

Package ‘scran’

March 30, 2021

Version 1.18.5

Date 2021-02-03

Title Methods for Single-Cell RNA-Seq Data Analysis

Description Implements miscellaneous functions for interpretation of single-cell RNA-seq data. Methods are provided for assignment of cell cycle phase, detection of highly variable and significantly correlated genes, identification of marker genes, and other common tasks in routine single-cell analysis workflows.

Depends SingleCellExperiment

Imports SummarizedExperiment, S4Vectors, BiocGenerics, BiocParallel, Rcpp, stats, methods, utils, Matrix, scuttle, edgeR, limma, BiocNeighbors, igraph, statmod, DelayedArray, DelayedMatrixStats, BiocSingular, bluster, dqrng, beachmat

Suggests testthat, BiocStyle, knitr, HDF5Array, scRNAseq, dynamicTreeCut, DESeq2, monocle, Biobase, pheatmap, scater

biocViews ImmunoOncology, Normalization, Sequencing, RNASeq, Software, GeneExpression, Transcriptomics, SingleCell, Visualization, BatchEffect, Clustering

LinkingTo Rcpp, beachmat, BH, dqrng, scuttle

License GPL-3

NeedsCompilation yes

VignetteBuilder knitr

SystemRequirements C++11

RoxygenNote 7.1.1

git_url <https://git.bioconductor.org/packages/scran>

git_branch RELEASE_3_12

git_last_commit 1287359

git_last_commit_date 2021-02-04

Date/Publication 2021-03-29

Author Aaron Lun [aut, cre],
Karsten Bach [aut],
Jong Kyoung Kim [ctb],
Antonio Scialdone [ctb]

Maintainer Aaron Lun <infinite.monkeys.with.keyboards@gmail.com>

R topics documented:

.logBH	3
bootstrapCluster	4
buildSNNGraph	4
cleanSizeFactors	6
clusterModularity	8
clusterPurity	9
clusterSNNGraph	10
coassignProb	13
combineBlocks	14
combineMarkers	16
combinePValues	20
combineVar	22
computeSpikeFactors	24
computeSumFactors	25
convertTo	30
correlateGenes	32
correlateNull	33
correlatePairs	34
createClusterMST	38
cyclone	41
decideTestsPerLabel	44
defunct	45
denoisePCA	46
Distance-to-median	50
doubletCells	51
doubletCluster	54
doubletRecovery	56
findMarkers	59
fitTrendCV2	62
fitTrendPoisson	64
fitTrendVar	65
Gene selection	68
getClusteredPCs	69
getMarkerEffects	70
getTopHVGs	71
getTopMarkers	73
modelGeneCV2	75
modelGeneCV2WithSpikes	78
modelGeneVar	82
modelGeneVarByPoisson	85
modelGeneVarWithSpikes	89
multiMarkerStats	93
pairwiseBinom	94
pairwiseTTests	98
pairwiseWilcox	102
pseudoBulkDGE	106
pseudoBulkSpecific	109
quickCluster	112
quickPseudotime	115
quickSubCluster	117

<i>.logBH</i>	3
sandbag	119
scaledColRanks	121
summaryMarkerStats	122
testLinearModel	124
testPseudotime	126
Index	129

<i>.logBH</i>	<i>BH correction on log-p-values</i>
---------------	--------------------------------------

Description

Perform a Benjamini-Hochberg correction on log-transformed p-values to get log-adjusted p-values, without the loss of precision from undoing and redoing the log-transformations.

Usage

```
.logBH(log.p.val)
```

Arguments

log.p.val Numeric vector of log-transformed p-values.

Value

A numeric vector of the same length as log.p.val containing log-transformed BH-corrected p-values.

Author(s)

Aaron Lun

Examples

```
log.p.values <- log(runif(1000))
obs <- .logBH(log.p.values)
head(obs)

ref <- log(p.adjust(exp(log.p.values), method="BH"))
head(ref)
```

bootstrapCluster	<i>Assess cluster stability by bootstrapping</i>
------------------	--

Description

Generate bootstrap replicates and recluster on them to determine the stability of clusters with respect to sampling noise. This has been migrated to the **bluster** package as [bootstrapStability](#).

Usage

```
bootstrapCluster(...)
```

Arguments

... Arguments to pass to [bootstrapStability](#).

Value

See [bootstrapStability](#) for details.

Author(s)

Aaron Lun

See Also

[bootstrapStability](#), the new name for this function.

buildSNNGraph	<i>Build a nearest-neighbor graph</i>
---------------	---------------------------------------

Description

[SingleCellExperiment](#)-friendly wrapper around the [makeSNNGraph](#) and [makeKNNGraph](#) functions for creating nearest-neighbor graphs.

Usage

```
buildSNNGraph(x, ...)  
  
## S4 method for signature 'ANY'  
buildSNNGraph(  
  x,  
  ...,  
  d = 50,  
  transposed = FALSE,  
  subset.row = NULL,  
  BSPARAM = bsparam(),  
  BPPARAM = SerialParam()  
)
```

```

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

## S4 method for signature 'SingleCellExperiment'
buildSNNGraph(x, ..., use.dimred = NULL)

buildKNNGraph(x, ...)

## S4 method for signature 'ANY'
buildKNNGraph(
  x,
  ...,
  d = 50,
  transposed = FALSE,
  subset.row = NULL,
  BSPARAM = bsparam(),
  BPPARAM = SerialParam()
)

## S4 method for signature 'SingleCellExperiment'
buildKNNGraph(x, ..., use.dimred = NULL)

## S4 method for signature 'SingleCellExperiment'
buildKNNGraph(x, ..., use.dimred = NULL)

```

Arguments

x	A matrix-like object containing expression values for each gene (row) in each cell (column). These dimensions can be transposed if <code>transposed=TRUE</code> . Alternatively, a SummarizedExperiment or SingleCellExperiment containing such an expression matrix. If x is a SingleCellExperiment and <code>use.dimred</code> is set, its reducedDims will be used instead.
...	For the generics, additional arguments to pass to the specific methods. For the ANY methods, additional arguments to pass to makeSNNGraph or makeKNNGraph . For the SummarizedExperiment methods, additional arguments to pass to the corresponding ANY method. For the SingleCellExperiment methods, additional arguments to pass to the corresponding SummarizedExperiment method.
d	An integer scalar specifying the number of dimensions to use for a PCA on the expression matrix prior to the nearest neighbor search. Ignored for the ANY method if <code>transposed=TRUE</code> and for the SingleCellExperiment methods if <code>use.dimred</code> is set.
transposed	A logical scalar indicating whether x is transposed (i.e., rows are cells).
subset.row	See ?"scrn-gene-selection" . Only used when <code>transposed=FALSE</code> .
BSPARAM	A BiocSingularParam object specifying the algorithm to use for PCA, if d is not NA.
BPPARAM	A BiocParallelParam object to use for parallel processing.
assay.type	A string specifying which assay values to use.
use.dimred	A string specifying whether existing values in <code>reducedDims(x)</code> should be used.

Value

A [graph](#) where nodes are cells and edges represent connections between nearest neighbors, see [?makeSNNGraph](#) for more details.

Author(s)

Aaron Lun

See Also

[makeSNNGraph](#) and [makeKNNGraph](#), for the underlying functions that do the work.

See [cluster_walktrap](#) and related functions in **igraph** for clustering based on the produced graph.

Also see [findKNN](#) for specifics of the nearest-neighbor search.

Examples

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

g <- buildSNNGraph(sce)
clusters <- igraph::cluster_fast_greedy(g)$membership
table(clusters)

# Any clustering method from igraph can be used:
clusters <- igraph::cluster_walktrap(g)$membership
table(clusters)

# Smaller 'k' usually yields finer clusters:
g <- buildSNNGraph(sce, k=5)
clusters <- igraph::cluster_walktrap(g)$membership
table(clusters)

# Graph can be built off existing reducedDims results:
sce <- scater::runPCA(sce)
g <- buildSNNGraph(sce, use.dimred="PCA")
clusters <- igraph::cluster_fast_greedy(g)$membership
table(clusters)
```

cleanSizeFactors

Clean size factors

Description

Coerce non-positive size factors to positive values based on the number of detected features.

Usage

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

Arguments

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

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 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 `computeSumFactors`, 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

[computeSumFactors](#), which can occasionally generate negative size factors.

[nls](#), which performs the curve fitting.

Examples

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

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

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

clusterModularity	<i>Compute the cluster-wise modularity</i>
-------------------	--

Description

Calculate the modularity of each cluster from a graph, based on a null model of random connections between nodes. This function has been moved to the **bluster** package as [pairwiseModularity](#).

Usage

```
clusterModularity(...)
```

Arguments

... Arguments to pass to [pairwiseModularity](#).

Value

See [?pairwiseModularity](#) for more details.

Author(s)

Aaron Lun

See Also

[pairwiseModularity](#), the new name for this function.

clusterPurity	<i>Evaluate cluster purity</i>
---------------	--------------------------------

Description

Determine whether cells are surrounded by neighbors that are assigned to the same cluster. This function has now been deprecated in favor of [neighborPurity](#) from the **bluster** package.

Usage

```
clusterPurity(x, ...)

## S4 method for signature 'ANY'
clusterPurity(x, ..., transposed = FALSE, subset.row = NULL)

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

## S4 method for signature 'SingleCellExperiment'
clusterPurity(
  x,
  clusters = colLabels(x, onAbsence = "error"),
  ...,
  assay.type = "logcounts",
  use.dimred = NULL
)
```

Arguments

x	A matrix-like object containing expression values for each gene (row) in each cell (column). These dimensions can be transposed if <code>transposed=TRUE</code> . Alternatively, a SummarizedExperiment or SingleCellExperiment containing such an expression matrix. If x is a SingleCellExperiment and <code>use.dimred</code> is set, its reducedDims will be used instead.
...	For the generic, arguments to pass to specific methods. For the ANY method, arguments to pass to neighborPurity . For the SummarizedExperiment method, arguments to pass to the ANY method. For the SingleCellExperiment method, arguments to pass to the SummarizedExperiment method.
transposed	A logical scalar indicating whether x is transposed (i.e., rows are cells).
subset.row	See <code>?scran-gene-selection</code> . Only used when <code>transposed=FALSE</code> .
assay.type	A string specifying which assay values to use.
clusters	A vector or factor of cluster IDs to pass to neighborPurity .
use.dimred	A string specifying whether existing values in <code>reducedDims(x)</code> should be used.

Value

A [DataFrame](#) of purity statistics where each row corresponds to a cell in x, see `?neighborPurity` for details.

Author(s)

Aaron Lun

See Also[approxSilhouette](#), for another method of evaluating cluster separation.**Examples**

```

library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

g <- buildSNNGraph(sce)
clusters <- igraph::cluster_walktrap(g)$membership
out <- clusterPurity(sce, clusters)
boxplot(split(out$purity, clusters))

# Mocking up a stronger example:
ngenes <- 1000
centers <- matrix(rnorm(ngenes*3), ncol=3)
clusters <- sample(1:3, ncol(sce), replace=TRUE)

y <- centers[,clusters]
y <- y + rnorm(length(y))

out2 <- clusterPurity(y, clusters)
boxplot(split(out2$purity, clusters))

```

clusterSNNGraph

*Wrappers for graph-based clustering***Description**

Perform graph-based clustering using community detection methods on a nearest-neighbor graph, where nodes represent cells or k-means centroids. This has been deprecated in favor of directly using [clusterRows](#) from the **bluster** package, with BLUSPARAM set to [NNGraphParam\(\)](#) or [TwoStepParam\(\)](#).

Usage

```

clusterSNNGraph(x, ...)

