of underlying counts. Users wishing to obtain a measure of uncertainty are advised to perform simulations instead, using the original size factor estimates to scale the mean counts for each cell. Standard errors can then be calculated as the standard deviation of the size factor estimates across simulation iterations.

Author(s)

Aaron Lun and Karsten Bach

References

Lun ATL, Bach K and Marioni JC (2016). Pooling across cells to normalize single-cell RNA sequencing data with many zero counts. Genome Biol. 17:75

See Also

logNormCounts, which uses the computed size factors to compute normalized expression values. librarySizeFactors and medianSizeFactors, for simpler approaches to computing size factors. quickCluster from the scran package, to obtain a rough clustering for use in clusters.
computeSpikeFactors

Examples

```r
library(scuttle)
sce <- mockSCE(ncells=500)

# Computing the size factors.
sce <- computePooledFactors(sce)
head(sizeFactors(sce))
plot(librarySizeFactors(sce), sizeFactors(sce), log="xy")

# Using pre-clustering.
library(scran)
preclusters <- quickCluster(sce)
table(preclusters)
sce2 <- computePooledFactors(sce, clusters=preclusters)
head(sizeFactors(sce2))
```

computeSpikeFactors  Normalization with spike-in counts

Description

Compute size factors based on the coverage of spike-in transcripts.

Usage

```r
computeSpikeFactors(x, spikes, assay.type = "counts")
```

Arguments

- `x`: A `SingleCellExperiment` object containing spike-in transcripts in an `altExps` entry.
- `spikes`: Support for spike-in transcripts in the rows of `x` itself is deprecated.
- `assay.type`: String or integer scalar specifying the alternative experiment containing the spike-in transcripts.

Details

The spike-in size factor for each cell is computed from the sum of all spike-in counts in each cell. This aims to scale the counts to equalize spike-in coverage between cells, thus removing differences in coverage due to technical effects like capture or amplification efficiency.

Spike-in normalization can be helpful for preserving changes in total RNA content between cells, if this is of interest. Such changes would otherwise be lost when normalizing with methods that assume a non-DE majority. Indeed, spike-in normalization is the only available approach if a majority of genes are DE between two cell types or states.
Size factors are computed by applying `librarySizeFactors` to the spike-in count matrix. This ensures that the mean of all size factors is unity for standardization purposes, if one were to compare expression values normalized with sets of size factors (e.g., in `modelGeneVarWithSpikes`). Users who want the spike-in size factors without returning a SingleCellExperiment object can simply call `librarySizeFactors(altExp(x, spikes))`, which gives the same result.

Value

A modified x is returned, containing spike-in-derived size factors for all cells in `sizeFactors`.

Author(s)

Aaron Lun

References


See Also

`altExps`, for the concept of alternative experiments.

`librarySizeFactors`, for how size factors are derived from library sizes.

Examples

```r
correctGroupSummary(  
  x,  
  group,  
  block,  
  transform = c("raw", "log", "logit"),  
  offset = NULL,  
)```
correctGroupSummary 23

weights = NULL,
subset.row = NULL
)

Arguments

x A numeric matrix containing summary statistics for each gene (row) and combination of group and block (column), computed by functions such as summarizeAssayByGroup - see Examples.
group A factor or vector specifying the group identity for each column of x, usually clusters or cell types.
block A factor or vector specifying the blocking level for each column of x, e.g., batch of origin.
transform String indicating how the differences between groups should be computed, for the batch adjustment.
offset Numeric scalar specifying the offset to use when difference="log" (default 1) or difference="logit" (default 0.01).
weights A numeric vector containing the weight of each combination, e.g., due to differences in the number of cells used to compute each summary. If NULL, all combinations have equal weight.
subset.row Logical, integer or character vector specifying the rows in x to use to compute statistics.

Details

Here, we consider group-level summary statistics such as the average expression of all cells or the proportion with detectable expression. These are easy to interpret and helpful for any visualizations that operate on individual groups, e.g., heatmaps.

However, in the presence of unwanted factors of variation (e.g., batch effects), some adjustment is required to ensure these group-level statistics are comparable. We cannot directly average group-level statistics across batches as some groups may not exist in particular batches, e.g., due to the presence of unique cell types in different samples. A direct average would be biased by variable contributions of the batch effect for each group.

To overcome this, we use groups that are present across multiple levels of the unwanted factor in multiple batches to correct for the batch effect. (That is, any level of groups that occurs for multiple levels of block.) For each gene, we fit a linear model to the (transformed) values containing both the group and block factors. We then report the coefficient for each group as the batch-adjusted average for that group; this is possible as the fitted model has no intercept.

The default of transform="raw" will not transform the values, and is generally suitable for log-expression values. Setting transform="log" will perform a log-transformation after adding offset (default of 1), and is suitable for normalized counts. Setting transform="logit" will perform a logit transformation after adding offset (default of 0.01) - to the numerator and twice to the denominator, to shrink to 0.5 - and is suitable for proportional data such as the proportion of detected cells.

After the model is fitted to the transformed values, the reverse transformation is applied to the coefficients to obtain a corrected summary statistic on the original scale. For transform="log", 


any negative values are coerced to zero, while for \texttt{transform=\textquotedblleft logit\textquotedblright}, any values outside of [0, 1] are coerced to the closest boundary.

**Value**

A numeric matrix with number of columns equal to the number of unique levels in \texttt{group}. Each column corresponds to a group and contains the averaged statistic across batches. Each row corresponds to a gene in \texttt{x} (or that specified by \texttt{subset.row} if not \texttt{NULL}).

**Author(s)**

Aaron Lun

**See Also**

\texttt{summarizeAssayByGroup}, to generate the group-level summaries for this function.

\texttt{regressBatches} from the \texttt{batchelor} package, to remove the batch effect from per-cell expression values.

**Examples**

```r
y <- matrix(rnorm(10000), ncol=1000)
group <- sample(10, ncol(y), replace=TRUE)
block <- sample(5, ncol(y), replace=TRUE)

summaries <- summarizeAssayByGroup(y, DataFrame(group=group, block=block),
  statistics=c("mean", "prop.detected"))

# Computing batch-aware averages:
averaged <- correctGroupSummary(assay(summaries, "mean"),
  group=summaries$group, block=summaries$block)

num <- correctGroupSummary(assay(summaries, "prop.detected"),
  group=summaries$group, block=summaries$block, transform="logit")
```

downsampleBatches  

**Description**

A convenience function to downsample all batches so that the average per-cell total count is the same across batches. This mimics the downsampling functionality of \texttt{cellranger aggr}.
Usage

downsampleBatches(
  ..., 
  batch = NULL, 
  block = NULL, 
  method = c("median", "mean", "geomean"), 
  bycol = TRUE, 
  assay.type = 1 
)

Arguments

... Two or more count matrices, where each matrix has the same set of genes (rows) and contains cells (columns) from a separate batch. Alternatively, one or more entries may be a SummarizedExperiment, in which case the count matrix is extracted from the assays according to assay.type. A list containing two or more of these matrices or SummarizedExperiments can also be supplied. Alternatively, a single count matrix or SummarizedExperiment can be supplied. This is assumed to contain cells from all batches, in which case batch should also be specified.

batch A factor of length equal to the number of columns in the sole entry of ..., specifying the batch of origin for each column of the matrix. Ignored if there are multiple entries in ...

block If ... contains multiple matrices or SummarizedExperiments, this should be a character vector of length equal to the number of objects in ..., specifying the blocking level for each object (see Details). Alternatively, if ... contains a single object, this should be a character vector or factor of length specifying the blocking level for each cell in that object.

method String indicating how the average total should be computed. The geometric mean is computed with a pseudo-count of 1.

bycol A logical scalar indicating whether downsampling should be performed on a column-by-column basis, see ?downsampleMatrix for more details.

assay.type String or integer scalar specifying the assay of the SummarizedExperiment containing the count matrix, if any SummarizedExperiments are present in ...

Details

Downsampling batches with strong differences in sequencing coverage can make it easier to compare them to each other, reducing the burden on the normalization and batch correction steps. This is especially true when the number of cells cannot be easily controlled across batches, resulting in large differences in per-cell coverage even when the total sequencing depth is the same.

Generally speaking, the matrices in ... should be filtered so that only libraries with cells are present. This is most relevant to high-throughput scRNA-seq experiments (e.g., using droplet-based protocols) where the majority of libraries do not actually contain cells. If these are not filtered
out, downsampling will equalize coverage among the majority of empty libraries rather than among cell-containing libraries.

In more complex experiments, batches can be organized into blocks where downsampling is performed to the lowest-coverage batch within each block. This is most useful for larger datasets involving technical replicates for the same biological sample. By setting `block=` to the biological sample, we can equalize coverage across replicates within each sample without forcing all samples to have the same coverage (e.g., to avoid loss of information if they are to be analyzed separately anyway).

**Value**

If ... contains two or more matrices, a List of downsampled matrices is returned. Otherwise, if ... contains only one matrix, the downsampled matrix is returned directly.

**Author(s)**

Aaron Lun

**See Also**

downsampleMatrix, which is called by this function under the hood.

**Examples**

```r
sce1 <- mockSCE()
sce2 <- mockSCE()

# Downsampling for multiple batches in a single matrix:
combined <- cbind(sce1, sce2)
batches <- rep(1:2, c(ncol(sce1), ncol(sce2)))
downscaled <- downsampleBatches(counts(combined), batch=batches)
downscaled[1:10,1:10]

# Downsampling for multiple matrices:
downscaled2 <- downsampleBatches(counts(sce1), counts(sce2))
downscaled2
```

---

### downsampleMatrix

**Downsample a count matrix**

**Description**

Downsample a count matrix to a desired proportion, either on a whole-matrix or per-cell basis.

**Usage**

```r
downsampleMatrix(x, prop, bycol = TRUE, sink = NULL)
```
**downsampleMatrix**

**Arguments**

- **x**: An integer or numeric matrix-like object containing counts.
- **prop**: A numeric scalar or, if bycol=TRUE, a vector of length ncol(x). All values should lie in [0, 1] specifying the downsampling proportion for the matrix or for each cell.
- **bycol**: A logical scalar indicating whether downsampling should be performed on a column-by-column basis.
- **sink**: A RealizationSink object specifying the format of the downsampled matrix should be returned.

**Details**

Given multiple batches of very different sequencing depths, it can be beneficial to downsample the deepest batches to match the coverage of the shallowest batches. This avoids differences in technical noise that can drive clustering by batch.

If bycol=TRUE, sampling without replacement is performed on the count vector for each cell. This yields a new count vector where the total is equal to prop times the original total count. Each count in the returned matrix is guaranteed to be smaller than the original value in x. Different proportions can be specified for different cells by setting prop to a vector; in this manner, downsampling can be used as an alternative to scaling for per-cell normalization.

If bycol=FALSE, downsampling without replacement is performed on the entire matrix. This yields a new matrix where the total count across all cells is equal to prop times the original total. The new total count for each cell may not be exactly equal to prop times the original value, which may or may not be more appropriate than bycol=TRUE for particular applications.

By default, the output format will be a dense matrix for an ordinary matrix x, and a sparse matrix for any sparse format (i.e., if is_sparse(x) returns TRUE). This can be overridden to specify custom formats with sink, e.g., for HDF5-backed matrices, which may be helpful for dealing with large matrices where the downsampled result may not fit in memory.

Note that this function was originally implemented in the scater package as downsampleCounts, was moved to the DropletUtils package as downsampleMatrix, and finally found a home here.

**Value**

An numeric matrix-like object of downsampled counts. This is a dgCMatrix unless sink is set, in which case it is a DelayedMatrix.

**Author(s)**

Aaron Lun

**See Also**

downsampleReads in the DropletUtils package, which downsamples reads rather than observed counts.

normalizeCounts, where downsampling can be used as an alternative to scaling normalization.
Examples