## S4 method for signature 'ANY'
clusterSNNGraph(
  x,
  ...,
  clusterFUN = cluster_walktrap,
  subset.row = NULL,
  transposed = FALSE,
  use.kmeans = FALSE,
  kmeans.centers = NULL,

```

```

    kmeans.args = list(),
    full.stats = FALSE
  )

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

## S4 method for signature 'SingleCellExperiment'
clusterSNNGraph(x, ..., use.dimred = NULL)

clusterKNNGraph(x, ...)

## S4 method for signature 'ANY'
clusterKNNGraph(
  x,
  ...,
  clusterFUN = cluster_walktrap,
  subset.row = NULL,
  transposed = FALSE,
  use.kmeans = FALSE,
  kmeans.centers = NULL,
  kmeans.args = list(),
  full.stats = FALSE
)

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

## S4 method for signature 'SingleCellExperiment'
clusterKNNGraph(x, ..., use.dimred = NULL)

```

Arguments

x	A matrix-like object containing expression values for each gene (row) in each cell (column). These dimensions can be transposed if <code>transposed=TRUE</code> . Alternatively, a SummarizedExperiment or SingleCellExperiment containing such an expression matrix. If x is a SingleCellExperiment and <code>use.dimred</code> is set, its reducedDims will be used instead.
...	For the generics, additional arguments to pass to the specific methods. For the ANY methods, additional arguments to pass to buildSNNGraph or buildKNNGraph . For the SummarizedExperiment method, additional arguments to pass to the corresponding ANY method. For the SingleCellExperiment method, additional arguments to pass to the corresponding SummarizedExperiment method.
clusterFUN	Function that can take a graph object and return a communities object, see examples in the igraph package.
subset.row	See <code>?"scran-gene-selection"</code> . Only used when <code>transposed=FALSE</code> .
transposed	A logical scalar indicating whether x is transposed (i.e., rows are cells).
use.kmeans	Logical scalar indicating whether k-means clustering should be performed.
kmeans.centers	Integer scalar specifying the number of clusters to use for k-means clustering. Defaults to the square root of the number of cells in x.

<code>kmeans.args</code>	List containing additional named arguments to pass to kmeans .
<code>full.stats</code>	Logical scalar indicating whether to return more statistics regarding the k-means clustering.
<code>assay.type</code>	A string specifying which assay values to use.
<code>use.dimred</code>	A string specifying whether existing values in <code>reducedDims(x)</code> should be used.

Details

We suggest using the [clusterRows](#) functionality instead as it provides a more general interface to clustering.

- `clusterSNNGraph(x)` for a matrix-like object `x` is equivalent to `clusterRows(t(x), NNGraphParam())`.
- `clusterKNNGraph(x)` for a matrix-like object `x` is equivalent to `clusterRows(t(x), NNGraphParam(shared=FALSE))`.
- `clusterSNNGraph(x)` for a `SummarizedExperiment` object `x` is equivalent to `clusterRows(t(logcounts(x)), NNGraphParam())`.
- `clusterSNNGraph(x, use.dimred="PCA")` for a `SingleCellExperiment` object `x` is equivalent to `clusterRows(reducedDim(x, "PCA"), NNGraphParam())`.
- `clusterSNNGraph(x, use.kmeans=TRUE, use.dimred="PCA")` for a `SingleCellExperiment` object `x` is equivalent to `clusterRows(reducedDim(x, "PCA"), TwoStepParam())`.

Value

If `full.stats=FALSE`, a factor is returned containing the cluster assignment for each cell.

If `full.stats=TRUE` and `use.kmeans=TRUE`, a [DataFrame](#) is returned with one row per cell. This contains the columns `kmeans`, specifying the assignment of each cell to a k-means cluster; and `igraph`, specifying the assignment of each cell to a graph-based cluster operating on the k-means clusters. In addition, the `metadata` contains `graph`, a [graph](#) object where each node is a k-means cluster; and `membership`, the graph-based cluster to which each node is assigned.

Author(s)

Aaron Lun

See Also

[clusterRows](#) with `BLUSPARAM` set to an instance of [NNGraphParam](#) or [TwoStepParam](#).

Examples

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

clusters <- clusterSNNGraph(sce)
table(clusters)

# Can pass usual arguments to buildSNNGraph:
clusters2 <- clusterSNNGraph(sce, k=5)
table(clusters2)

# Works with low-dimensional inputs:
sce <- scater::runPCA(sce, ncomponents=10)
clusters3 <- clusterSNNGraph(sce, use.dimred="PCA")
table(clusters3)
```

```

# Turn on k-means for larger datasets, e.g.,
# assuming we already have a PCA result:
set.seed(101010)
bigpc <- matrix(rnorm(2000000), ncol=20)
clusters4 <- clusterSNNGraph(bigpc, d=NA, use.kmeans=TRUE, transposed=TRUE)
table(clusters4)

# Extract the graph for more details:
clusters5 <- clusterSNNGraph(sce, use.dimred="PCA",
  use.kmeans=TRUE, full.stats=TRUE)
head(clusters5)
metadata(clusters5)$graph

```

coassignProb

Compute coassignment probabilities

Description

Compute coassignment probabilities for each label in a reference grouping when compared to an alternative grouping of samples. This is now deprecated for [pairwiseRand](#).

Usage

```
coassignProb(ref, alt, summarize = FALSE)
```

Arguments

ref	A character vector or factor containing one set of groupings, considered to be the reference.
alt	A character vector or factor containing another set of groupings, to be compared to alt.
summarize	Logical scalar indicating whether the output matrix should be converted into a per-label summary.

Details

The coassignment probability for each pair of labels in `ref` is the probability that a randomly chosen cell from each of the two reference labels will have the same label in `alt`. High coassignment probabilities indicate that a particular pair of labels in `ref` are frequently assigned to the same label in `alt`, which has some implications for cluster stability.

When `summarize=TRUE`, we summarize the matrix of coassignment probabilities into a set of per-label values. The “self” coassignment probability is simply the diagonal entry of the matrix, i.e., the probability that two cells from the same label in `ref` also have the same label in `alt`. The “other” coassignment probability is the maximum probability across all pairs involving that label.

In general, `ref` is well-recapitulated by `alt` if the diagonal entries of the matrix is much higher than the sum of the off-diagonal entries. This manifests as higher values for the self probabilities compared to the other probabilities.

Note that the coassignment probability is closely related to the Rand index-based ratios broken down by cluster pair in [pairwiseRand](#) with `mode="ratio"` and `adjusted=FALSE`. The off-diagonal coassignment probabilities are simply 1 minus the off-diagonal ratio, while the on-diagonal values differ only by the lack of consideration of pairs of the same cell in [pairwiseRand](#).

Value

If `summarize=FALSE`, a numeric matrix is returned with upper triangular entries filled with the coassignment probabilities for each pair of labels in `ref`.

Otherwise, a [DataFrame](#) is returned with one row per label in `ref` containing the self and other coassignment probabilities.

Author(s)

Aaron Lun

See Also

[bootstrapCluster](#), to compute coassignment probabilities across bootstrap replicates.

[pairwiseRand](#), for another way to compare different clusterings.

Examples

```
library(scuttle)
sce <- mockSCE(ncells=200)
sce <- logNormCounts(sce)

clust1 <- kmeans(t(logcounts(sce)),3)$cluster
clust2 <- kmeans(t(logcounts(sce)),5)$cluster

coassignProb(clust1, clust2)
coassignProb(clust1, clust2, summarize=TRUE)
```

combineBlocks

Combine blockwise statistics

Description

Combine [DataFrames](#) of statistics computed separately for each block. This usually refers to feature-level statistics and sample-level blocks.

Usage

```
combineBlocks(
  blocks,
  ave.fields,
  pval.field,
  method,
  geometric,
  equiweight,
  weights,
  valid
)
```

Arguments

blocks	A list of DataFrames containing blockwise statistics. These should have the same number of rows and the same set of columns.
ave.fields	Character vector specifying the columns of blocks to be averaged. The value of each column is averaged across blocks, potentially in a weighted manner.
pval.field	String specifying the column of blocks containing the p-value. This is combined using combinePValues .
method	String specifying how p-values should be combined, see ?combinePValues .
geometric	Logical scalar indicating whether the geometric mean should be computed when averaging ave.fields.
equiweight	Logical scalar indicating whether each block should be given equal weight.
weights	Numeric vector of length equal to blocks, containing the weight for each block. Only used if equiweight=TRUE.
valid	Logical vector indicating whether each block is valid. Invalid blocks are still stored in the per.block output but are not used to compute the combined statistics.

Value

A [DataFrame](#) containing all fields in ave.fields and the p-values, where each column is created by combining the corresponding block-specific columns. A per.block column is also reported, containing a [DataFrame](#) of the [DataFrames](#) of blockwise statistics.

Author(s)

Aaron Lun

See Also

This function is used in [modelGeneVar](#) and friends, [combineVar](#) and [testLinearModel](#).

Examples

```
library(scuttle)
sce <- mockSCE()

y1 <- sce[,1:100]
y1 <- logNormCounts(y1) # normalize separately after subsetting.
results1 <- modelGeneVar(y1)

y2 <- sce[,1:100 + 100]
y2 <- logNormCounts(y2) # normalize separately after subsetting.
results2 <- modelGeneVar(y2)

# A manual implementation of combineVar:
combineBlocks(list(results1, results2),
  ave.fields=c("mean", "total", "bio", "tech"),
  pval.field='p.value',
  method='fisher',
  geometric=FALSE,
  equiweight=TRUE,
  weights=NULL,
```

```
valid=c(TRUE, TRUE))
```

```
combineMarkers
```

```
Combine pairwise DE results into a marker list
```

Description

Combine multiple pairwise differential expression comparisons between groups or clusters into a single ranked list of markers for each cluster.

Usage

```
combineMarkers(
  de.lists,
  pairs,
  pval.field = "p.value",
  effect.field = "logFC",
  pval.type = c("any", "some", "all"),
  min.prop = NULL,
  log.p.in = FALSE,
  log.p.out = log.p.in,
  output.field = NULL,
  full.stats = FALSE,
  sorted = TRUE,
  flatten = TRUE,
  BPPARAM = SerialParam()
)
```

Arguments

<code>de.lists</code>	A list-like object where each element is a <code>data.frame</code> or DataFrame . Each element should represent the results of a pairwise comparison between two groups/clusters, in which each row should contain the statistics for a single gene/feature. Rows should be named by the feature name in the same order for all elements.
<code>pairs</code>	A matrix, <code>data.frame</code> or DataFrame with two columns and number of rows equal to the length of <code>de.lists</code> . Each row should specify the pair of clusters being compared for the corresponding element of <code>de.lists</code> .
<code>pval.field</code>	A string specifying the column name of each element of <code>de.lists</code> that contains the p-value.
<code>effect.field</code>	A string specifying the column name of each element of <code>de.lists</code> that contains the effect size. If <code>NULL</code> , effect sizes are not reported in the output.
<code>pval.type</code>	A string specifying how p-values are to be combined across pairwise comparisons for a given group/cluster.
<code>min.prop</code>	Numeric scalar specifying the minimum proportion of significant comparisons per gene, Defaults to 0.5 when <code>pval.type="some"</code> , otherwise defaults to zero.
<code>log.p.in</code>	A logical scalar indicating if the p-values in <code>de.lists</code> were log-transformed.
<code>log.p.out</code>	A logical scalar indicating if log-transformed p-values/FDRs should be returned.

<code>output.field</code>	A string specifying the prefix of the field names containing the effect sizes. Defaults to "stats" if <code>full.stats=TRUE</code> , otherwise it is set to <code>effect.field</code> .
<code>full.stats</code>	A logical scalar indicating whether all statistics in <code>de.lists</code> should be stored in the output for each pairwise comparison.
<code>sorted</code>	Logical scalar indicating whether each output DataFrame should be sorted by a statistic relevant to <code>pval.type</code> .
<code>flatten</code>	Logical scalar indicating whether the individual effect sizes should be flattened in the output DataFrame. If <code>FALSE</code> , effect sizes are reported as a nested matrix for easier programmatic use.
<code>BPPARAM</code>	A BiocParallelParam object indicating whether and how parallelization should be performed across genes.

Details

An obvious strategy to characterizing differences between clusters is to look for genes that are differentially expressed (DE) between them. However, this entails a number of comparisons between all pairs of clusters to comprehensively identify genes that define each cluster. For all pairwise comparisons involving a single cluster, we would like to consolidate the DE results into a single list of candidate marker genes. Doing so is the purpose of the `combineMarkers` function.

DE statistics from any testing regime can be supplied to this function - see the Examples for how this is done with t-tests from [pairwiseTTests](#). The effect size field in the output will vary according to the type of input statistics, for example:

- `logFC.Y` from [pairwiseTTests](#), containing log-fold changes in mean expression (usually in base 2).
- `AUC.Y` from [pairwiseWilcox](#), containing the area under the curve, i.e., the concordance probability.
- `logFC.Y` from [pairwiseBinom](#), containing log2-fold changes in the expressing proportion.

Value

A named [List](#) of [DataFrames](#) where each DataFrame contains the consolidated marker statistics for each gene (row) for the cluster of the same name. The DataFrame for cluster X contains the fields:

Top: Integer, the minimum rank across all pairwise comparisons. This is only reported if `pval.type="any"`.

p.value: Numeric, the combined p-value across all comparisons if `log.p.out=FALSE`.

FDR: Numeric, the BH-adjusted p-value for each gene if `log.p.out=FALSE`.

log.p.value: Numeric, the (natural) log-transformed version p-value. Replaces the `p.value` field if `log.p.out=TRUE`.

log.FDR: Numeric, the (natural) log-transformed adjusted p-value. Replaces the `FDR` field if `log.p.out=TRUE`.

summary.<OUTPUT>: Numeric, named by replacing `<OUTPUT>` with `output.field`. This contains the summary effect size, obtained by combining effect sizes from all pairwise comparison into a single value. Only reported when `effect.field` is not `NULL`.

<OUTPUT>.Y: Comparison-specific statistics, named by replacing `<OUTPUT>` with `output.field`. One of these fields is present for every other cluster Y in `clusters` and contains statistics for the comparison of X to Y . If `full.stats=FALSE`, each field is numeric and contains the effect size of the comparison of X over Y . Otherwise, each field is a nested DataFrame containing the full statistics for that comparison (i.e., the same as the corresponding entry of `de.lists`). Only reported if `flatten=FALSE` and (for `full.stats=FALSE`) if `effect.field` is not `NULL`.

each.<OUTPUT>: A nested DataFrame of comparison-specific statistics, named by replacing <OUTPUT> with `output.field`. If `full.stats=FALSE`, one column is present for every other cluster Y in `clusters` and contains the effect size of the comparison of X to Y . Otherwise, each column contains another nested DataFrame containing the full set of statistics for that comparison. Only reported if `flatten=FALSE` and (for `full.stats=FALSE`) if `effect.field` is not NULL.

Consolidating with DE against any other cluster

By default, each DataFrame is sorted by the Top value when `pval.type="any"`. Taking all rows with Top values less than or equal to T yields a marker set containing the top T genes (ranked by significance) from each pairwise comparison. This guarantees the inclusion of genes that can distinguish between any two clusters.

To demonstrate, let us define a marker set with an T of 1 for a given cluster. The set of genes with $\text{Top} \leq 1$ will contain the top gene from each pairwise comparison to every other cluster. If T is instead, say, 5, the set will consist of the *union* of the top 5 genes from each pairwise comparison. Obviously, multiple genes can have the same Top as different genes may have the same rank across different pairwise comparisons. Conversely, the marker set may be smaller than the product of Top and the number of other clusters, as the same gene may be shared across different comparisons.

This approach does not explicitly favour genes that are uniquely expressed in a cluster. Rather, it focuses on combinations of genes that - together - drive separation of a cluster from the others. This is more general and robust but tends to yield a less focused marker set compared to the other `pval.type` settings.

For each gene and cluster, the summary effect size is defined as the effect size from the pairwise comparison with the lowest p-value. The combined p-value is computed by applying Simes' method to all p-values. Neither of these values are directly used for ranking and are only reported for the sake of the user.

Consolidating with DE against all other clusters

If `pval.type="all"`, the null hypothesis is that the gene is not DE in all contrasts. A combined p-value for each gene is computed using Berger's intersection union test (IUT). Ranking based on the IUT p-value will focus on genes that are DE in that cluster compared to *all* other clusters. This strategy is particularly effective when dealing with distinct clusters that have a unique expression profile. In such cases, it yields a highly focused marker set that concisely captures the differences between clusters.

However, it can be too stringent if the cluster's separation is driven by combinations of gene expression. For example, consider a situation involving four clusters expressing each combination of two marker genes A and B. With `pval.type="all"`, neither A nor B would be detected as markers as it is not uniquely defined in any one cluster. This is especially detrimental with overclustering where an otherwise acceptable marker is discarded if it is not DE between two adjacent clusters.

For each gene and cluster, the summary effect size is defined as the effect size from the pairwise comparison with the *largest* p-value. This reflects the fact that, with this approach, a gene is only as significant as its weakest DE. Again, this value is not directly used for ranking and are only reported for the sake of the user.

Consolidating with DE against some other clusters

The `pval.type="some"` setting serves as a compromise between "all" and "any". A combined p-value is calculated by taking the middlemost value of the Holm-corrected p-values for each gene. (By default, this the median for odd numbers of contrasts and one-after-the-median for even numbers, but the exact proportion can be changed by setting `min.prop` - see [?combinePValues](#).) Here, the null hypothesis is that the gene is not DE in at least half of the contrasts.

Genes are then ranked by the combined p-value. The aim is to provide a more focused marker set without being overly stringent, though obviously it loses the theoretical guarantees of the more extreme settings. For example, there is no guarantee that the top set contains genes that can distinguish a cluster from any other cluster, which would have been possible with `pval.type="any"`.

For each gene and cluster, the summary effect size is defined as the effect size from the pairwise comparison with the `min.prop`-smallest p-value. This mirrors the p-value calculation but, again, is reported only for the benefit of the user.

Consolidating against some other clusters, rank-style

A slightly different flavor of the “some cluster” approach is achieved by setting `method="any"` with `min.prop` set to some positive value in (0, 1). A gene will only be high-ranked if it is among the top-ranked genes in at least `min.prop` of the pairwise comparisons. For example, if `min.prop=0.3`, any gene with a value of `Top` less than or equal to 5 will be in the top 5 DEGs of at least 30

This method increases the stringency of the “any” setting in a safer manner than `pval.type="some"`. Specifically, we avoid comparing p-values across pairwise comparisons, which can be problematic if there are power differences across comparisons, e.g., due to differences in the number of cells across the other clusters.

Note that the value of `min.prop` does not affect the combined p-value and summary effect size calculations for `pval.type="any"`.

Correcting for multiple testing

The BH method is then applied on the consolidated p-values across all genes to obtain the FDR field. The reported FDRs are intended only as a rough measure of significance. Properly correcting for multiple testing is not generally possible when `clusters` is determined from the same `x` used for DE testing.

If `log.p=TRUE`, log-transformed p-values and FDRs will be reported. This may be useful in overpowered studies with many cells, where directly reporting the raw p-values would result in many zeroes due to the limits of machine precision.

Ordering of the output

- Within each DataFrame, if `sorted=TRUE`, genes are ranked by the `Top` column if available and the `p.value` (or `log.p.value`) if not. Otherwise, the input order of the genes is preserved.
- For the DataFrame corresponding to cluster X , the `<OUTPUT>.Y` columns are sorted according to the order of cluster IDs in `pairs[,2]` for all rows where `pairs[,1]` is X .
- In the output List, the DataFrames themselves are sorted according to the order of cluster IDs in `pairs[,1]`. Note that DataFrames are only created for clusters present in `pairs[,1]`. Clusters unique to `pairs[,2]` will only be present within a DataFrame as Y .

Author(s)

Aaron Lun

References

- Simes RJ (1986). An improved Bonferroni procedure for multiple tests of significance. *Biometrika* 73:751-754.
- Berger RL and Hsu JC (1996). Bioequivalence trials, intersection-union tests and equivalence confidence sets. *Statist. Sci.* 11, 283-319.

See Also

[pairwiseTTests](#) and [pairwiseWilcox](#), for functions that can generate de.lists and pairs.
[findMarkers](#), which automatically performs combineMarkers on the t-test or Wilcoxon test results.