```r
sce <- mockSCE()
sum(counts(sce))

downsampled <- downsampleMatrix(counts(sce), prop = 0.5, bycol=FALSE)
sum(downsampled)

downsampled2 <- downsampleMatrix(counts(sce), prop = 0.5, bycol=TRUE)
sum(downsampled2)
```

---

**fitLinearModel**

*Fit a linear model*

---

**Description**

No-frills fitting of a linear model to the rows of any matrix-like object with numeric values.

**Usage**

```r
fitLinearModel(
  x,
  design,
  get.coefs = TRUE,
  subset.row = NULL,
  BPPARAM = SerialParam(),
  rank.error = TRUE
)
```

**Arguments**

- **x**
  - A numeric matrix-like object where columns are samples (e.g., cells) and rows are usually features (e.g., genes).
- **design**
  - A numeric design matrix with number of rows equal to `ncol(x)`. This should be of full column rank.
- **get.coefs**
  - A logical scalar indicating whether the coefficients should be returned.
- **subset.row**
  - An integer, character or logical vector indicating the rows of `x` to use for model fitting.
- **BPPARAM**
  - A `BiocParallelParam` object specifying the parallelization backend to use.
- **rank.error**
  - Logical scalar indicating whether to throw an error when `design` is not of full rank.
geometricSizeFactors

Details

This function is basically a stripped-down version of \texttt{lm.fit}, made to operate on any matrix representation (ordinary, sparse, whatever). It is generally intended for use inside other functions that require robust and efficient linear model fitting.

If design is not of full rank and \texttt{rank.error=TRUE}, an error is raised. If \texttt{rank.error=FALSE}, NA values are returned for all entries of the output list.

Value

If \texttt{get.coefs=TRUE}, a list is returned containing:

- \texttt{coefficients}, a numeric matrix of coefficient estimates, with one row per row of \texttt{x} (or a subset thereof specified by \texttt{subset.row}) and one column per column of \texttt{design}.
- \texttt{mean}, a numeric vector of row means of \texttt{x}. Computed as a courtesy to avoid iterating over the matrix twice.
- \texttt{variance}, a numeric vector of residual variances per row of \texttt{x}. Computed by summing the residual effects from the fitted model.
- \texttt{residual.df}, an integer scalar containing the residual degrees of freedom for \texttt{design}.

Otherwise, if \texttt{get.coefs=FALSE}, the same list is returned without \texttt{coefficients}.

Author(s)

Aaron Lun

Examples

```r
y <- Matrix::rsparsematrix(1000, 1000, 0.1)
design <- model.matrix(~runif(1000))
output <- fitLinearModel(y, design)
head(output$coefficients)
head(output$variance)
```

geometricSizeFactors  Compute geometric size factors

Description

Define per-cell size factors from the geometric mean of counts per cell.
Usage

geometricSizeFactors(x, ...)

## S4 method for signature 'ANY'
geometricSizeFactors(
  x,
  subset.row = NULL,
  pseudo.count = 1,
  BPPARAM = SerialParam()
)

## S4 method for signature 'SummarizedExperiment'
geometricSizeFactors(x, ..., assay.type = "counts")

calculateGeometricFactors(x, ...)

Arguments

x For geometricSizeFactors, a numeric matrix of counts with one row per feature and column per cell. Alternatively, a SummarizedExperiment or SingleCellExperiment containing such counts.

For computeGeometricFactors, only a SingleCellExperiment containing a count matrix is accepted.

... For the geometricSizeFactors generic, arguments to pass to specific methods.

For the SummarizedExperiment method, further arguments to pass to the ANY method.

For computeGeometricFactors, further arguments to pass to geometricSizeFactors.

subset.row A vector specifying whether the size factors should be computed from a subset of rows of x.

pseudo.count Numeric scalar specifying the pseudo-count to add during log-transformation.

BPPARAM A BiocParallelParam object indicating how calculations are to be parallelized. Only relevant when x is a DelayedArray object.

assay.type String or integer scalar indicating the assay of x containing the counts.

Details

The geometric mean provides an alternative measure of the average coverage per cell, in contrast to the library size factors (i.e., the arithmetic mean) computed by librarySizeFactors. The main advantage of the geometric mean is that it is more robust to large outliers, due to the slowly increasing nature of the log-transform at large values; in the normalization context, this translates to greater resistance to coposition biases from a few strongly upregulated genes.

On the other hand, the geometric mean is a poor estimator of the relative bias at low or zero counts. This is because the scaling of the coverage applies to the expectation of the raw counts, so the geometric mean only becomes an accurate estimator if the mean of the logs approaches the log of the mean (usually at high counts). The arbitrary pseudo-count also has a bigger influence at low counts.
As such, the geometric mean is only well-suited for deeply sequenced features, e.g., antibody-derived tags.

**Value**

For `geometricSizeFactors`, a numeric vector of size factors is returned for all methods. For `computeGeometricFactors`, `x` is returned containing the size factors in `sizeFactors(x)`.

**Author(s)**

Aaron Lun

**See Also**

`normalizeCounts` and `logNormCounts`, where these size factors are used by default. `geometricSizeFactors` and `medianSizeFactors`, for two other simple methods of computing size factors.

**Examples**

```r
example_sce <- mockSCE()
summary(geometricSizeFactors(example_sce))
```

---

**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**

```r
isOutlier(
  metric,
  nmads = 3,
  type = c("both", "lower", "higher"),
  log = FALSE,
  subset = NULL,
  batch = NULL,
  share.medians = FALSE,
  share.mads = FALSE,
  share.missing = TRUE,
  min.diff = NA,
  share.medians = NULL,
  share.mads = NULL,
  share.missing = NULL,
  min_diff = NULL
)
```
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 log2 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.
- **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.
- **share.medians**: Logical scalar indicating whether the median calculation should be shared across batches. Only used if batch is specified.
- **share.mads**: Logical scalar indicating whether the MAD calculation should be shared across batches. Only used if batch is specified.
- **share.missing**: Logical scalar indicating whether a common MAD/median should be used for any batch that has no values left after subsetting. Only relevant when both batch and subset are specified.
- **min.diff**: A numeric scalar indicating the minimum difference from the median to consider as an outlier. Ignored if NA.

**Details**

Lower and upper thresholds are stored in the "thresholds" attribute of the returned vector. By default, this is a numeric vector of length 2 for the threshold on each side. If type="lower", the higher limit is Inf, while if type="higher", the lower limit is -Inf.

If min.diff is not NA, the minimum distance from the median required to define an outlier is set as the larger of nmads MADs and min.diff. This aims to avoid calling many outliers when the MAD is very small, e.g., due to discreteness of the metric. If log=TRUE, this difference is defined on the log2 scale.

If subset is specified, the median and MAD are computed from a subset of cells and the values are used to define the outlier threshold that is applied to all cells. In a quality control context, this can be handy for excluding groups of cells that are known to be low quality (e.g., failed plates) so that they do not distort the outlier definitions for the rest of the dataset.

Missing values trigger a warning and are automatically ignored during estimation of the median and MAD. The corresponding entries of the output vector are also set to NA values.

The outlier.filter class is derived from an ordinary logical vector. The only difference is that any subsetting will not discard the "thresholds", which avoids unnecessary loss of information. Users can simply call as.logical to convert this into a logical vector.
Value

An outlier.filter object of the same length as the metric argument. This is effectively a logical vector specifying the observations that are considered as outliers. The chosen thresholds are stored in the "thresholds" attribute.

Handling batches

If batch is specified, outliers are defined within each batch separately using batch-specific median and MAD values. This gives the same results as if the input metrics were subsetted by batch and isOutlier was run on each subset, and is often useful when batches are known a priori to have technical differences (e.g., in sequencing depth).

If share.medians=TRUE, a shared median is computed across all cells. If share.mads=TRUE, a shared MAD is computed using all cells (based on either a batch-specific or shared median, depending on share.medians). These settings are useful to enforce a common location or spread across batches, e.g., we might set share.mads=TRUE for log-library sizes if coverage varies across batches but the variance across cells is expected to be consistent across batches.

If a batch does not have sufficient cells to compute the median or MAD (e.g., after applying subset), the default setting of share.missing=TRUE will set these values to the shared median and MAD. This allows us to define thresholds for low-quality batches based on information in the rest of the dataset. (Note that the use of shared values only affects this batch and not others unless share.medians and share.mads are also set.) Otherwise, if share.missing=FALSE, all cells in that batch will have NA in the output.

If batch is specified, the "threshold" attribute in the returned vector is a matrix with one named column per level of batch and two rows (one per threshold).

Author(s)

Aaron Lun

See Also

quickPerCellQC, a convenience wrapper to perform outlier-based quality control.
perCellQCMetrics, to compute potential QC metrics.

Examples

example_sce <- mockSCE()
stats <- perCellQCMetrics(example_sce)

str(isOutlier(stats$sum))
str(isOutlier(stats$sum, type="lower"))
str(isOutlier(stats$sum, type="higher"))

str(isOutlier(stats$sum, log=TRUE))

b <- sample(LETTERS[1:3], ncol(example_sce), replace=TRUE)
str(isOutlier(stats$sum, log=TRUE, batch=b))
librarySizeFactors  Compute library size factors

Description
Define per-cell size factors from the library sizes (i.e., total sum of counts per cell).

Usage
librarySizeFactors(x, ...)

## S4 method for signature 'ANY'
librarySizeFactors(
  x,
  subset.row = NULL,
  geometric = FALSE,
  BPPARAM = SerialParam(),
  subset_row = NULL,
  pseudo_count = 1
)

## S4 method for signature 'SummarizedExperiment'
librarySizeFactors(x, ..., assay.type = "counts", exprs_values = NULL)

computeLibraryFactors(x, ...)

Arguments
x  For librarySizeFactors, a numeric matrix of counts with one row per feature and column per cell. Alternatively, a SummarizedExperiment or SingleCellExperiment containing such counts.
For computeLibraryFactors, only a SingleCellExperiment containing a count matrix is accepted.

... For the librarySizeFactors generic, arguments to pass to specific methods.
For the SummarizedExperiment method, further arguments to pass to the ANY method.
For computeLibraryFactors, further arguments to pass to librarySizeFactors.

subset.row  A vector specifying whether the size factors should be computed from a subset of rows of x.

geometric  Deprecated, logical scalar indicating whether the size factor should be defined using the geometric mean.

BPPARAM  A BiocParallelParam object indicating how calculations are to be parallelized.
Only relevant when x is a DelayedArray object.

subset_row, exprs_values  Soft-deprecated equivalents to the arguments above.
**Description**

Compute log-transformed normalized expression values from a count matrix in a `SingleCellExperiment` object.
Usage

logNormCounts(x, ...)

## S4 method for signature 'SummarizedExperiment'
logNormCounts(
  x,
  size.factors = NULL,
  log = NULL,
  transform = c("log", "none", "asinh"),
  pseudo.count = 1,
  center.size.factors = TRUE,
  ...,
  subset.row = NULL,
  normalize.all = FALSE,
  assay.type = "counts",
  name = NULL,
  BPPARAM = SerialParam(),
  size_factors = NULL,
  pseudo_count = NULL,
  center_size_factors = NULL,
  exprs_values = NULL
)

## S4 method for signature 'SingleCellExperiment'
logNormCounts(
  x,
  size.factors = sizeFactors(x),
  log = NULL,
  transform = c("log", "none", "asinh"),
  pseudo.count = 1,
  center.size.factors = TRUE,
  ...,
  subset.row = NULL,
  normalize.all = FALSE,
  assay.type = "counts",
  use.altexps = FALSE,
  name = NULL,
  BPPARAM = SerialParam(),
  size_factors = NULL,
  pseudo_count = NULL,
  center_size_factors = NULL,
  exprs_values = NULL,
  use_altexps = NULL
)
Arguments

- **x**: A `SinglecellExperiment` or `SummarizedExperiment` object containing a count matrix.
- **...**: For the generic, additional arguments passed to specific methods.
  - For the methods, additional arguments passed to `normalizeCounts`.
- **size.factors**: A numeric vector of cell-specific size factors. Alternatively NULL, in which case the size factors are computed from x.
- **log**: Logical scalar indicating whether normalized values should be log2-transformed. This is retained for back-compatibility and will override any setting of `transform`. Users should generally use `transform` instead to specify the transformation.
- **transform**: String specifying the transformation (if any) to apply to the normalized expression values.
- **pseudo.count**: Numeric scalar specifying the pseudo-count to add when `transform="log"`.
- **center.size.factors**: Logical scalar indicating whether size factors should be centered at unity before being used.
- **subset.row**: A vector specifying the subset of rows of x for which to return normalized values. If `size.factors=NULL`, the size factors are also computed from this subset.
- **normalize.all**: Logical scalar indicating whether to return normalized values for all genes. If `TRUE`, any non-NULL value for `subset.row` is only used to compute the size factors. Ignored if `subset.row=NULL` or `size.factors` is supplied.
- **assay.type**: A string or integer scalar specifying the assay of `x` containing the count matrix.
- **name**: String containing an assay name for storing the output normalized values. Defaults to "logcounts" when `transform="log", "ashcounts" when `transform="asinh", and "normcounts" otherwise.
- **BPPARAM**: A `BiocParallelParam` object specifying how library size factor calculations should be parallelized. Only used if `size.factors` is not specified.
- **size_factors, pseudo_count, center_size_factors, exprs_values**: Deprecated.
- **use.alterexps, use_altexps**: Deprecated, use `applySCE` instead (see Examples).

Details

This function is a convenience wrapper around `normalizeCounts`. It returns a `SingleCellExperiment` or `SummarizedExperiment` containing the normalized values in a separate assay. This makes it easier to perform normalization by avoiding book-keeping errors during a long analysis workflow.

If `NULL`, size factors are determined as described in `normalizeCounts`. `subset.row` and `normalize.all` have the same interpretation as for `normalizeCounts`.

If `x` is a `SingleCellExperiment`, normalization is not applied to any alternative Experiments. Users can call `applySCE` to perform the normalization on each alternative Experiment - see Examples. Any Experiment-specific size factors will be automatically used, otherwise library size-based factors will be derived from the column sums.
Value

\(x\) is returned containing the (log-)normalized expression values in an additional assay named as \(\text{name}\).

If \(x\) is a \texttt{SingleCellExperiment}, the size factors used for normalization are stored in \texttt{sizeFactors}. These are centered if \texttt{center.size.factors=}TRUE.

Author(s)

Aaron Lun, based on code by Davis McCarthy

See Also

\texttt{normalizeCounts}, which is used to compute the normalized expression values.

Examples

```r
example_sce <- mockSCE()

# Standard library size normalization:
example_sce2 <- logNormCounts(example_sce)
assayNames(example_sce2)
logcounts(example_sce2)[1:5,1:5]

# Without logging, the assay is 'normcounts':
example_sce2 <- logNormCounts(example_sce, log=FALSE)
assayNames(example_sce2)
normcounts(example_sce2)[1:5,1:5]

# Pre-loading with size factors:
example_sce2 <- computeMedianFactors(example_sce)
example_sce2 <- logNormCounts(example_sce2)
logcounts(example_sce2)[1:5,1:5]

# Also normalizing the alternative experiments:
example_sce2 <- applySCE(example_sce, logNormCounts)
logcounts(altExp(example_sce2))[1:5,1:5]
```

---

\textbf{makePerCellDF}

\textit{Create a per-cell data.frame}

Description

Create a per-cell data.frame (i.e., where each row represents a cell) from a \texttt{SingleCellExperiment}, most typically for creating custom \texttt{ggplot2} plots.
Usage

```R
makePerCellDF(
  x,
  features = NULL,
  assay.type = "logcounts",
  use.coldata = TRUE,
  use.dimred = TRUE,
  use.altexps = TRUE,
  prefix.altexps = FALSE,
  check.names = FALSE,
  swap.rownames = NULL,
  exprs_values = NULL,
  use_dimred = NULL,
  use_altexps = NULL,
  prefix_altexps = NULL,
  check_names = NULL
)
```

Arguments

- **x**: A `SingleCellExperiment` object. This is expected to have non-NULL row names.
- **features**: Character vector specifying the features for which to extract expression profiles across cells. May also include features in alternative Experiments if permitted by `use.altexps`.
- **assay.type**: String or integer scalar indicating the assay to use to obtain expression values. Must refer to a matrix-like object with integer or numeric values.
- **use.coldata**: Logical scalar indicating whether column metadata of `x` should be included. Alternatively, a character or integer vector specifying the column metadata fields to use.
- **use.dimred**: Logical scalar indicating whether data should be extracted for dimensionality reduction results in `x`. Alternatively, a character or integer vector specifying the dimensionality reduction results to use.
- **use.altexps**: Logical scalar indicating whether (meta)data should be extracted for alternative experiments in `x`. Alternatively, a character or integer vector specifying the alternative experiments to use.
- **prefix.altexps**: Logical scalar indicating whether `altExp`-derived fields should be prefixed with the name of the alternative Experiment.
- **check.names**: Logical scalar indicating whether column names of the output should be made syntactically valid and unique.
- **swap.rownames**: String specifying the `rowData` column containing the features. If NULL, `rownames(x)` is used.
- **exprs_values, use_dimred, use_altexps, prefix_altexps, check_names**: Soft-deprecated equivalents of the arguments described above.
makePerCellDF

Details

This function enables us to conveniently create a per-feature data.frame from a SingleCellExperiment. Each row of the returned data.frame corresponds to a column in x, while each column of the data.frame corresponds to one aspect of the (meta)data in x.

Columns are provided in the following order:

1. Columns named according to the entries of features represent the expression values across cells for the specified feature in the assay.type assay.
2. Columns named according to the columns of colData(x) represent column metadata variables. This consists of all variables if use.coldata=TRUE, no variables if use.coldata=FALSE, and only the specified variables if use.coldata is set to an integer or character vector.
3. Columns named in the format of <DIM>.<NUM> represent the <NUM>th dimension of the dimensionality reduction result <DIM>. This is generated for all dimensionality reduction results if use.dimred=TRUE, none if use.dimred=FALSE, and only the specified results if use.dimred is set to an integer or character vector.
4. Columns named according to the row names of successive alternative Experiments, representing the assay data in these objects. These columns are only included if they are specified in features and if use.altexps is set. Column names are prefixed with the name of the alternative Experiment if prefix.altexps=TRUE.

By default, nothing is done to resolve syntactically invalid or duplicated column names. check.names=TRUE, this is resolved by passing the column names through make.names. Of course, as a result, some columns may not have the same names as the original fields in x.

Value

A data.frame containing one field per aspect of data in x - see Details. Each row corresponds to a cell (i.e., column) of x.

Author(s)

Aaron Lun

See Also

makePerFeatureDF, for the feature-level equivalent.

Examples

sce <- mockSCE()
sce <- logNormCounts(sce)
reducedDim(sce, "PCA") <- matrix(rnorm(ncol(sce)*10), ncol=10) # made-up PCA.

df <- makePerCellDF(sce, features="Gene_0001")
head(df)
Create a per-feature data.frame (i.e., where each row represents a feature) from a `SingleCellExperiment`, most typically for creating custom `ggplot2` plots.

### Usage