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Any clustering method is okay.
kout <- kmeans(t(logcounts(sce)), centers=3)
clusters <- paste0("Cluster", kout$cluster)

out <- pairwiseTTests(logcounts(sce), groups=clusters)
comb <- combineMarkers(out$statistics, out$pairs)
comb[["Cluster1"]]

out <- pairwiseWilcox(logcounts(sce), groups=clusters)
comb <- combineMarkers(out$statistics, out$pairs, effect.field="AUC")
comb[["Cluster2"]]

out <- pairwiseBinom(logcounts(sce), groups=clusters)
comb <- combineMarkers(out$statistics, out$pairs)
comb[["Cluster3"]]
```

 combinePValues

Combine p-values

Description

Combine p-values from independent or dependent hypothesis tests using a variety of meta-analysis methods.

Usage

```
combinePValues(
  ...,
  method = c("fisher", "z", "simes", "berger", "holm-middle"),
  weights = NULL,
  log.p = FALSE,
  min.prop = 0.5
)
```

Arguments

... Two or more numeric vectors of p-values of the same length.
 method A string specifying the combining strategy to use.

weights	A numeric vector of positive weights, with one value per vector in Alternatively, a list of numeric vectors of weights, with one vector per element in This is only used when method="z".
log.p	Logical scalar indicating whether the p-values in . . . are log-transformed.
min.prop	Numeric scalar in [0, 1] specifying the minimum proportion of tests to reject for each set of p-values when method="holm-middle".

Details

This function will operate across elements on . . . in parallel to combine p-values. That is, the set of first p-values from all vectors will be combined, followed by the second p-values and so on. This is useful for combining p-values for each gene across different hypothesis tests.

Fisher's method, Stouffer's Z method and Simes' method test the global null hypothesis that all of the individual null hypotheses in the set are true. The global null is rejected if any of the individual nulls are rejected. However, each test has different characteristics:

- Fisher's method requires independence of the test statistic. It is useful in asymmetric scenarios, i.e., when the null is only rejected in one of the tests in the set. Thus, a low p-value in any test is sufficient to obtain a low combined p-value.
- Stouffer's Z method require independence of the test statistic. It favours symmetric rejection and is less sensitive to a single low p-value, requiring more consistently low p-values to yield a low combined p-value. It can also accommodate weighting of the different p-values.
- Simes' method technically requires independence but tends to be quite robust to dependencies between tests. See Sarkar and Chung (1997) for details, as well as work on the related Benjamini-Hochberg method. It favours asymmetric rejection and is less powerful than the other two methods under independence.

Berger's intersection-union test examines a different global null hypothesis - that at least one of the individual null hypotheses are true. Rejection in the IUT indicates that all of the individual nulls have been rejected. This is the statistically rigorous equivalent of a naive intersection operation.

In the Holm-middle approach, the global null hypothesis is that more than $1 - \text{min.prop}$ proportion of the individual nulls in the set are true. We apply the Holm-Bonferroni correction to all p-values in the set and take the $\text{ceiling}(\text{min.prop} * N)$ -th smallest value where N is the size of the set (excluding NA values). This method works correctly in the presence of correlations between p-values.

Value

A numeric vector containing the combined p-values.

Author(s)

Aaron Lun

References

- Fisher, R.A. (1925). *Statistical Methods for Research Workers*. Oliver and Boyd (Edinburgh).
- Whitlock MC (2005). Combining probability from independent tests: the weighted Z-method is superior to Fisher's approach. *J. Evol. Biol.* 18, 5:1368-73.
- Simes RJ (1986). An improved Bonferroni procedure for multiple tests of significance. *Biometrika* 73:751-754.

Berger RL and Hsu JC (1996). Bioequivalence trials, intersection-union tests and equivalence confidence sets. *Statist. Sci.* 11, 283-319.

Sarkar SK and Chung CK (1997). The Simes method for multiple hypothesis testing with positively dependent test statistics. *J. Am. Stat. Assoc.* 92, 1601-1608.

Examples

```
p1 <- runif(10000)
p2 <- runif(10000)
p3 <- runif(10000)

fish <- combinePValues(p1, p2, p3)
hist(fish)

z <- combinePValues(p1, p2, p3, method="z", weights=1:3)
hist(z)

simes <- combinePValues(p1, p2, p3, method="simes")
hist(simes)

berger <- combinePValues(p1, p2, p3, method="berger")
hist(berger)
```

combineVar

Combine variance decompositions

Description

Combine the results of multiple variance decompositions, usually generated for the same genes across separate batches of cells.

Usage

```
combineVar(
  ...,
  method = "fisher",
  pval.field = "p.value",
  other.fields = NULL,
  equiweight = TRUE,
  ncells = NULL
)

combineCV2(
  ...,
  method = "fisher",
  pval.field = "p.value",
  other.fields = NULL,
  equiweight = TRUE,
  ncells = NULL
)
```

Arguments

<code>...</code>	Two or more DataFrames of variance modelling results. For <code>combineVar</code> , these should be produced by modelGeneVar or modelGeneVarWithSpikes . For <code>combineCV2</code> , these should be produced by modelGeneCV2 or modelGeneCV2WithSpikes . Alternatively, one or more lists of DataFrames containing variance modelling results. Mixed inputs are also acceptable, e.g., lists of DataFrames alongside the DataFrames themselves.
<code>method</code>	String specifying how p-values are to be combined, see combinePValues for options.
<code>pval.field</code>	A string specifying the column name of each element of <code>...</code> that contains the p-value.
<code>other.fields</code>	A character vector specifying the fields containing other statistics to combine.
<code>equiweight</code>	Logical scalar indicating whether each result is to be given equal weight in the combined statistics.
<code>ncells</code>	Numeric vector containing the number of cells used to generate each element of <code>...</code> . Only used if <code>equiweight=FALSE</code> .

Details

These functions are designed to merge results from separate calls to [modelGeneVar](#), [modelGeneCV2](#) or related functions, where each result is usually computed for a different batch of cells. Separate variance decompositions are necessary in cases where the mean-variance relationships vary across batches (e.g., different concentrations of spike-in have been added to the cells in each batch), which precludes the use of a common trend fit. By combining these results into a single set of statistics, we can apply standard strategies for feature selection in multi-batch integrated analyses.

By default, statistics in `other.fields` contain all common non-numeric fields that are not `pval.field` or "FDR". This usually includes "mean", "total", "bio" (for `combineVar`) or "ratio" (for `combineCV2`).

- For `combineVar`, statistics are combined by averaging them across all input [DataFrames](#).
- For `combineCV2`, statistics are combined by taking the geometric mean across all inputs.

This difference between functions reflects the method by which the relevant measure of overdispersion is computed. For example, "bio" is computed by subtraction, so taking the average bio remains consistent with subtraction of the total and technical averages. Similarly, "ratio" is computed by division, so the combined ratio is consistent with division of the geometric means of the total and trend values.

If `equiweight=FALSE`, each per-batch statistic is weighted by the number of cells used to compute it. The number of cells can be explicitly set using `ncells`, and is otherwise assumed to be equal for all batches. No weighting is performed by default, which ensures that all batches contribute equally to the combined statistics and avoids situations where batches with many cells dominate the output.

The [combinePValues](#) function is used to combine p-values across batches. Only `method="z"` will perform any weighting of batches, and only if `weights` is set.

Value

A [DataFrame](#) with the same numeric fields as that produced by [modelGeneVar](#) or [modelGeneCV2](#). Each row corresponds to an input gene. Each field contains the (weighted) arithmetic/geometric mean across all batches except for `p.value`, which contains the combined p-value based on `method`; and `FDR`, which contains the adjusted p-value using the BH method.

Author(s)

Aaron Lun

See Also

[modelGeneVar](#) and [modelGeneCV2](#), for two possible inputs into this function.
[combinePValues](#), for details on how the p-values are combined.

Examples

```
library(scuttle)
sce <- mockSCE()

y1 <- sce[,1:100]
y1 <- logNormCounts(y1) # normalize separately after subsetting.
results1 <- modelGeneVar(y1)

y2 <- sce[,1:100 + 100]
y2 <- logNormCounts(y2) # normalize separately after subsetting.
results2 <- modelGeneVar(y2)

head(combineVar(results1, results2))
head(combineVar(results1, results2, method="simes"))
head(combineVar(results1, results2, method="berger"))
```

computeSpikeFactors *Normalization with spike-in counts*

Description

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

Usage

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

Arguments

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

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

Lun ATL, McCarthy DJ and Marioni JC (2016). A step-by-step workflow for low-level analysis of single-cell RNA-seq data with Bioconductor. *F1000Res*. 5:2122

See Also

`altExps`, for the concept of alternative experiments.

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

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- computeSpikeFactors(sce, "Spikes")
summary(sizeFactors(sce))
```

computeSumFactors

Normalization by deconvolution

Description

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

Usage

```

calculateSumFactors(x, ...)

## S4 method for signature 'ANY'
calculateSumFactors(
  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 = SerialParam()
)

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

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

```

Arguments

x	For calculateSumFactors, a numeric matrix-like object of counts, where rows are genes and columns are cells. Alternatively, a SummarizedExperiment object containing such a matrix. For computeSumFactors, a SingleCellExperiment object containing a count matrix.
...	For the calculateSumFactors generic, additional arguments to pass to each method. For the SummarizedExperiment method, additional methods to pass to the ANY method. For the computeSumFactors function, additional arguments to pass to calculateSumFactors.
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.
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	See ?"scran-gene-selection".
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 `calculateSumFactors`, a numeric vector of size factors for all cells in `x` is returned.

For `computeSumFactors`, an object of class `x` is returned containing the vector of size factors in `sizeFactors(x)`.

Overview of the deconvolution method

The `computeSumFactors` function implements the deconvolution strategy (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 over-determined 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 `computeSumFactors` 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>). 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.

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. A convenience function `quickCluster` is provided for this purpose, though any reasonable clustering can be used. Only a rough clustering is required here, as `computeSumFactors` is robust to a moderate level of DE within each cluster.

Size factors computed within each cluster must be rescaled for comparison between clusters. This is done by normalizing between the per-cluster pseudo-cells to identify the rescaling factor. One cluster is chosen 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).

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 estimate of each factor. 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, `computeSumFactors` will degrade to library size normalization.

Dealing with negative size factors

It is possible for the deconvolution algorithm to yield negative estimates for the size factors. These values are obviously nonsensical and `computeSumFactors` 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 negative size factors, the best solution is to increase the stringency of the filtering.

- If only a few negative 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 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 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 negative 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, `computeSumFactors` is robust to some level of DE genes - that is, after all, its *raison d'être* - so one should only explicitly remove DE genes if it is convenient to do so.

Obtaining standard errors

Previous versions of `computeSumFactors` 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

[quickCluster](#), to obtain a rough clustering for use in clusters.

[logNormCounts](#), which uses the computed size factors to compute normalized expression values.