```r
makePerFeatureDF(
  x, 
  cells = NULL, 
  assay.type = "logcounts", 
  use.rowdata = TRUE, 
  check.names = FALSE, 
  exprs_values = NULL, 
  check_names = NULL 
)
```

### Arguments

- **x**: A `SingleCellExperiment` object. This is expected to have non-NULL row names.
- **cells**: Character vector specifying the features for which to extract expression profiles across cells.
- **assay.type**: String or integer scalar indicating the assay to use to obtain expression values. Must refer to a matrix-like object with integer or numeric values.
- **use.rowdata**: Logical scalar indicating whether row metadata of `x` should be included. Alternatively, a character or integer vector specifying the row metadata fields to use.
- **check.names**: Logical scalar indicating whether column names of the output should be made syntactically valid and unique.
- **exprs_values, check_names**: Soft-deprecated equivalents to the arguments above.

### Details

This function enables us to conveniently create a per-feature data.frame from a `SingleCellExperiment`. Each row of the returned data.frame corresponds to a row in `x`, while each column of the data.frame corresponds to one aspect of the (meta)data in `x`.

Columns are provided in the following order:

1. Columns named according to the entries of `cells` represent the expression values across features for the specified cell in the `assay.type` assay.
2. Columns named according to the columns of rowData(x) represent the row metadata variables. This consists of all variables if use.rowdata=TRUE, no variables if use.rowdata=FALSE, and only the specified variables if use.rowdata is set to an integer or character vector. By default, nothing is done to resolve syntactically invalid or duplicated column names. check_names=TRUE, this is resolved by passing the column names through make.names. Of course, as a result, some columns may not have the same names as the original fields in x.

Value
A data.frame containing one field per aspect of data in x - see Details. Each row corresponds to a feature (i.e., row) of x.

Author(s)
Aaron Lun

See Also
makePerCellDF, for the cell-level equivalent.

Examples
example_sce <- mockSCE()
example_sce <- logNormCounts(example_sce)
rowData(example_sce)$Length <- runif(nrow(example_sce))

df <- makePerFeatureDF(example_sce, cells="Cell_001")
head(df)

medianSizeFactors  Compute median-based size factors

Description
Define per-cell size factors by taking the median of ratios to a reference expression profile (a la DESeq).

Usage
medianSizeFactors(x, ...)

## S4 method for signature 'ANY'
medianSizeFactors(x, subset.row = NULL, reference = NULL, subset_row = NULL)

## S4 method for signature 'SummarizedExperiment'
medianSizeFactors(x, ..., assay.type = "counts", exprs_values = NULL)

computeMedianFactors(x, ...)

---

medianSizeFactors

Compute median-based size factors

---

Description
Define per-cell size factors by taking the median of ratios to a reference expression profile (a la DESeq).

Usage
medianSizeFactors(x, ...)

## S4 method for signature 'ANY'
medianSizeFactors(x, subset.row = NULL, reference = NULL, subset_row = NULL)

## S4 method for signature 'SummarizedExperiment'
medianSizeFactors(x, ..., assay.type = "counts", exprs_values = NULL)

computeMedianFactors(x, ...)

---
medianSizeFactors

Arguments

x  
   For medianSizeFactors, a numeric matrix of counts with one row per feature and column per cell. Alternatively, a SummarizedExperiment or SingleCellExperiment containing such counts.

For computeMedianFactors, only a SingleCellExperiment is accepted.

...  
   For the medianSizeFactors generic, arguments to pass to specific methods.

For the SummarizedExperiment method, further arguments to pass to the ANY method.

For computeMedianFactors, further arguments to pass to medianSizeFactors.

subset.row  
   A vector specifying whether the size factors should be computed from a subset of rows of x.

reference  
   A numeric vector of length equal to nrow(x), containing the reference expression profile. Defaults to rowMeans(x).

subset_row, exprs_values  
   Soft-deprecated equivalent to the arguments above.

assay.type  
   String or integer scalar indicating the assay of x containing the counts.

Details

This function implements a modified version of the DESeq2 size factor calculation. For each cell, the size factor is proportional to the median of the ratios of that cell’s counts to reference. The assumption is that most genes are not DE between the cell and the reference, such that the median captures any systematic increase due to technical biases.

The modification stems from the fact that we use the arithmetic mean instead of the geometric mean to compute the default reference, as the former is more robust to the many zeros in single-cell RNA sequencing data. We also ignore all genes with values of zero in reference, as this usually results in undefined ratios when reference is itself computed from x.

Value

For medianSizeFactors, a numeric vector of size factors is returned for all methods.

For computeMedianFactors, x is returned containing the size factors in sizeFactors(x).

Caveats

For typical scRNA-seq datasets, the median-based approach tends to perform poorly, for various reasons:

• The high number of zeroes in the count matrix means that the median ratio for each cell is often zero. If this method must be used, we recommend subsetting to only the highest-abundance genes to avoid problems with zeroes. (Of course, the smaller the subset, the more sensitive the results are to noise or violations of the non-DE majority.)

• The default reference effectively requires a non-DE majority of genes between any pair of cells in the dataset. This is a strong assumption for heterogeneous populations containing many cell types; most genes are likely to exhibit DE between at least one pair of cell types.
For these reasons, the simpler `librarySizeFactors` is usually preferred, which is no less inaccurate but is at least guaranteed to return a positive size factor for any cell with non-zero counts.

One valid application of this method lies in the normalization of antibody-derived tag counts for quantifying surface proteins. These counts are usually large enough to avoid zeroes yet are also susceptible to strong composition biases that preclude the use of `librarySizeFactors`. In such cases, we would also set `reference` to some estimate of the the ambient profile. This assumes that most proteins are not expressed in each cell; thus, counts for most tags for any given cell can be attributed to background contamination that should not be DE between cells.

**Author(s)**

Aaron Lun

**See Also**

`normalizeCounts` and `logNormCounts`, where these size factors can be used.

`librarySizeFactors` and `geometricSizeFactors` for other simple methods for computing size factors.

**Examples**

```r
example_sce <- mockSCE()
summary(medianSizeFactors(example_sce))
```

---

**mockSCE**

*Mock up a SingleCellExperiment*

**Description**

Mock up a `SingleCellExperiment` containing simulated data, for use in documentation examples.

**Usage**

```r
mockSCE(ncells = 200, ngenes = 2000, nspikes = 100)
```

**Arguments**

- `ncells` Integer scalar, number of cells to simulate.
- `ngenes` Integer scalar, number of genes to simulate.
- `nspikes` Integer scalar, number of spike-in transcripts to simulate.

**Details**

Users should set a seed to obtain reproducible results from this function.
Value

A SingleCellExperiment object containing a count matrix in the "counts" assay, a set of simulated colData fields, and spike-in data in the "Spikes" field of altExps.

Author(s)

Aaron Lun

See Also

SingleCellExperiment, for the constructor.

Examples

set.seed(1000)
sce <- mockSCE()
sce

normalizeCounts

Compute normalized expression values

Description

Compute (log-)normalized expression values by dividing counts for each cell by the corresponding size factor.

Usage

normalizeCounts(x, ...)

## S4 method for signature 'ANY'
normalizeCounts(
x,
size.factors = NULL,
log = NULL,
transform = c("log", "none", "asinh"),
pseudo.count = 1,
center.size.factors = TRUE,
subset.row = NULL,
normalize.all = FALSE,
downsamp;e = FALSE,
down.target = NULL,
down.prop = 0.01,
BPPARAM = SerialParam(),
size_factors = NULL,
pseudo_count = NULL,
normalizeCounts

center_size_factors = NULL,
subset_row = NULL,
down_target = NULL,
down_prop = NULL
)

## S4 method for signature 'SummarizedExperiment'
normalizeCounts(x, ..., assay.type = "counts", exprs_values = NULL)

## S4 method for signature 'SingleCellExperiment'
normalizeCounts(x, size.factors = sizeFactors(x), ...)

Arguments

x               A numeric matrix-like object containing counts for cells in the columns and features in the rows. Alternatively, a SingleCellExperiment or SummarizedExperiment object containing such a count matrix.
...             For the generic, arguments to pass to specific methods.
size.factors    A numeric vector of cell-specific size factors. Alternatively NULL, in which case the size factors are computed from x.
log             Logical scalar indicating whether normalized values should be log2-transformed. This is retained for back-compatibility and will override any setting of transform. Users should generally use transform instead to specify the transformation.
transform       String specifying the transformation (if any) to apply to the normalized expression values.
pseudo.count    Numeric scalar specifying the pseudo-count to add when transform="log".
center.size.factors
subsample       Logical scalar indicating whether size factors should be centered at unity before being used.
subset.row      A vector specifying the subset of rows of x for which to return normalized values. If size.factors=NULL, the size factors are also computed from this subset.
normalize.all   Logical scalar indicating whether to return normalized values for all genes. If TRUE, any non-NULL value for subset.row is only used to compute the size factors. Ignored if subset.row=NULL or size.factors is supplied.
downsampling    Logical scalar indicating whether downsampling should be performed prior to scaling and log-transformation.
down.target     Numeric scalar specifying the downsampling target when downsample=TRUE. If NULL, this is defined by down.prop and a warning is emitted.
down.prop       Numeric scalar between 0 and 1 indicating the quantile to use to define the downsampling target. Only used when downsample=TRUE.
**Details**

Normalized expression values are computed by dividing the counts for each cell by the size factor for that cell. This removes cell-specific scaling biases due to differences in sequencing coverage, capture efficiency or total RNA content. The assumption is that such biases affect all genes equally (in a scaling manner) and thus can be removed through division by a per-cell size factor.

If `transform="log"`, log-normalized values are calculated by adding `pseudo_count` to the normalized count and performing a log2-transformation. Differences in values between cells can be interpreted as log-fold changes, which are generally more relevant than differences on the untransformed scale. This provides a suitable input to downstream functions computing, e.g., Euclidean differences, which are effectively an average of the log-fold changes across genes.

Alternatively, if `transform="asinh"`, an inverse hyperbolic transformation is performed. This is commonly used in cytometry and converges to the log2-transformation at high normalized values. (We adjust the scale so that the results are comparable to log2-values, though the actual definition uses natural log.) For non-negative inputs, the main practical difference from a log2-transformation is that there is a bigger gap between transformed values derived from zero and those derived from non-zero inputs.

If the size factors are `NULL`, they are determined automatically from `x`. The sum of counts for each cell is used to compute a size factor via the `librarySizeFactors` function. For the SingleCellExperiment method, size factors are extracted from `sizeFactors(x)` if available, otherwise they are computed from the assay containing the count matrix.

If `subset.row` is specified, the output of the function is equivalent to supplying `x[subset.row,]` in the first place. The exception is if `normalize.all=TRUE`, in which case `subset.row` is only used during the size factor calculation; once computed, the size factors are then applied to all genes and the full matrix is returned.

**Value**

A numeric matrix-like object containing normalized expression values, possibly transformed according to `transform`. This has the same dimensions as `x`, unless `subset.row` is specified and `normalize.all=FALSE`.

**Centering the size factors**

If `center.size.factors=TRUE`, size factors are centred at unity prior to calculation of normalized expression values. This ensures that the computed expression values can be interpreted as being on the same scale as original counts. We can then compare abundances between features normalized with different sets of size factors; the most common use of this fact is in the comparison between spike-in and endogenous abundances when modelling technical noise (see `modelGeneVarWithSpikes` package for an example).
In the specific case of transform="log", centering of the size factors ensures the pseudo-count can actually be interpreted as a count. This is important as it implies that the pseudo-count's impact will diminish as sequencing coverage improves. Thus, if the size factors are centered, differences between log-normalized expression values will more closely approximate the true log-fold change with increasing coverage, whereas this would not be true of other metrics like log-CPMs with a fixed offset.

The disadvantage of using centered size factors is that the expression values are not directly comparable across different calls to normalizeCounts, typically for multiple batches. In theory, this is not a problem for metrics like the CPM, but in practice, we have to apply batch correction methods anyway to perform any joint analysis - see multiBatchNorm for more details.

### Downsampling instead of scaling

If downsample=TRUE, counts for each cell are randomly downsampled instead of being scaled. This is occasionally useful for avoiding artifacts caused by scaling count data with a strong mean-variance relationship. Each cell is downsampled according to the ratio between down.target and that cell's size factor. (Cells with size factors below the target are not downsampled and are directly scaled by this ratio.) Any transformation specified by transform is then applied to the downsampled counts.

We automatically set down.target to the 1st percentile of size factors across all cells involved in the analysis, but this is only appropriate if the resulting expression values are not compared across different normalizeCounts calls. To obtain expression values that are comparable across different normalizeCounts calls (e.g., in modelGeneVarWithSpikes or multiBatchNorm), down_target should be manually set to a constant target value that can be considered a low size factor in every call.

### Author(s)

Aaron Lun

### See Also

- logNormCounts, which wraps this function for convenient use with SingleCellExperiment instances.
- librarySizeFactors, to compute the default size factors.
- downsampleMatrix, to perform the downsampling.

### Examples