Examples

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

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

# Using pre-clustering.
preclusters <- quickCluster(sce)
table(preclusters)

sce2 <- computeSumFactors(sce, clusters=preclusters)
head(sizeFactors(sce2))
```

convertTo

Convert to other classes

Description

Convert a [SingleCellExperiment](#) object into other classes for entry into other analysis pipelines.

Usage

```
convertTo(
  x,
  type = c("edgeR", "DESeq2", "monocle"),
  ...,
  assay.type = 1,
  subset.row = NULL
)
```

Arguments

x	A SingleCellExperiment object.
type	A string specifying the analysis for which the object should be prepared.
...	Other arguments to be passed to pipeline-specific constructors.
assay.type	A string specifying which assay of x should be put in the returned object.
subset.row	See ?" scran-gene-selection ".

Details

This function converts an `SingleCellExperiment` object into various other classes in preparation for entry into other analysis pipelines, as specified by type.

Value

For type="edgeR", a `DGEList` object is returned containing the count matrix. Size factors are converted to normalization factors. Gene-specific `rowData` is stored in the `genes` element, and cell-specific `colData` is stored in the `samples` element.

For type="DESeq2", a `DESeqDataSet` object is returned containing the count matrix and size factors. Additional gene- and cell-specific data is stored in the `mcols` and `colData` respectively.

For type="monocle", a `CellDataSet` object is returned containing the count matrix and size factors. Additional gene- and cell-specific data is stored in the `featureData` and `phenoData` respectively.

Author(s)

Aaron Lun

See Also

[DGEList](#), [DESeqDataSetFromMatrix](#), [newCellDataSet](#), for specific class constructors.

Examples

```
library(scuttle)
sce <- mockSCE()

# Adding some additional embellishments.
sizeFactors(sce) <- 2^rnorm(ncol(sce))
rowData(sce)$SYMBOL <- paste0("X", seq_len(nrow(sce)))
sce$other <- sample(LETTERS, ncol(sce), replace=TRUE)

# Converting to various objects.
convertTo(sce, type="edgeR")
convertTo(sce, type="DESeq2")
convertTo(sce, type="monocle")
```

correlateGenes	<i>Per-gene correlation statistics</i>
----------------	--

Description

Compute per-gene correlation statistics by combining results from gene pair correlations.

Usage

```
correlateGenes(stats)
```

Arguments

`stats` A [DataFrame](#) of pairwise correlation statistics, returned by [correlatePairs](#).

Details

For each gene, all of its pairs are identified and the corresponding p-values are combined using Simes' method. This tests whether the gene is involved in significant correlations to *any* other gene. Per-gene statistics are useful for identifying correlated genes without regard to what they are correlated with (e.g., during feature selection).

Value

A [DataFrame](#) with one row per unique gene in `stats` and containing the fields:

`gene`: A field of the same type as `stats$gene1` specifying the gene identity.

`rho`: Numeric, the correlation with the largest magnitude across all gene pairs involving the corresponding gene.

`p.value`: Numeric, the Simes p-value for this gene.

`FDR`: Numeric, the adjusted p.value across all rows.

`limited`: Logical, indicates whether the combined p-value is at its lower bound.

Author(s)

Aaron Lun

References

Simes RJ (1986). An improved Bonferroni procedure for multiple tests of significance. *Biometrika* 73:751-754.

See Also

[correlatePairs](#), to compute `stats`.

Examples

```

library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)
pairs <- correlatePairs(sce, iters=1e5, subset.row=1:100)

g.out <- correlateGenes(pairs)
head(g.out)

```

correlateNull *Build null correlations*

Description

Build a distribution of correlations under the null hypothesis of independent expression between pairs of genes.

Usage

```

correlateNull(
  ncells,
  iters = 1e+06,
  block = NULL,
  design = NULL,
  equiweight = TRUE,
  BPPARAM = SerialParam()
)

```

Arguments

ncells	An integer scalar indicating the number of cells in the data set.
iters	An integer scalar specifying the number of values in the null distribution.
block	A factor specifying the blocking level for each cell.
design	A numeric design matrix containing uninteresting factors to be ignored.
equiweight	A logical scalar indicating whether statistics from each block should be given equal weight. Otherwise, each block is weighted according to its number of cells. Only used if block is specified.
BPPARAM	A BiocParallelParam object that specifies the manner of parallel processing to use.

Details

The `correlateNull` function constructs an empirical null distribution for Spearman's rank correlation when it is computed with `ncells` cells. This is done by shuffling the ranks, calculating the correlation and repeating until `iters` values are obtained. No consideration is given to tied ranks, which has implications for the accuracy of p-values in [correlatePairs](#).

If `block` is specified, a null correlation is created within each level of `block` using the shuffled ranks. The final correlation is then defined as the average of the per-level correlations, weighted by

the number of cells in that level if `equiweight=FALSE`. Levels with fewer than 3 cells are ignored, and if no level has 3 or more cells, all returned correlations will be NA.

If `design` is specified, the same process is performed on ranks derived from simulated residuals computed by fitting the linear model to a vector of normally distributed values. If there are not at least 3 residual d.f., all returned correlations will be NA. The `design` argument cannot be used at the same time as `block`.

Value

A numeric vector of length `iters` is returned containing the sorted correlations under the null hypothesis of no correlations.

Author(s)

Aaron Lun

See Also

[correlatePairs](#), where the null distribution is used to compute p-values.

Examples

```
set.seed(0)
ncells <- 100

# Simplest case:
null.dist <- correlateNull(ncells, iters=10000)
hist(null.dist)

# With a blocking factor:
block <- sample(LETTERS[1:3], ncells, replace=TRUE)
null.dist <- correlateNull(block=block, iters=10000)
hist(null.dist)

# With a design matrix.
cov <- runif(ncells)
X <- model.matrix(~cov)
null.dist <- correlateNull(design=X, iters=10000)
hist(null.dist)
```

correlatePairs

Test for significant correlations

Description

Identify pairs of genes that are significantly correlated in their expression profiles, based on Spearman's rank correlation.

Usage

```

correlatePairs(x, ...)

## S4 method for signature 'ANY'
correlatePairs(
  x,
  null.dist = NULL,
  ties.method = c("expected", "average"),
  iters = 1e+06,
  block = NULL,
  design = NULL,
  equiweight = TRUE,
  use.names = TRUE,
  subset.row = NULL,
  pairings = NULL,
  BPPARAM = SerialParam()
)

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

```

Arguments

x	A numeric matrix-like object of log-normalized expression values, where rows are genes and columns are cells. Alternatively, a SummarizedExperiment object containing such a matrix.
...	For the generic, additional arguments to pass to specific methods. For the SummarizedExperiment method, additional methods to pass to the ANY method.
null.dist	A numeric vector of rho values under the null hypothesis.
ties.method	String specifying how tied ranks should be handled.
iters	Integer scalar specifying the number of iterations to use in correlateNull to build the null distribution.
block	A factor specifying the blocking level for each cell in x. If specified, correlations are computed separately within each block and statistics are combined across blocks.
design	A numeric design matrix containing uninteresting factors to be ignored.
equiweight	A logical scalar indicating whether statistics from each block should be given equal weight. Otherwise, each block is weighted according to its number of cells. Only used if block is specified.
use.names	A logical scalar specifying whether the row names of x should be used in the output. Alternatively, a character vector containing the names to use.
subset.row	See ?"scran-gene-selection" .
pairings	A NULL value indicating that all pairwise correlations should be computed; or a list of 2 vectors of genes between which correlations are to be computed; or a integer/character matrix with 2 columns of specific gene pairs - see below for details.
BPPARAM	A BiocParallelParam object that specifies the manner of parallel processing to use.
assay.type	A string specifying which assay values to use.

Details

The `correlatePairs` function identifies significant correlations between all pairs of genes in `x`. This allows prioritization of genes that are driving systematic substructure in the data set. By definition, such genes should be correlated as they are behaving in the same manner across cells. In contrast, genes driven by random noise should not exhibit any correlations with other genes.

We use Spearman's rho to quantify correlations robustly based on ranked gene expression. To identify correlated gene pairs, the significance of non-zero correlations is assessed using a permutation test. The null hypothesis is that the ranking of normalized expression across cells should be independent between genes. This allows us to construct a null distribution by randomizing the ranks within each gene. A pre-computed empirical distribution can be supplied as `null.dist`, which saves some time by avoiding repeated calls to `correlateNull`.

The p-value for each gene pair is defined as the tail probability of this distribution at the observed correlation (with some adjustment to avoid zero p-values). Correction for multiple testing is done using the BH method. The lower bound of the p-values is determined by the value of `iters`. If the `limited` field is TRUE in the returned dataframe, it may be possible to obtain lower p-values by increasing `iters`. This should be examined for non-significant pairs, in case some correlations are overlooked due to computational limitations. The function will automatically raise a warning if any genes are limited in their significance at a FDR of 5%.

For the `SingleCellExperiment` method, correlations should be computed for normalized expression values in the specified `assay.type`.

Value

A `DataFrame` is returned with one row per gene pair and the following fields:

`gene1`, `gene2`: Character or integer fields specifying the genes in the pair. If `use.names=FALSE`, integers are returned representing row indices of `x`, otherwise gene names are returned.

`rho`: A numeric field containing the approximate Spearman's rho.

`p.value`, `FDR`: Numeric fields containing the permutation p-value and its BH-corrected equivalent.

`limited`: A logical scalar indicating whether the p-value is at its lower bound, defined by `iters`.

Rows are sorted by increasing `p.value` and, if tied, decreasing absolute size of `rho`. The exception is if `subset.row` is a matrix, in which case each row in the dataframe correspond to a row of `subset.row`.

Accounting for uninteresting variation

If the experiment has known (and uninteresting) factors of variation, these can be included in `design` or `block`. `correlatePairs` will then attempt to ensure that these factors do not drive strong correlations between genes. Examples might be to block on batch effects or cell cycle phase, which may have substantial but uninteresting effects on expression.

The approach used to remove these factors depends on whether `design` or `block` is used. If there is only one factor, e.g., for plate or animal of origin, `block` should be used. Each level of the factor is defined as a separate group of cells. For each pair of genes, correlations are computed within each group, and a weighted mean based on the group size) of the correlations is taken across all groups. The same strategy is used to generate the null distribution where ranks are computed and shuffled within each group.

For experiments containing multiple factors or covariates, a design matrix should be passed into `design`. The `correlatePairs` function will fit a linear model to the (log-normalized) expression values. The correlation between a pair of genes is then computed from the residuals of the fitted model. Similarly, to obtain a null distribution of rho values, normally-distributed random errors are

simulated in a fitted model based on design; the corresponding residuals are generated from these errors; and the correlation between sets of residuals is computed at each iteration.

We recommend using `block` wherever possible. While `design` can also be used for one-way layouts, this is not ideal as it assumes normality of errors and deals poorly with ties. Specifically, zero counts within or across groups may no longer be tied when converted to residuals, potentially resulting in spuriously large correlations.

If any level of `block` has fewer than 3 cells, it is ignored. If all levels of `block` have fewer than 3 cells, all output statistics are set to NA. Similarly, if `design` has fewer than 3 residual d.f., all output statistics are set to NA.

Gene selection

The `pairings` argument specifies the pairs of genes to compute correlations for:

- By default, correlations will be computed between all pairs of genes with `pairings=NULL`. Genes that occur earlier in `x` are labelled as `gene1` in the output DataFrame. Redundant permutations are not reported.
- If `pairings` is a list of two vectors, correlations will be computed between one gene in the first vector and another gene in the second vector. This improves efficiency if the only correlations of interest are those between two pre-defined sets of genes. Genes in the first vector are always reported as `gene1`.
- If `pairings` is an integer/character matrix of two columns, each row is assumed to specify a gene pair. Correlations will then be computed for only those gene pairs, and the returned dataframe will *not* be sorted by p-value. Genes in the first column of the matrix are always reported as `gene1`.

If `subset.row` is not NULL, only the genes in the selected subset are used to compute correlations - see `?"scran-gene-selection"`. This will interact properly with `pairings`, such that genes in `pairings` and not in `subset.row` will be ignored.

We recommend setting `subset.row` and/or `pairings` to contain only the subset of genes of interest. This reduces computational time and memory usage by only computing statistics for the gene pairs of interest. For example, we could select only HVGs to focus on genes contributing to cell-to-cell heterogeneity (and thus more likely to be involved in driving substructure). There is no need to account for HVG pre-selection in multiple testing, because rank correlations are unaffected by the variance.

Handling tied values

By default, `ties.method="expected"` which uses the expectation of the pairwise correlation from randomly broken ties. Imagine obtaining unique ranks by randomly breaking ties in expression values, e.g., by addition of some very small random jitter. The reported value of the correlation is simply the expected value across all possible permutations of broken ties.

With `ties.method="average"`, each set of tied observations is assigned the average rank across all tied values. This yields the standard value of Spearman's rho as computed by `cor`.

We use the expected rho by default as avoids inflated correlations in the presence of many ties. This is especially relevant for single-cell data containing many zeroes that remain tied after scaling normalization. An extreme example is that of two genes that are only expressed in the same cell, for which the standard rho is 1 while the expected rho is close to zero.

Note that the p-value calculations are not accurate in the presence of ties, as tied ranks are not considered by `correlationNull`. With `ties.method="expected"`, the p-values are probably a bit conservative. With `ties.method="average"`, they will be very anticonservative.

Author(s)

Aaron Lun

References

Lun ATL (2019). Some thoughts on testing for correlations. <https://l1tla.github.io/SingleCellThoughts/software/correlations/corsim.html>

Phipson B and Smyth GK (2010). Permutation P-values should never be zero: calculating exact P-values when permutations are randomly drawn. *Stat. Appl. Genet. Mol. Biol.* 9:Article 39.

See Also

Compare to [cor](#) for the standard Spearman's calculation.

Use [correlateGenes](#) to get per-gene correlation statistics.

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Basic pairwise application (turning down iters for speed).
out <- correlatePairs(sce, subset.row=1:100, iters=1e5)
head(out)

# Computing between specific subsets of genes:
out <- correlatePairs(sce, pairings=list(1:10, 110:120), iters=1e5)
head(out)

# Computing between specific pairs:
out <- correlatePairs(sce, pairings=rbind(c(1,10), c(2, 50)), iters=1e5)
head(out)
```

`createClusterMST`*Minimum spanning trees on cluster centroids*

Description

Perform basic trajectory analyses with minimum spanning trees (MST) computed on cluster centroids, based on the methodology in the **TSCAN** package. These functions are now deprecated as they have been moved to the **TSCAN** package itself.

Usage

```
createClusterMST(centers, outgroup = FALSE, outscale = 3)
```

```
connectClusterMST(centers, mst, combined = TRUE)
```

```
orderClusterMST(x, ids, centers, mst, start = NULL)
```

Arguments

centers	A numeric matrix of cluster centroids where each <i>row</i> represents a cluster and each column represents a dimension (usually a PC or another low-dimensional embedding). Each row should be named with the cluster name.
outgroup	A logical scalar indicating whether an outgroup should be inserted to split unrelated trajectories. Alternatively, a numeric scalar specifying the distance threshold to use for this splitting.
outscale	A numeric scalar specifying the scaling to apply to the median distance between centroids to define the threshold for outgroup splitting. Only used if outgroup=TRUE.
mst	A graph object containing a MST, typically the output of createClusterMST(centers). For connectClusterMSTNodes, the MST may be computed from a different centers.
combined	Logical scalar indicating whether a single data.frame of edge coordinates should be returned.
x	A numeric matrix of per-cell coordinates where each <i>row</i> represents a cell and each column represents a dimension (again, usually a low-dimensional embedding).
ids	A character vector of length equal to the number of cells, specifying the cluster to which each cell is assigned.
start	A character vector specifying the starting node from which to compute pseudotimes in each component of mst. Defaults to an arbitrarily chosen node of degree 1 or lower in each component.

Details

These functions represent some basic utilities for a simple trajectory analysis based on the algorithm in the **TSCAN** package.

createClusterMST builds a MST where each node is a cluster centroid and each edge is weighted by the Euclidean distance between centroids. This represents the most parsimonious explanation for a particular trajectory and has the advantage of being directly interpretable with respect to any pre-existing clusters.

connectClusterMST provides the coordinates of the start and end of every edge. This is mostly useful for plotting purposes in [segments](#) or the equivalent [ggplot2](#) functionality. We suggest using [aggregateAcrossCells](#) to obtain centers for multiple low-dimensional results at once.

orderClusterMST will map each cell to the closest edge involving the cluster to which it is assigned. (Here, edges are segments terminated by their nodes, so some cells may simply be mapped to the edge terminus.) It will then calculate the distance of that cell along the MST from the starting node specified by start. This distance represents the pseudotime for that cell and can be used in further quantitative analyses.

Value

createClusterMST returns a [graph](#) object containing an MST computed on centers.

connectClusterMST returns, by default, a data.frame containing the start and end coordinates of segments representing all the edges in mst. If combined=FALSE, a list of two data.frames is returned where corresponding rows represent the start and end coordinates of the same edge.

orderClusterMST returns a numeric matrix containing the pseudotimes of all cells (rows) across all paths (columns) through mst.

Introducing an outgroup

If `outgroup=TRUE`, we add an outgroup to avoid constructing a trajectory between “unrelated” clusters. This is done by adding an extra row/column to the distance matrix corresponding to an artificial outgroup cluster, where the distance to all of the other real clusters is set to $\omega/2$. Large jumps in the MST between real clusters that are more distant than ω will then be rerouted through the outgroup, allowing us to break up the MST into multiple subcomponents by removing the outgroup.

The default ω value is computed by constructing the MST from the original distance matrix, computing the median edge length in that MST, and then scaling it by `outscale`. This adapts to the magnitude of the distances and the internal structure of the dataset while also providing some margin for variation across cluster pairs. Alternatively, `outgroup` can be set to a numeric scalar in which case it is used directly as ω .

Confidence on the edges

For the MST, we obtain a measure of the confidence in each edge by computing the distance gained if that edge were not present. Ambiguous parts of the tree will be less penalized from deletion of an edge, manifesting as a small distance gain. In contrast, parts of the tree with clear structure will receive a large distance gain upon deletion of an obvious edge.

For each edge, we divide the distance gain by the length of the edge to normalize for cluster resolution. This avoids overly penalizing edges in parts of the tree involving broad clusters while still retaining sensitivity to detect distance gain in overclustered regions. As an example, a normalized gain of unity for a particular edge means that its removal requires an alternative path that increases the distance travelled by that edge’s length.

The normalized gain is reported as the “`gain`” attribute in the edges of the MST from `createClusterMST`. Note that the “`weight`” attribute represents the edge length.

Interpreting the pseudotime matrix

The pseudotimes are returned as a matrix where each row corresponds to cell in `x` and each column corresponds to a path through the MST from `start` to all nodes of degree 1. (If `start` is itself a node of degree 1, then paths are only considered to all other such nodes.) This format is inspired by that from the `slingshot` package and provides a compact representation of branching events.

Each branching event in the MST results in a new path and thus a new column in the pseudotime matrix. For any given row in this matrix, entries are either NA or they are identical. This reflects the fact that multiple paths will share a section of the MST for which the pseudotimes are the same.

The starting node in `start` is *completely arbitrarily chosen* by `orderClusterMST`, as directionality is impossible to infer from the expression matrix alone. However, it is often possible to use prior biological knowledge to pick an appropriate cluster as the starting node.

Author(s)

Aaron Lun

References

Ji Z and Ji H (2016). TSCAN: Pseudo-time reconstruction and evaluation in single-cell RNA-seq analysis. *Nucleic Acids Res.* 44, e117

See Also

[quickPseudotime](#), a wrapper to quickly perform these calculations.

Examples

```

# Mocking up a Y-shaped trajectory.
centers <- rbind(c(0,0), c(0, -1), c(1, 1), c(-1, 1))
rownames(centers) <- seq_len(nrow(centers))
clusters <- sample(nrow(centers), 1000, replace=TRUE)
cells <- centers[clusters,]
cells <- cells + rnorm(length(cells), sd=0.5)

# Creating the MST first:
mst <- createClusterMST(centers)
plot(mst)

# Also plotting the MST on top of existing visualizations:
edges <- connectClusterMST(centers, mst, combined=FALSE)
plot(cells[,1], cells[,2], col=clusters)
segments(edges$start$dim1, edges$start$dim2, edges$end$dim1,
         edges$end$dim2, lwd=5)

# Finally, obtaining pseudo-time orderings.
ordering <- orderClusterMST(cells, clusters, centers, mst)
unified <- rowMeans(ordering, na.rm=TRUE)
plot(cells[,1], cells[,2], col=topo.colors(21)[cut(unified, 21)], pch=16)

```

cyclone

Cell cycle phase classification

Description

Classify single cells into their cell cycle phases based on gene expression data.

Usage

```

cyclone(x, ...)

## S4 method for signature 'ANY'
cyclone(
  x,
  pairs,
  gene.names = rownames(x),
  iter = 1000,
  min.iter = 100,
  min.pairs = 50,
  BPPARAM = SerialParam(),
  verbose = FALSE,
  subset.row = NULL
)

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

```

Arguments

x	A numeric matrix-like object of gene expression values where rows are genes and columns are cells. Alternatively, a SummarizedExperiment object containing such a matrix.
...	For the generic, additional arguments to pass to specific methods. For the <code>SummarizedExperiment</code> method, additional arguments to pass to the <code>ANY</code> method.
pairs	A list of data.frames produced by sandbag , containing pairs of marker genes.
gene.names	A character vector of gene names, with one value per row in x.
iter	An integer scalar specifying the number of iterations for random sampling to obtain a cycle score.
min.iter	An integer scalar specifying the minimum number of iterations for score estimation.
min.pairs	An integer scalar specifying the minimum number of pairs for cycle estimation.
BPPARAM	A BiocParallelParam object to use for parallel processing across cells.
verbose	A logical scalar specifying whether diagnostics should be printed to screen.
subset.row	See ?" scrn-gene-selection ".
assay.type	A string specifying which assay values to use, e.g., "counts" or "logcounts".

Details

This function implements the classification step of the pair-based prediction method described by Scialdone et al. (2015). To illustrate, consider classification of cells into G1 phase. Pairs of marker genes are identified with [sandbag](#), where the expression of the first gene in the training data is greater than the second in G1 phase but less than the second in all other phases. For each cell, `cyclone` calculates the proportion of all marker pairs where the expression of the first gene is greater than the second in the new data x (pairs with the same expression are ignored). A high proportion suggests that the cell is likely to belong in G1 phase, as the expression ranking in the new data is consistent with that in the training data.

Proportions are not directly comparable between phases due to the use of different sets of gene pairs for each phase. Instead, proportions are converted into scores (see below) that account for the size and precision of the proportion estimate. The same process is repeated for all phases, using the corresponding set of marker pairs in `pairs`. Cells with G1 or G2M scores above 0.5 are assigned to the G1 or G2M phases, respectively. (If both are above 0.5, the higher score is used for assignment.) Cells can be assigned to S phase based on the S score, but a more reliable approach is to define S phase cells as those with G1 and G2M scores below 0.5.