```r
example_sce <- mockSCE()

# Standard scaling + log-transformation:
normed <- normalizeCounts(example_sce)
normed[1:5,1:5]

# Scaling without transformation:
normed <- normalizeCounts(example_sce, log=FALSE)
normed[1:5,1:5]

# Downscaling with transformation:
```


numDetectedAcrossCells

Number of detected expression values per group of cells

Description
Computes the number of detected expression values (by default, defined as non-zero counts) for each feature in each group of cells. This function is deprecated: use `summarizeAssayByGroup` instead.

Usage

numDetectedAcrossCells(x, ...)

## S4 method for signature 'ANY'
numDetectedAcrossCells(
  x,
  ids,
  subset.row = NULL,
  subset.col = NULL,
  store.number = "ncells",
  average = FALSE,
  threshold = 0,
  BPPARAM = SerialParam(),
  subset_row = NULL,
  subset_col = NULL,
  store_number = NULL,
  detection.limit = NULL
)

## S4 method for signature 'SummarizedExperiment'
numDetectedAcrossCells(x, ..., assay.type = "counts", exprs_values = NULL)

Arguments

x A numeric matrix of counts containing features in rows and cells in columns. Alternatively, a `SummarizedExperiment` object containing such a count matrix.
... For the generic, further arguments to pass to specific methods.
For the SummarizedExperiment method, further arguments to pass to the ANY method.

ids A factor specifying the group to which each cell in x belongs. Alternatively, a DataFrame of such vectors or factors, in which case each unique combination of levels defines a group.

subset.row An integer, logical or character vector specifying the features to use. If NULL, defaults to all features. For the SingleCellExperiment method, this argument will not affect alternative Experiments, where aggregation is always performed for all features (or not at all, depending on use_alt_exps).

subset.col An integer, logical or character vector specifying the cells to use. Defaults to all cells with non-NA entries of ids.

store.number String specifying the field of the output colData to store the number of cells in each group. If NULL, nothing is stored.

average Logical scalar indicating whether the proportion of non-zero counts in each group should be computed instead.

threshold A numeric scalar specifying the threshold above which a gene is considered to be detected.

BPPARAM A BiocParallelParam object specifying whether summation should be parallelized.

subset_row, subset_col, detection_limit, store_number, exprs_values
Soft-deprecated equivalents of the arguments above.

assay.type A string or integer scalar specifying the assay of x containing the matrix of counts (or any other expression quantity that can be meaningfully summed).

Value
A SummarizedExperiment is returned containing a count matrix in the first assay. Each column corresponds to group as defined by a unique level or combination of levels in ids. Each entry of the matrix contains the number of cells with detected expression for a feature and group. The identities of the levels for each column are reported in the colData. If average=TRUE, the assay is instead a numeric matrix containing the proportion of detected values.

Author(s)
Aaron Lun

See Also
sumCountsAcrossCells, which computes the sum of counts within a group.

Examples

eexample_sce <- mockSCE()

d <- sample(LETTERS[1:5], ncol(example_sce), replace=TRUE)
bycol <- numDetectedAcrossCells(example_sce, d)
numDetectedAcrossFeatures

Number of detected expression values per group of features

Description

Computes the number of detected expression values (by default, defined as non-zero counts) for each group of features in each cell.

Usage

numDetectedAcrossFeatures(x, ...)

## S4 method for signature 'ANY'
numDetectedAcrossFeatures(
  x,
  ids,
  subset.row = NULL,
  subset.col = NULL,
  average = FALSE,
  threshold = 0,
  BPPARAM = SerialParam(),
  subset_row = NULL,
  subset_col = NULL,
  detection_limit = NULL
)

## S4 method for signature 'SummarizedExperiment'
numDetectedAcrossFeatures(x, ..., assay.type = "counts", exprs_values = NULL)

Arguments

x          A numeric matrix of counts containing features in rows and cells in columns. Alternatively, a SummarizedExperiment object containing such a count matrix.

...         For the generic, further arguments to pass to specific methods.

For the SummarizedExperiment method, further arguments to pass to the ANY method.

ids        A factor of length nrow(x), specifying the set to which each feature in x belongs. Alternatively, a list of integer or character vectors, where each vector specifies the indices or names of features in a set. Logical vectors are also supported.

subset.row An integer, logical or character vector specifying the features to use. Defaults to all features.
subset.col  An integer, logical or character vector specifying the cells to use. Defaults to all cells with non-NA entries of ids.

average  Logical scalar indicating whether the proportion of non-zero counts in each group should be computed instead.

threshold  A numeric scalar specifying the threshold above which a gene is considered to be detected.

BPPARAM  A BiocParallelParam object specifying whether summation should be parallelized.

subset_row, subset_col, detection_limit, exprs_values
  Soft-deprecated equivalents of the arguments above.

assay.type  A string or integer scalar specifying the assay of x containing the matrix of counts (or any other expression quantity that can be meaningfully summed).

Value

An integer matrix containing the number of detected expression values in each group of features (row) and cell (column). If average=TRUE, this is instead a numeric matrix containing the proportion of detected values.

Author(s)

Aaron Lun

See Also

sumCountsAcrossFeatures, on which this function is based.

Examples

element_sce <- mockSCE()

ids <- sample(paste0("GENE_", 1:100), nrow(element_sce), replace=TRUE)
byrow <- numDetectedAcrossFeatures(element_sce, ids)
head(byrow[,1:10])

perCellQCFilters  Compute filters for low-quality cells

Description

Identifies low-quality cells as outliers for frequently used QC metrics.
Usage

perCellQCFilters(
  x,
  sum.field = "sum",
  detected.field = "detected",
  sub.fields = NULL,
  ...
)

Arguments

x A DataFrame containing per-cell QC statistics, as computed by perCellQCMetrics.
sum.field String specifying the column of x containing the library size for each cell.
detected.field String specifying the column of x containing the number of detected features per cell.
sub.fields Character vector specifying the column(s) of x containing the percentage of counts in subsets of “control features”, usually mitochondrial genes or spike-in transcripts.
  If set to TRUE, this will default to all columns in x with names following the patterns "subsets_.*_percent" and "altexps_.*_percent".
...
Further arguments to pass to isOutlier.

Details

This function simply calls isOutlier on the various QC metrics in x.

• For sum.field, small outliers are detected. These are considered to represent low-quality cells that have not been insufficiently sequenced. Detection is performed on the log-scale to adjust for a heavy right tail and to improve resolution at zero.

• For detected.field, small outliers are detected. These are considered to represent low-quality cells with low-complexity libraries. Detection is performed on the log-scale to adjust for a heavy right tail. This is done on the log-scale to adjust for a heavy right tail and to improve resolution at zero.

• For each column specified by sub.fields, large outliers are detected. This aims to remove cells with high spike-in or mitochondrial content, usually corresponding to damaged cells. While these distributions often have heavy right tails, the putative low-quality cells are often present in this tail; thus, transformation is not performed to ensure maintain resolution of the filter.

Users can control the outlier detection (e.g., change the number of MADs, specify batches) by passing appropriate arguments to ....

Value

A DataFrame with one row per cell and containing columns of logical vectors. Each column specifies a reason for why a cell was considered to be low quality, with the final discard column indicating whether the cell should be discarded.
Author(s)
Aaron Lun

See Also

perCellQCMetrics, for calculation of these metrics.
isOutlier, to identify outliers with a MAD-based approach.

Examples

```r
example_sce <- mockSCE()
x <- perCellQCMetrics(example_sce, subsets=list(Mito=1:100))

discarded <- perCellQCFilters(x,
   sub.fields=c("subsets_Mito_percent", "altexps_Spikes_percent"))
colSums(as.data.frame(discarded))
```

---

**perCellQCMetrics**  
Per-cell quality control metrics

Description

Compute per-cell quality control metrics for a count matrix or a SingleCellExperiment.

Usage

```r
perCellQCMetrics(x, ...)
```

### S4 method for signature 'ANY'
```r
perCellQCMetrics(
   x,
   subsets = NULL,
   percent.top = integer(0),
   threshold = 0,
   BPPARAM = SerialParam(),
   flatten = TRUE,
   percent_top = NULL,
   detection_limit = NULL
)
```

### S4 method for signature 'SummarizedExperiment'
```r
perCellQCMetrics(x, ..., assay.type = "counts", exprs_values = NULL)
```

### S4 method for signature 'SingleCellExperiment'
```r
perCellQCMetrics(
   x,
```
Arguments

x
A numeric matrix of counts with cells in columns and features in rows. Alternatively, a SummarizedExperiment or SingleCellExperiment object containing such a matrix.

... For the generic, further arguments to pass to specific methods. For the SummarizedExperiment and SingleCellExperiment methods, further arguments to pass to the ANY method.

subsets 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 interesting feature subsets such as ERCC spike-in transcripts or mitochondrial genes.

percent.top An integer vector specifying the size(s) of the top set of high-abundance genes. Used to compute the percentage of library size occupied by the most highly expressed genes in each cell.

threshold A numeric scalar specifying the threshold above which a gene is considered to be detected.

BPPARAM A BiocParallelParam object specifying how parallelization should be performed.

flatten Logical scalar indicating whether the nested DataFrames in the output should be flattened.

percent_top, detection_limit, exprs_values, use_altexps Soft deprecated equivalents to the arguments described above.

assay.type A string or integer scalar indicating which assays in the x contains the count matrix.

use.altexps Logical scalar indicating whether QC statistics should be computed for alternative Experiments in x. If TRUE, statistics are computed for all alternative experiments. Alternatively, an integer or character vector specifying the alternative Experiments to use to compute QC statistics. Alternatively NULL, in which case we only use alternative experiments that contain the specified assay.type. Alternatively FALSE, in which case alternative experiments are not used.
Details

This function calculates useful QC metrics for identification and removal of potentially problematic cells. Obvious per-cell metrics are the sum of counts (i.e., the library size) and the number of detected features. The percentage of counts in the top features also provides a measure of library complexity.

If subsets is specified, these statistics are also computed for each subset of features. This is useful for investigating gene sets of interest, e.g., mitochondrial genes, Y chromosome genes. These statistics are stored as nested DataFrames in the subsets field of the output. For example, if the input subsets contained "Mito" and "Sex", the output would look like:

```
output
|-- sum
|-- detected
|-- percent.top
 +-- subsets
    |-- Mito
       |-- sum
       |-- detected
       +-- percent
    +-- Sex
       |-- sum
       |-- detected
       +-- percent
```

Here, the percent field contains the percentage of each cell’s count sum assigned to each subset.

If use.altexps=TRUE, the same statistics are computed for each alternative experiment in x. This can also be an integer or character vector specifying the alternative Experiments to use. These statistics are also stored as nested DataFrames, this time in the altexps field of the output. For example, if x contained the alternative Experiments "Spike" and "Ab", the output would look like:

```
output
|-- sum
|-- detected
|-- percent.top
 +-- altexps
    |-- Spike
       |-- sum
       |-- detected
       +-- percent
    +-- Ab
       |-- sum
       |-- detected
       +-- percent
    +-- total
```

The total field contains the total sum of counts for each cell across the main and alternative Experiments. The percent field contains the percentage of the total count in each alternative Experiment for each cell.
Note that the denominator for `altexps`...`percent` is not the same as the denominator for `subset`...`percent`. For example, if `subsets` contains a set of mitochondrial genes, the mitochondrial percentage would be computed as a fraction of the total endogenous coverage, while the `altexps` percentage would be computed as a fraction of the total coverage across all (endogenous and artificial) features.

If `flatten=TRUE`, the nested DataFrames are flattened by concatenating the column names with underscores. This means that, say, the `subsets$Mito$sum` nested field becomes the top-level `subsets_Mito_sum` field. A flattened structure is more convenient for end-users performing interactive analyses, but less convenient for programmatic access as artificial construction of strings is required.

**Value**

A DataFrame of QC statistics where each row corresponds to a column in `x`. This contains the following fields:

- `sum`: numeric, the sum of counts for each cell.
- `detected`: numeric, the number of observations above `threshold`.

If `flatten=FALSE`, the DataFrame will contain the additional columns:

- `percent.top`: numeric matrix, the percentage of counts assigned to the top most highly expressed genes. Each column of the matrix corresponds to an entry of `percent.top`, sorted in increasing order.
- `subsets`: A nested DataFrame containing statistics for each subset, see Details.
- `altexps`: A nested DataFrame containing statistics for each alternative experiment, see Details. This is only returned for the `SingleCellExperiment` method.
- `total`: numeric, the total sum of counts for each cell across main and alternative Experiments. This is only returned for the `SingleCellExperiment` method.

If `flatten=TRUE`, nested matrices and DataFrames are flattened to remove the hierarchical structure from the output DataFrame.

**Author(s)**

Aaron Lun

**See Also**

- `addPerCellQCMetrics`, to add the QC metrics to the column metadata.

**Examples**

```r
example_sce <- mockSCE()
stats <- perCellQCMetrics(example_sce)
stats

# With subsets.
stats2 <- perCellQCMetrics(example_sce, subsets=list(Mito=1:10),
                           flatten=FALSE)
```
stats2$subsets

# With alternative Experiments.
pretend.spike <- ifelse(seq_len(nrow(example_sce)) < 10, "Spike", "Gene")
alt_sce <- splitAltExps(example_sce, pretend.spike)
stats3 <- perCellQCMetrics(alt_sce, flatten=FALSE)
stats3$altexps

---

**perFeatureQCMetrics**  
*Per-feature quality control metrics*

**Description**

Compute per-feature quality control metrics for a count matrix or a SummarizedExperiment.

**Usage**

```r
perFeatureQCMetrics(x, ...)  
## S4 method for signature 'ANY'
perFeatureQCMetrics(
  x,
  subsets = NULL,
  threshold = 0,
  BPPARAM = SerialParam(),
  flatten = TRUE,
  detection_limit = NULL
)

## S4 method for signature 'SummarizedExperiment'
perFeatureQCMetrics(x, ..., assay.type = "counts", exprs_values = NULL)
```

**Arguments**

- `x`  
  A numeric matrix of counts with cells in columns and features in rows.  
  Alternatively, a SummarizedExperiment or SingleCellExperiment object containing such a matrix.

- `...`  
  For the generic, further arguments to pass to specific methods.  
  For the SummarizedExperiment and SingleCellExperiment methods, further arguments to pass to the ANY method.

- `subsets`  
  A named list containing one or more vectors (a character vector of cell names, a logical vector, or a numeric vector of indices), used to identify interesting sample subsets such as negative control wells.

- `threshold`  
  A numeric scalar specifying the threshold above which a gene is considered to be detected.
**perFeatureQCMetrics**

**BPPARAM**
A `BiocParallelParam` object specifying how parallelization should be performed.

**flatten**
Logical scalar indicating whether the nested `DataFrames` in the output should be flattened.

**detection_limit, exprs_values**
Soft deprecated equivalents to the arguments described above.

**assay.type**
A string or integer scalar indicating which assays in the `x` contains the count matrix.

**Details**

This function calculates useful QC metrics for features, including the mean across all cells and the number of expressed features (i.e., counts above the detection limit).

If `subsets` is specified, the same statistics are computed for each subset of cells. This is useful for obtaining statistics for cell sets of interest, e.g., negative control wells. These statistics are stored as nested `DataFrames` in the output. For example, if `subsets` contained "empty" and "cellpool", the output would look like:

```
output
|-- mean
|-- detected
+++ subsets
  |-- empty
    |-- mean
    |-- detected
    +- ratio
  +- cellpool
    |-- mean
    |-- detected
    +- ratio
```

The ratio field contains the ratio of the mean within each subset to the mean across all cells.

If `flatten=TRUE`, the nested `DataFrames` are flattened by concatenating the column names with underscores. This means that, say, the `subsets$empty$mean` nested field becomes the top-level `subsets_empty_mean` field. A flattened structure is more convenient for end-users performing interactive analyses, but less convenient for programmatic access as artificial construction of strings is required.

**Value**

A `DataFrame` of QC statistics where each row corresponds to a row in `x`. This contains the following fields:

- `mean`: numeric, the mean counts for each feature.
- `detected`: numeric, the percentage of observations above threshold.

If `flatten=FALSE`, the output `DataFrame` also contains the `subsets` field. This a nested `DataFrame` containing per-feature QC statistics for each subset of columns.

If `flatten=TRUE`, `subsets` is flattened to remove the hierarchical structure.
**Author(s)**

Aaron Lun

**See Also**

`addPerFeatureQCMetrics`, to add the QC metrics to the row metadata.

**Examples**

```r
eample_sce <- mockSCE()
stats <- perFeatureQCMetrics(example_sce)
stats

# With subsets.
stats2 <- perFeatureQCMetrics(example_sce, subsets=list(Empty=1:10))
stats2
```

---

**quickPerCellQC**  
*Quick cell-level QC*

**Description**

A convenient utility that identifies low-quality cells based on frequently used QC metrics.

**Usage**

```r
quickPerCellQC(x, ...)
```

### S4 method for signature 'ANY'

```r
quickPerCellQC(
  x,
  sum.field = "sum",
  detected.field = "detected",
  sub.fields = NULL,
  ...,
  lib_size = NULL,
  n_features = NULL,
  percent_subsets = NULL
)
```

### S4 method for signature 'SummarizedExperiment'

```r
quickPerCellQC(
  x,
  ...,
  subsets = NULL,
  assay.type = "counts",
  ```
other.args = list(),
filter = TRUE
)

Arguments

x A DataFrame containing per-cell QC statistics, as computed by `perCellQCMetrics`. Alternatively, a SummarizedExperiment object that can be used to create such a DataFrame via `perCellQCMetrics`.

... For the generic, further arguments to pass to specific methods.
For the ANY method, further arguments to pass to `isOutlier`.
For the SummarizedExperiment method, further arguments to pass to the ANY method.

sum.field String specifying the column of `x` containing the library size for each cell.
detected.field String specifying the column of `x` containing the number of detected features per cell.

sub.fields Character vector specifying the column(s) of `x` containing the percentage of counts in subsets of “control features”, usually mitochondrial genes or spike-in transcripts.
If set to TRUE, this will default to all columns in `x` with names following the patterns "subsets_.*_percent" and "altexps_.*_percent".

lib_size, n_features, percent_subsets
Soft-deprecated equivalents of the arguments above.

subsets, assay.type Arguments to pass to the `perCellQCMetrics` function, exposed here for convenience.

other.args A named list containing other arguments to pass to the `perCellQCMetrics` function.

filter Logical scalar indicating whether to filter out low-quality cells from `x`.

Details

For DataFrame `x`, this function simply calls `perCellQCFilters`. The latter should be directly used in such cases; DataFrame inputs are soft-deprecated here.

For SummarizedExperiment `x`, this function is simply a convenient wrapper around `perCellQCMetrics` and `perCellQCFilters`.

Value

If `filter=FALSE` or `x` is a DataFrame, a DataFrame is returned with one row per cell and containing columns of logical vectors. Each column specifies a reason for why a cell was considered to be low quality, with the final `discard` column indicating whether the cell should be discarded.

If `filter=TRUE`, `x` is returned with the low-quality cells removed. QC statistics and filtering information for all remaining cells are stored in the `colData`.
Author(s)

Aaron Lun

See Also

perCellQC_metrics, for calculation of these metrics.

perCellQC_filters, to define filter thresholds based on those metrics.

Examples

```r
example_sce <- mockSCE()
filtered_sce <- quickPerCellQC(example_sce, subsets=list(Mito=1:100),
  sub.fields=c("subsets_Mito_percent", "altexps_Spikes_percent"))
ncol(filtered_sce)

# Same result as the longer chain of expressions:
stats <- perCellQC_metrics(example_sce, subsets=list(Mito=1:100))
discard <- perCellQC_filters(stats,
  sub.fields=c("subsets_Mito_percent", "altexps_Spikes_percent"))
filtered_sce2 <- example_sce[,!discard$discard]
ncol(filtered_sce2)
```

---

**readSparseCounts**

*Read sparse count matrix from file*

**Description**

Reads a sparse count matrix from file containing a dense tabular format.

**Usage**