Pre-trained classifiers are provided for mouse and human datasets, see ?[sandbag](#) for more details. However, note that the classifier may not be accurate for data that are substantially different from those used in the training set, e.g., due to the use of a different protocol. In such cases, users can construct a custom classifier from their own training data using the [sandbag](#) function. This is usually necessary for other model organisms where pre-trained classifiers are not available.

Users should *not* filter out low-abundance genes before applying `cyclone`. Even if a gene is not expressed in any cell, it may still be useful for classification if it is phase-specific. Its lack of expression relative to other genes will still yield informative pairs, and filtering them out would reduce power.

Value

A list is returned containing:

phases: A character vector containing the predicted phase for each cell.

scores: A data frame containing the numeric phase scores for each phase and cell (i.e., each row is a cell).

normalized.scores: A data frame containing the row-normalized scores (i.e., where the row sum for each cell is equal to 1).

Description of the score calculation

To make the proportions comparable between phases, a distribution of proportions is constructed by shuffling the expression values within each cell and recalculating the proportion. The phase score is defined as the lower tail probability at the observed proportion. High scores indicate that the proportion is greater than what is expected by chance if the expression of marker genes were independent (i.e., with no cycle-induced correlations between marker pairs within each cell).

By default, shuffling is performed `iter` times to obtain the distribution from which the score is estimated. However, some iterations may not be used if there are fewer than `min.pairs` pairs with different expression, such that the proportion cannot be calculated precisely. A score is only returned if the distribution is large enough for stable calculation of the tail probability, i.e., consists of results from at least `min.iter` iterations.

Note that the score calculation in `cyclone` is slightly different from that described originally by Scialdone et al. The original code shuffles all expression values within each cell, while in this implementation, only the expression values of genes in the marker pairs are shuffled. This modification aims to use the most relevant expression values to build the null score distribution.

Author(s)

Antonio Scialdone, with modifications by Aaron Lun

References

Scialdone A, Natarajana KN, Saraiva LR et al. (2015). Computational assignment of cell-cycle stage from single-cell transcriptome data. *Methods* 85:54–61

See Also

[sandbag](#), to generate the pairs from reference data.

Examples

```
set.seed(1000)
library(scuttle)
sce <- mockSCE(ncells=200, ngenes=1000)

# Constructing a classifier:
is.G1 <- which(sce$Cell_Cycle %in% c("G1", "G0"))
is.S <- which(sce$Cell_Cycle=="S")
is.G2M <- which(sce$Cell_Cycle=="G2M")
out <- sandbag(sce, list(G1=is.G1, S=is.S, G2M=is.G2M))

# Classifying a new dataset:
test <- mockSCE(ncells=50)
```

```
assignments <- cyclone(test, out)
head(assignments$scores)
table(assignments$phases)
```

decideTestsPerLabel *Decide tests for each label*

Description

Decide which tests (i.e., genes) are significant for differential expression between conditions in each label, using the output of `pseudoBulkDGE`. This mimics the `decideTests` functionality from `limma`.

Usage

```
decideTestsPerLabel(
  results,
  method = c("separate", "global"),
  threshold = 0.05,
  pval.field = NULL,
  lfc.field = "logFC"
)

summarizeTestsPerLabel(results, ...)
```

Arguments

<code>results</code>	A List containing the output of <code>pseudoBulkDGE</code> . Each entry should be a <code>DataFrame</code> with the same number and order of rows, containing at least a numeric "PValue" column (and usually a "logFC" column). For <code>summarizeTestsPerLabel</code> , this may also be a matrix produced by <code>decideTestsPerLabel</code> .
<code>method</code>	String specifying whether the Benjamini-Hochberg correction should be applied across all clusters or separately within each label.
<code>threshold</code>	Numeric scalar specifying the FDR threshold to consider genes as significant.
<code>pval.field</code>	String containing the name of the column containing the p-value in each entry of <code>results</code> . Defaults to "PValue", "P.Value" or "p.value" based on fields in the first entry of <code>results</code> .
<code>lfc.field</code>	String containing the name of the column containing the log-fold change. Ignored if the column is not available. Defaults to "logFC" if this field is available.
<code>...</code>	Further arguments to pass to <code>decideTestsPerLabel</code> if <code>results</code> is a <code>List</code> .

Details

If a log-fold change field is available and specified in `lfc.field`, values of 1, -1 and 0 indicate that the gene is significantly upregulated, downregulated or not significant, respectively. Note, the interpretation of "up" and "down" depends on the design and contrast in `pseudoBulkDGE`.

Otherwise, if no log-fold change is available or if `lfc.field=NULL`, values of 1 or 0 indicate that a gene is significantly DE or not, respectively.

NA values indicate either that the relevant gene was low-abundance for a particular label and filtered out, or that the DE comparison for that label was not possible (e.g., no residual d.f.).

Value

For `decideTestsPerLabel`, an integer matrix indicating whether each gene (row) is significantly DE between conditions for each label (column).

For `summarizeTestsPerLabel`, an integer matrix containing the number of genes of each DE status (column) in each label (row).

Author(s)

Aaron Lun

See Also

[pseudoBulkDGE](#), which generates the input to this function.

[decideTests](#), which inspired this function.

Examples

```
example(pseudoBulkDGE)
head(decideTestsPerLabel(out))
summarizeTestsPerLabel(out)
```

defunct

Defunct functions

Description

Functions that have passed on to the function afterlife. Their successors are also listed.

Usage

`trendVar(...)`

`decomposeVar(...)`

`testVar(...)`

`improvedCV2(...)`

`technicalCV2(...)`

`makeTechTrend(...)`

`multiBlockVar(...)`

`multiBlockNorm(...)`

`overlapExprs(...)`

`parallelPCA(...)`

Arguments

... Ignored arguments.

Details

trendVar, decomposeVar and testVar are succeeded by a suite of funtions related to [modelGeneVar](#) and [fitTrendVar](#).

improvedCV2 and technicalCV2 are succeeded by [modelGeneCV2](#) and [fitTrendCV2](#).

makeTechTrend is succeeded by [modelGeneVarByPoisson](#).

multiBlockVar is succeeded by the block argument in many of the modelling functions, and multiBlockNorm is no longer necessary.

[overlapExprs](#) is succeeded by [findMarkers](#) with test.type="wilcox".

[parallelPCA](#) has been moved over to the **PCAtools** package.

Value

All functions error out with a defunct message pointing towards its descendent (if available).

Author(s)

Aaron Lun

Examples

```
try(trendVar())
```

denoisePCA

Denoise expression with PCA

Description

Denoise log-expression data by removing principal components corresponding to technical noise.

Usage

```
getDenoisedPCs(x, ...)
```

```
## S4 method for signature 'ANY'
```

```
getDenoisedPCs(
  x,
  technical,
  subset.row = NULL,
  min.rank = 5,
  max.rank = 50,
  fill.missing = FALSE,
  BSPARAM = bsparam(),
  BPPARAM = SerialParam()
)
```

```
## S4 method for signature 'SummarizedExperiment'
```

```

getDenoisedPCs(x, ..., assay.type = "logcounts")

denoisePCA(x, ..., value = c("pca", "lowrank"), assay.type = "logcounts")

denoisePCANumber(var.exp, var.tech, var.total)

```

Arguments

<code>x</code>	For <code>getDenoisedPCs</code> , a numeric matrix of log-expression values, where rows are genes and columns are cells. Alternatively, a SummarizedExperiment object containing such a matrix. For <code>denoisePCA</code> , a SingleCellExperiment object containing a log-expression matrix.
<code>...</code>	For the <code>getDenoisedPCs</code> generic, further arguments to pass to specific methods. For the <code>SingleCellExperiment</code> method, further arguments to pass to the ANY method. For the <code>denoisePCA</code> function, further arguments to pass to the <code>getDenoisedPCs</code> function.
<code>technical</code>	An object containing the technical components of variation for each gene in <code>x</code> . This can be: <ul style="list-style-type: none"> • a function that computes the technical component of the variance for a gene with a given mean log-expression, as generated by <code>fitTrendVar</code>. • a numeric vector of length equal to the number of rows in <code>x</code>, containing the technical component for each gene. • a DataFrame of variance decomposition results generated by <code>modelGeneVarWithSpikes</code> or related functions.
<code>subset.row</code>	See <code>?"scrnan-gene-selection"</code> .
<code>min.rank</code> , <code>max.rank</code>	Integer scalars specifying the minimum and maximum number of PCs to retain.
<code>fill.missing</code>	Logical scalar indicating whether entries in the rotation matrix should be imputed for genes that were not used in the PCA.
<code>BSPARAM</code>	A BiocSingularParam object specifying the algorithm to use for PCA.
<code>BPPARAM</code>	A BiocParallelParam object to use for parallel processing.
<code>assay.type</code>	A string specifying which assay values to use.
<code>value</code>	String specifying the type of value to return. <code>"pca"</code> will return the PCs, <code>"n"</code> will return the number of retained components, and <code>"lowrank"</code> will return a low-rank approximation.
<code>var.exp</code>	A numeric vector of the variances explained by successive PCs, starting from the first (but not necessarily containing all PCs).
<code>var.tech</code>	A numeric scalar containing the variance attributable to technical noise.
<code>var.total</code>	A numeric scalar containing the total variance in the data.

Details

This function performs a principal components analysis to eliminate random technical noise in the data. Random noise is uncorrelated across genes and should be captured by later PCs, as the variance in the data explained by any single gene is low. In contrast, biological processes should be captured by earlier PCs as more variance can be explained by the correlated behavior of sets

of genes in a particular pathway. The idea is to discard later PCs to remove noise and improve resolution of population structure. This also has the benefit of reducing computational work for downstream steps.

The choice of the number of PCs to discard is based on the estimates of technical variance in technical. This argument accepts a number of different values, depending on how the technical noise is calculated - this generally involves functions such as `modelGeneVarWithSpikes` or `modelGeneVarByPoisson`. The percentage of variance explained by technical noise is estimated by summing the technical components across genes and dividing by the summed total variance. Genes with negative biological components are ignored during downstream analyses to ensure that the total variance is greater than the overall technical estimate.

Now, consider the retention of the first d PCs. For a given value of d , we compute the variance explained by all of the later PCs. We aim to find the smallest value of d such that the sum of variances explained by the later PCs is still less than the variance attributable to technical noise. This choice of d represents a lower bound on the number of PCs that can be retained before biological variation is definitely lost. We use this value to obtain a “reasonable” dimensionality for the PCA output.

Note that d will be coerced to lie between `min.rank` and `max.rank`. This mitigates the effect of occasional extreme results when the percentage of noise is very high or low.

Value

For `getDenoisedPCs`, a list is returned containing:

- `components`, a numeric matrix containing the selected PCs (columns) for all cells (rows). This has number of columns between `min.rank` and `max.rank` inclusive.
- `rotation`, a numeric matrix containing rotation vectors (columns) for all genes (rows). This has number of columns between `min.rank` and `max.rank` inclusive.
- `percent.var`, a numeric vector containing the percentage of variance explained by the first `max.rank` PCs.

`denoisePCA` will return a modified `x` with:

- the PC results stored in the `reducedDims` as a “PCA” entry, if `type="pca"`.
- a low-rank approximation stored as a new “lowrank” assay, if `type="lowrank"`.