```r
readSparseCounts(
  file,
  sep = "\t",
  quote = NULL,
  comment.char = "",
  row.names = TRUE,
  col.names = TRUE,
  ignore.row = 0L,
  skip.row = 0L,
  ignore.col = 0L,
  skip.col = 0L,
  chunk = 1000L
)
```
**Arguments**

- **file**: A string containing a file path to a count table, or a connection object opened in read-only text mode.
- **sep**: A string specifying the delimiter between fields in `file`.
- **quote**: A string specifying the quote character, e.g., in column or row names.
- **comment.char**: A string specifying the comment character after which values are ignored.
- **row.names**: A logical scalar specifying whether row names are present.
- **col.names**: A logical scalar specifying whether column names are present.
- **ignore.row**: An integer scalar specifying the number of rows to ignore at the start of the file, *before* the column names.
- **skip.row**: An integer scalar specifying the number of rows to ignore at the start of the file, *after* the column names.
- **ignore.col**: An integer scalar specifying the number of columns to ignore at the start of the file, *before* the column names.
- **skip.col**: An integer scalar specifying the number of columns to ignore at the start of the file, *after* the column names.
- **chunk**: A integer scalar indicating the chunk size to use, i.e., number of rows to read at any one time.

**Details**

This function provides a convenient method for reading dense arrays from flat files into a sparse matrix in memory. Memory usage can be further improved by setting `chunk` to a smaller positive value.

The `ignore.*` and `skip.*` parameters allow irrelevant rows or columns to be skipped. Note that the distinction between the two parameters is only relevant when `row.names=FALSE` (for skipping/ignoring columns) or `col.names=FALSE` (for rows).

**Value**

A `dgCMatrix` containing double-precision values (usually counts) for each row (gene) and column (cell).

**Author(s)**

Aaron Lun

**See Also**

`read.table`, `readMM`
Examples

```r
tfile <- tempfile()
write.table(data.frame(A=1:5, B=0, C=0:4, row.names=letters[1:5]),
            file=tfile, col.names=NA, sep="\t", quote=FALSE)

readSparseCounts(tfile)
```

---

**reexports**

*Objects exported from other packages*

**Description**

These objects are imported from other packages. Follow the links below to see their documentation.

- beachmat
- whichNonZero

---

**scuttle-pkg**

*Single-cell utilities*

**Description**

The **scuttle** package provides some utility functions for single-cell 'omics data analysis. This includes some simple methods for computing and filtering on quality control; basic data transformations involving various types of scaling normalization; and flexible aggregation across cells or features.

**scuttle** also implements wrapper functions that simplify boilerplate for developers of client packages. This includes packages such as **scran**, **scater** and **DropletUtils**, to name a few. Note that much of the code here was inherited from the **scater** package.

**Author(s)**

Aaron Lun

---

**scuttle-utils**

*Developer utilities*

**Description**

Various utilities for re-use in packages that happen to depend on **scuttle**. These are exported simply to avoid re-writing them in downstream packages, and should not be touched by end-users.

**Author(s)**

Aaron Lun
sumCountsAcrossCells

**Description**

Sum counts or average expression values for each feature across groups of cells. This function is deprecated; use `summarizeAssayByGroup` instead.

**Usage**

```r
sumCountsAcrossCells(x, ...)
```

```r
## S4 method for signature 'ANY'
sumCountsAcrossCells(
  x,
  ids,
  subset.row = NULL,
  subset.col = NULL,
  store.number = "ncells",
  average = FALSE,
  BPPARAM = SerialParam(),
  subset_row = NULL,
  subset_col = NULL,
  store_number = NULL
)
```

```r
## S4 method for signature 'SummarizedExperiment'
sumCountsAcrossCells(x, ..., assay.type = "counts", exprs_values = NULL)
```

**Arguments**

- `x` A numeric matrix of expression values (usually counts) containing features in rows and cells in columns. Alternatively, a `SummarizedExperiment` object containing such a matrix.
- `...` For the generics, further arguments to be passed to specific methods.
- `ids` A factor specifying the group to which each cell in `x` belongs. Alternatively, a `DataFrame` of such vectors or factors, in which case each unique combination of levels defines a group.
- `subset.row` An integer, logical or character vector specifying the features to use. If `NULL`, defaults to all features. For the `SingleCellExperiment` method, this argument will not affect alternative Experiments, where aggregation is always performed for all features (or not at all, depending on `use_alt_exps`).
- `subset.col` An integer, logical or character vector specifying the cells to use. Defaults to all cells with non-NA entries of `ids`. 

store.number  String specifying the field of the output `colData` to store the number of cells in each group. If `NULL`, nothing is stored.

average  Logical scalar indicating whether the average should be computed instead of the sum. Alternatively, a string containing "mean", "median" or "none", specifying the type of average. ("none" is equivalent to FALSE.)

BPPARAM  A `BiocParallelParam` object specifying whether summation should be parallelized.

subset_row, subset_col, exprs_values, store_number  Soft-deprecated equivalents to the arguments described above.

assay.type  A string or integer scalar specifying the assay of `x` containing the matrix of counts (or any other expression quantity that can be meaningfully summed).

Details

These functions provide a convenient method for summing or averaging expression values across multiple columns for each feature. A typical application would be to sum counts across all cells in each cluster to obtain “pseudo-bulk” samples for further analyses, e.g., differential expression analyses between conditions.

The behaviour of `sumCountsAcrossCells` is equivalent to that of `colsum`. However, this function can operate on any matrix representation in `object`; can do so in a parallelized manner for large matrices without resorting to block processing; and can natively support combinations of multiple factors in `ids`.

Any NA values in `ids` are implicitly ignored and will not be considered during summation. This may be useful for removing undesirable cells by setting their entries in `ids` to NA. Alternatively, we can explicitly select the cells of interest with `subset_col`.

Setting `average=TRUE` will compute the average in each set rather than the sum. This is particularly useful if `x` contains expression values that have already been normalized in some manner, as computing the average avoids another round of normalization to account for differences in the size of each set. The same effect is obtained by setting `average="mean"`, while setting `average="median"` will instead compute the median across all cells.

Value

A `SummarizedExperiment` is returned with one column per level of `ids`. Each entry of the assay contains the sum or average across all cells in a given group (column) for a given feature (row). Columns are ordered by `levels(ids)` and the number of cells per level is reported in the "ncells" column metadata. For DataFrame `ids`, each column corresponds to a unique combination of levels (recorded in the `colData`).

Author(s)

Aaron Lun

See Also

`aggregateAcrossCells`, which also combines information in the `colData`.

`numDetectedAcrossCells`, which computes the number of cells with detected expression in each group.
sumCountsAcrossFeatures

Examples

```r
example_sce <- mockSCE()
ids <- sample(LETTERS[1:5], ncol(example_sce), replace=TRUE)

out <- sumCountsAcrossCells(example_sce, ids)
head(out)

batches <- sample(1:3, ncol(example_sce), replace=TRUE)
out2 <- sumCountsAcrossCells(example_sce, DataFrame(label=ids, batch=batches))
head(out2)
```

Description

Sum counts across feature sets

Usage

```r
sumCountsAcrossFeatures(x, 
```
ids
A factor of length nrow(x), specifying the set to which each feature in x belongs. Alternatively, a list of integer or character vectors, where each vector specifies the indices or names of features in a set. Logical vectors are also supported.

subset.row
An integer, logical or character vector specifying the features to use. Defaults to all features.

subset.col
An integer, logical or character vector specifying the cells to use. Defaults to all cells with non-NA entries of ids.

average
Logical scalar indicating whether the average should be computed instead of the sum.

BPPARAM
A BiocParallelParam object specifying whether summation should be parallelized.

subset_row, subset_col, exprs_values
Soft-deprecated equivalents of the arguments described above.

assay.type
A string or integer scalar specifying the assay of x containing the matrix of counts (or any other expression quantity that can be meaningfully summed).

Details
This function provides a convenient method for aggregating counts across multiple rows for each cell. Several possible applications are listed below:

- Using a list of genes in ids, we can obtain a summary expression value for all genes in one or more gene sets. This allows the activity of various pathways to be compared across cells.
- Genes with multiple mapping locations in the reference will often manifest as multiple rows with distinct Ensembl/Entrez IDs. These counts can be aggregated into a single feature by setting the shared identifier (usually the gene symbol) as ids.
- It is theoretically possible to aggregate transcript-level counts to gene-level counts with this function. However, it is often better to do so with dedicated functions (e.g., from the tximport or tximeta packages) that account for differences in length across isoforms.

The behaviour of this function is equivalent to that of rowsum. However, this function can operate on any matrix representation in object, and can do so in a parallelized manner for large matrices without resorting to block processing.

If ids is a factor, any NA values are implicitly ignored and will not be considered or reported. This may be useful, e.g., to remove undesirable feature sets by setting their entries in ids to NA.

Setting average=TRUE will compute the average in each set rather than the sum. This is particularly useful if x contains expression values that have already been normalized in some manner, as computing the average avoids another round of normalization to account for differences in the size of each set.

Value
A count matrix is returned with one row per level of ids. In each cell, counts for all features in the same set are summed together (or averaged, if average=TRUE). Rows are ordered according to levels(ids).
**summarizeAssayByGroup**

**Author(s)**

Aaron Lun

**See Also**

*aggregateAcrossFeatures*, to perform additional aggregation of row-level metadata.

*numDetectedAcrossFeatures*, to compute the number of detected features per cell.

**Examples**

```r
example_sce <- mockSCE()
ids <- sample(LETTERS, nrow(example_sce), replace=TRUE)
out <- sumCountsAcrossFeatures(example_sce, ids)
str(out)
```

**summarizeAssayByGroup  Summarize an assay by group**

**Description**

From an assay matrix, compute summary statistics for groups of cells. A typical example would be to compute various summary statistics for clusters.

**Usage**

```r
summarizeAssayByGroup(x, ...)
```

## S4 method for signature 'ANY'

```r
summarizeAssayByGroup(
  x,
  ids,
  subset.row = NULL,
  subset.col = NULL,
  statistics = c("mean", "sum", "num.detected", "prop.detected", "median"),
  store.number = "ncells",
  threshold = 0,
  BPPARAM = SerialParam()
)
```

## S4 method for signature 'SummarizedExperiment'

```r
summarizeAssayByGroup(x, ..., assay.type = "counts")
```
Arguments

`x`  
A numeric matrix containing features in rows and cells in columns. Alternatively, a `SummarizedExperiment` object containing such a matrix.

`...`  
For the generics, further arguments to be passed to specific methods.  
For the SummarizedExperiment method, further arguments to be passed to the ANY method.

`ids`  
A factor (or vector coercible into a factor) specifying the group to which each cell in `x` belongs. Alternatively, a `DataFrame` of such vectors or factors, in which case each unique combination of levels defines a group.

`subset.row`  
An integer, logical or character vector specifying the features to use. If `NULL`, defaults to all features.

`subset.col`  
An integer, logical or character vector specifying the cells to use. Defaults to all cells with non-NA entries of `ids`.

`statistics`  
Character vector specifying the type of statistics to be computed, see Details.

`store.number`  
String specifying the field of the output `colData` to store the number of cells in each group. If `NULL`, nothing is stored.

`threshold`  
A numeric scalar specifying the threshold above which a gene is considered to be detected.

`BPPARAM`  
A `BiocParallelParam` object specifying whether summation should be parallelized.

`assay.type`  
A string or integer scalar specifying the assay of `x` containing the assay to be summarized.

Details

These functions provide a convenient method for summing or averaging expression values across multiple columns for each feature. A typical application would be to sum counts across all cells in each cluster to obtain “pseudo-bulk” samples for further analyses, e.g., differential expression analyses between conditions.

For each feature, the chosen assay can be aggregated by:

- "sum", the sum of all values in each group. This makes the most sense for raw counts, to allow models to account for the mean-variance relationship.
- "mean", the mean of all values in each group. This makes the most sense for normalized and/or transformed assays.
- "median", the median of all values in each group. This makes the most sense for normalized and/or transformed assays, usually generated from large counts where discreteness is less of an issue.
- "num.detected" and "prop.detected", the number and proportion of values in each group that are non-zero. This makes the most sense for raw counts or sparsity-preserving transformations.

Any NA values in `ids` are implicitly ignored and will not be considered during summation. This may be useful for removing undesirable cells by setting their entries in `ids` to NA. Alternatively, we can explicitly select the cells of interest with `subset.col`.

If `ids` is a factor and contains unused levels, they will not be represented as columns in the output.
uniquifyFeatureNames

Value

A SummarizedExperiment is returned with one column per level of ids. Each entry of the assay contains the sum or average across all cells in a given group (column) for a given feature (row). Columns are ordered by levels(ids) and the number of cells per level is reported in the "ncells" column metadata. For DataFrame ids, each column corresponds to a unique combination of levels (recorded in the colData).

Author(s)

Aaron Lun

See Also

aggregateAcrossCells, which also combines information in the colData of x.

Examples

```r
example_sce <- mockSCE()
ids <- sample(LETTERS[1:5], ncol(example_sce), replace=TRUE)
out <- summarizeAssayByGroup(example_sce, ids)
out

batches <- sample(1:3, ncol(example_sce), replace=TRUE)
out2 <- summarizeAssayByGroup(example_sce, 
    DataFrame(label=ids, batch=batches))
head(out2)
```

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 and valid (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

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

* internal
  * reexports, 64
    * .assignIndicesToWorkers (scuttle-utils), 64
    * .bpNotSharedOrUp (scuttle-utils), 64
    * .checkCountMatrix (scuttle-utils), 64
    * .guessMinMean (scuttle-utils), 64
    * .ranksafeQR (scuttle-utils), 64
    * .splitColsByWorkers (scuttle-utils), 64
    * .splitRowsByWorkers (scuttle-utils), 64
    * .splitVectorByWorkers (scuttle-utils), 64
    * .subset2index (scuttle-utils), 64
    * .unpackLists (scuttle-utils), 64
  * [.outlier.filter (isOutlier), 31
    * addPerCellQC (addPerCellQCMetrics), 3
    * addPerCellQCMetrics, 3, 57
    * addPerFeatureQC (addPerCellQCMetrics), 3
    * addPerFeatureQCMetrics, 60
    * addPerFeatureQCMetrics (addPerCellQCMetrics), 3
    * aggregateAcrossCells, 4, 66, 71
    * aggregateAcrossCells, SingleCellExperiment-method (aggregateAcrossCells), 4
    * aggregateAcrossCells, SummarizedExperiment-method (aggregateAcrossCells), 4
    * aggregateAcrossFeatures, 7, 69
    * altExp, 22, 39
    * altExps, 5, 21, 22, 45
    * applySCE, 5, 6, 37
    * as.logical, 32
  * BiocParallelParam, 9, 28, 30, 34, 37, 47, 50, 52, 55, 59, 66, 68, 70
  * calculateAverage, 8, 20
    * calculateAverage, ANY-method (calculateAverage), 8
    * calculateAverage, SingleCellExperiment-method (calculateAverage), 8
    * calculateAverage, SummarizedExperiment-method (calculateAverage), 8
    * calculateCPM, 10, 12–14
    * calculateCPM, ANY-method (calculateCPM), 10
    * calculateCPM, SingleCellExperiment-method (calculateCPM), 10
    * calculateCPM, SummarizedExperiment-method (calculateCPM), 10
    * calculateFPKM, 12, 13
    * calculateTPM, 13
    * calculateTPM, ANY-method (calculateTPM), 13
    * calculateTPM, SingleCellExperiment-method (calculateTPM), 13
    * calculateTPM, SummarizedExperiment-method (calculateTPM), 13
    * cleanSizeFactors, 14, 19
    * colData, 3–6, 45, 50, 61, 66, 70, 71
    * colsum, 66
    * computeGeometricFactors (geometricSizeFactors), 29
    * computeLibraryFactors (librarySizeFactors), 34
    * computeMedianFactors (medianSizeFactors), 42
    * computePooledFactors, 16
    * computeSpikeFactors, 21
    * correctGroupSummary, 22
  * DataFrame, 5, 6, 50, 53, 55–57, 59, 61, 65, 70
  * DelayedArray, 30, 34
  * DelayedMatrix, 9, 27, 46
  * dgCMatrix, 27
  * downsampleBatches, 24
  * downsampleMatrix, 25, 26, 26, 48
  * downsampleReads, 27
INDEX

fitLinearModel, 28
geometricSizeFactors, 29, 31, 35, 44
geometricSizeFactors, ANY-method
  (geometricSizeFactors), 29
geometricSizeFactors, SummarizedExperiment-method
  (geometricSizeFactors), 29
isOutlier, 31, 53, 54, 61
librarySizeFactors, 10, 18, 20, 22, 30, 34, 44, 47, 48
librarySizeFactors, ANY-method
  (librarySizeFactors), 34
librarySizeFactors, SummarizedExperiment-method
  (librarySizeFactors), 34
List, 26
lm.fit, 29
logNormCounts, 10, 20, 31, 35, 44, 48
logNormCounts, SingleCellExperiment-method
  (logNormCounts), 35
logNormCounts, SummarizedExperiment-method
  (logNormCounts), 35
make.names, 40, 42
makePerCellDF, 38, 42
makePerFeatureDF, 40, 41
median, 6
medianSizeFactors, 20, 31, 35, 42
medianSizeFactors, ANY-method
  (medianSizeFactors), 42
medianSizeFactors, SummarizedExperiment-method
  (medianSizeFactors), 42
mockSCE, 44
modelGeneVarWithSpikes, 22, 47, 48
multiBatchNorm, 48
nls, 14, 15
nls.control, 14
normalizeCounts, 11, 27, 31, 35, 37, 38, 44, 45, 48
normalizeCounts, ANY-method
  (normalizeCounts), 45
normalizeCounts, SingleCellExperiment-method
  (normalizeCounts), 45
normalizeCounts, SummarizedExperiment-method
  (normalizeCounts), 45
numDetectedAcrossCells, ANY-method
  (numDetectedAcrossCells), 49
numDetectedAcrossCells, SummarizedExperiment-method
  (numDetectedAcrossCells), 49
numDetectedAcrossFeatures, 51, 69
numDetectedAcrossFeatures, ANY-method
  (numDetectedAcrossFeatures), 51
numDetectedAcrossFeatures, SummarizedExperiment-method
  (numDetectedAcrossFeatures), 51
outlier.filter (isOutlier), 31
outlier.filter-class (isOutlier), 31
perCellQCFilters, 52, 61, 62
perCellQCmetrics, 3, 33, 35, 54, 61, 62
perCellQCmetrics, ANY-method
  (perCellQCmetrics), 54
perCellQCmetrics, SingleCellExperiment-method
  (perCellQCmetrics), 54
perCellQCmetrics, SummarizedExperiment-method
  (perCellQCmetrics), 54
perFeatureQCmetrics, 3, 58
perFeatureQCmetrics, ANY-method
  (perFeatureQCmetrics), 58
perFeatureQCmetrics, SummarizedExperiment-method
  (perFeatureQCmetrics), 58
pooledSizeFactors, 14, 15
pooledSizeFactors
  (computePooledFactors), 16
pooledSizeFactors, ANY-method
  (computePooledFactors), 16
pooledSizeFactors, SummarizedExperiment-method
  (computePooledFactors), 16
quickCluster, 19, 20
quickPerCellQC, 33, 60
quickPerCellQC, ANY-method
  (quickPerCellQC), 60
quickPerCellQC, SummarizedExperiment-method
  (quickPerCellQC), 60
read.table, 63
readMM, 63
readSparseCounts, 62
RealizationSink, 27
reexports, 64
rowData, 3, 39
rowMeans, 43
rowsum, 68
rowSums, 9
<table>
<thead>
<tr>
<th>Function</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>scuttle-pkg</td>
<td>64</td>
</tr>
<tr>
<td>scuttle-utils</td>
<td>64</td>
</tr>
<tr>
<td>SingleCellExperiment</td>
<td>5, 6, 9–13, 16, 21, 30, 34, 35, 37–41, 43–46, 50, 54, 55, 58, 65</td>
</tr>
<tr>
<td>sizeFactors</td>
<td>10, 17, 22, 31, 35, 38, 43, 47</td>
</tr>
<tr>
<td>sum</td>
<td>6</td>
</tr>
<tr>
<td>sumCountsAcrossCells</td>
<td>50, 65</td>
</tr>
<tr>
<td>sumCountsAcrossCells,ANY-method</td>
<td>(sumCountsAcrossCells), 65</td>
</tr>
<tr>
<td>sumCountsAcrossCells,SummarizedExperiment-method</td>
<td>(sumCountsAcrossCells), 65</td>
</tr>
<tr>
<td>sumCountsAcrossFeatures</td>
<td>8, 52, 67</td>
</tr>
<tr>
<td>sumCountsAcrossFeatures,ANY-method</td>
<td>(sumCountsAcrossFeatures), 67</td>
</tr>
<tr>
<td>sumCountsAcrossFeatures,SummarizedExperiment-method</td>
<td>(sumCountsAcrossFeatures), 67</td>
</tr>
<tr>
<td>summarizeAssayByGroup</td>
<td>5–7, 23, 24, 49, 65, 69</td>
</tr>
<tr>
<td>summarizeAssayByGroup,ANY-method</td>
<td>(summarizeAssayByGroup), 69</td>
</tr>
<tr>
<td>summarizeAssayByGroup,SummarizedExperiment-method</td>
<td>(summarizeAssayByGroup), 69</td>
</tr>
<tr>
<td>SummarizedExperiment</td>
<td>3, 5–13, 16, 25, 30, 34, 37, 43, 46, 49, 51, 55, 58, 61, 65, 67, 70</td>
</tr>
<tr>
<td>uniquifyFeatureNames</td>
<td>71</td>
</tr>
<tr>
<td>whichNonZero</td>
<td>64</td>
</tr>
<tr>
<td>whichNonZero(reexports)</td>
<td>64</td>
</tr>
</tbody>
</table>