`denoisePCANumber` will return an integer scalar specifying the number of PCs to retain. This is equivalent to the output from `getDenoisedPCs` after setting `value="n"`, but ignoring any setting of `min.rank` or `max.rank`.

Effects of gene selection

We can use `subset.row` to perform the PCA on a subset of genes, e.g., HVGs. This can be used to only perform the PCA on genes with the largest biological components, thus increasing the signal-to-noise ratio of downstream analyses. Note that only rows with positive components are actually used in the PCA, even if we explicitly specified them in `subset.row`.

If `fill.missing=TRUE`, entries of the rotation matrix are imputed for all genes in `x`. This includes “unselected” genes, i.e., with negative biological components or that were not selected with `subset.row`. Rotation vectors are extrapolated to these genes by projecting their expression profiles into the low-dimensional space defined by the SVD on the selected genes. This is useful for guaranteeing that any low-rank approximation has the same dimensions as the input `x`. For example, `denoisePCA` will only ever use `fill.missing=TRUE` when `value="lowrank"`.

Caveats with interpretation

The function's choice of d is only optimal if the early PCs capture all the biological variation with minimal noise. This is unlikely to be true as the PCA cannot distinguish between technical noise and weak biological signal in the later PCs. In practice, the chosen d can only be treated as a lower bound for the retention of signal, and it is debatable whether this has any particular relation to the “best” choice of the number of PCs. For example, many aspects of biological variation are not that interesting (e.g., transcriptional bursting, metabolic fluctuations) and it is often the case that we do not need to retain this signal, in which case the chosen d - despite being a lower bound - may actually be higher than necessary.

Interpretation of the choice of d is even more complex if technical was generated with `modelGeneVar` rather than `modelGeneVarWithSpikes` or `modelGeneVarByPoisson`. The former includes “uninteresting” biological variation in its technical component estimates, increasing the proportion of variance attributed to technical noise and yielding a lower value of d . Indeed, use of results from `modelGeneVar` often results in d being set to `min.rank`, which can be problematic if secondary factors of biological variation are discarded.

Author(s)

Aaron Lun

References

Lun ATL (2018). Discussion of PC selection methods for scRNA-seq data. <https://github.com/LTLA/PCSelection2018>

See Also

`modelGeneVarWithSpikes` and `modelGeneVarByPoisson`, for methods of computing technical components.

`runSVD`, for the underlying SVD algorithm(s).

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Modelling the variance:
var.stats <- modelGeneVar(sce)

# Denoising:
pcs <- getDenoisedPCs(sce, technical=var.stats)
head(pcs$components)
head(pcs$rotation)
head(pcs$percent.var)

# Automatically storing the results.
sce <- denoisePCA(sce, technical=var.stats)
reducedDimNames(sce)
```

Distance-to-median *Compute the distance-to-median statistic*

Description

Compute the distance-to-median statistic for the CV2 residuals of all genes

Usage

```
DM(mean, cv2, win.size=51)
```

Arguments

mean	A numeric vector of average counts for each gene.
cv2	A numeric vector of squared coefficients of variation for each gene.
win.size	An integer scalar specifying the window size for median-based smoothing. This should be odd or will be incremented by 1.

Details

This function will compute the distance-to-median (DM) statistic described by Kolodziejczyk et al. (2015). Briefly, a median-based trend is fitted to the log-transformed cv2 against the log-transformed mean using `runmed`. The DM is defined as the residual from the trend for each gene. This statistic is a measure of the relative variability of each gene, after accounting for the empirical mean-variance relationship. Highly variable genes can then be identified as those with high DM values.

Value

A numeric vector of DM statistics for all genes.

Author(s)

Jong Kyoung Kim, with modifications by Aaron Lun

References

Kolodziejczyk AA, Kim JK, Tsang JCH et al. (2015). Single cell RNA-sequencing of pluripotent states unlocks modular transcriptional variation. *Cell Stem Cell* 17(4), 471–85.

Examples

```
# Mocking up some data
ngenes <- 1000
ncells <- 100
gene.means <- 2^runif(ngenes, 0, 10)
dispersions <- 1/gene.means + 0.2
counts <- matrix(rnbinom(ngenes*ncells, mu=gene.means, size=1/dispersions), nrow=ngenes)

# Computing the DM.
means <- rowMeans(counts)
cv2 <- apply(counts, 1, var)/means^2
```

```
dm.stat <- DM(means, cv2)
head(dm.stat)
```

doubletCells	<i>Detect doublet cells</i>
--------------	-----------------------------

Description

Identify potential doublet cells based on simulations of putative doublet expression profiles. This function is now deprecated, use `computeDoubletDensity` from **scDbFinder** instead.

Usage

```
doubletCells(x, ...)

## S4 method for signature 'ANY'
doubletCells(
  x,
  size.factors.norm = NULL,
  size.factors.content = NULL,
  k = 50,
  subset.row = NULL,
  niters = max(10000, ncol(x)),
  block = 10000,
  d = 50,
  force.match = FALSE,
  force.k = 20,
  force.ndist = 3,
  BNPARAM = KmknnParam(),
  BSPARAM = bsparam(),
  BPPARAM = SerialParam()
)

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

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

Arguments

<code>x</code>	A numeric matrix-like object of count values, where each column corresponds to a cell and each row corresponds to an endogenous gene. Alternatively, a SummarizedExperiment or SingleCellExperiment object containing such a matrix.
<code>...</code>	For the generic, additional arguments to pass to specific methods. For the <code>SummarizedExperiment</code> and <code>SingleCellExperiment</code> methods, additional arguments to pass to the ANY method.

<code>size.factors.norm</code>	A numeric vector of size factors for normalization of <code>x</code> prior to PCA and distance calculations. If NULL, defaults to size factors derived from the library sizes of <code>x</code> . For the <code>SingleCellExperiment</code> method, the default values are taken from <code>sizeFactors(x)</code> , if they are available.
<code>size.factors.content</code>	A numeric vector of size factors for RNA content normalization of <code>x</code> prior to simulating doublets. This is orthogonal to the values in <code>size.factors.norm</code> , see <code>Details</code> .
<code>k</code>	An integer scalar specifying the number of nearest neighbours to use to determine the bandwidth for density calculations.
<code>subset.row</code>	See <code>?"scran-gene-selection"</code> .
<code>niters</code>	An integer scalar specifying how many simulated doublets should be generated.
<code>block</code>	An integer scalar controlling the rate of doublet generation, to keep memory usage low.
<code>d</code>	An integer scalar specifying the number of components to retain after the PCA.
<code>force.match</code>	A logical scalar indicating whether remapping of simulated doublets to original cells should be performed.
<code>force.k</code>	An integer scalar specifying the number of neighbours to use for remapping if <code>force.match=TRUE</code> .
<code>force.ndist</code>	A numeric scalar specifying the bandwidth for remapping if <code>force.match=TRUE</code> .
<code>BNPARAM</code>	A <code>BiocNeighborParam</code> object specifying the nearest neighbor algorithm. This should be an algorithm supported by <code>findNeighbors</code> .
<code>BSPARAM</code>	A <code>BiocSingularParam</code> object specifying the algorithm to use for PCA, if <code>d</code> is not NA.
<code>BPPARAM</code>	A <code>BiocParallelParam</code> object specifying whether the neighbour searches should be parallelized.
<code>assay.type</code>	A string specifying which assay values contain the count matrix.

Details

This function simulates doublets by adding the count vectors for two randomly chosen cells in `x`. For each original cell, we compute the density of neighboring simulated doublets and compare it to the density of neighboring original cells. Genuine doublets should have a high density of simulated doublets relative to the density of its neighbourhood. Thus, the doublet score for each cell is defined as the ratio of densities of simulated doublets to the (squared) density of the original cells.

Densities are calculated in low-dimensional space after a PCA on the log-normalized expression matrix of `x`. Simulated doublets are projected into the low-dimensional space using the rotation vectors computed from the original cells. A tricube kernel is used to compute the density around each cell. The bandwidth of the kernel is set to the median distance to the `k` nearest neighbour across all cells.

The two size factor arguments have different roles:

- `size.factors.norm` contains the size factors to be used for normalization prior to PCA and distance calculations. This defaults to the values returned by `librarySizeFactors` but can be explicitly set to ensure that the low-dimensional space is consistent with that in the rest of the analysis.

- `size.factors.content` is much more important, and represents the size factors that preserve RNA content differences. This is usually computed from spike-in RNA and ensures that the simulated doublets have the correct ratio of contributions from the original cells.

It is possible to set both of these arguments as they are orthogonal to each other. Setting `size.factors.content` will not affect the calculation of log-normalized expression values from `x`. Conversely, setting `size.factors.norm` will not affect the ratio in which cells are added together when simulating doublets.

If `force.match=TRUE`, simulated doublets will be remapped to the nearest neighbours in the original data. This is done by taking the (tricube-weighted) average of the PC scores for the `force.k` nearest neighbors. The tricube bandwidth for remapping is chosen by taking the median distance and multiplying it by `force.ndist`, to protect against later neighbours that might be outliers. The aim is to adjust for unknown differences in RNA content that would cause the simulated doublets to be systematically displaced from their true locations. However, it may also result in spuriously high scores for single cells that happen to be close to a cluster of simulated doublets.

Value

A numeric vector of doublet scores for each cell in `x`.

Author(s)

Aaron Lun

References

Lun ATL (2018). Detecting doublet cells with *scran*. https://l1tla.github.io/SingleCellThoughts/software/doublet_detection/bycell.html

See Also

`doubletCluster`, to detect doublet clusters.

Examples

```
# Mocking up an example.
set.seed(100)
ngenes <- 1000
mu1 <- 2^rnorm(ngenes)
mu2 <- 2^rnorm(ngenes)
mu3 <- 2^rnorm(ngenes)
mu4 <- 2^rnorm(ngenes)

counts.1 <- matrix(rpois(ngenes*100, mu1), nrow=ngenes) # Pure type 1
counts.2 <- matrix(rpois(ngenes*100, mu2), nrow=ngenes) # Pure type 2
counts.3 <- matrix(rpois(ngenes*100, mu3), nrow=ngenes) # Pure type 3
counts.4 <- matrix(rpois(ngenes*100, mu4), nrow=ngenes) # Pure type 4
counts.m <- matrix(rpois(ngenes*20, mu1+mu2), nrow=ngenes) # Doublets (1 & 2)

counts <- cbind(counts.1, counts.2, counts.3, counts.4, counts.m)
clusters <- rep(1:5, c(rep(100, 4), ncol(counts.m)))

# Find potential doublets.
scores <- doubletCells(counts)
boxplot(split(log10(scores), clusters))
```

doubletCluster *Detect doublet clusters*

Description

Identify potential clusters of doublet cells based on whether they have intermediate expression profiles, i.e., their profiles lie between two other “source” clusters. This function is now deprecated, use `findDoubletClusters` from **scDbIFinder** instead.

Usage

```
doubletCluster(x, ...)

## S4 method for signature 'ANY'
doubletCluster(x, clusters, subset.row = NULL, threshold = 0.05, ...)

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

## S4 method for signature 'SingleCellExperiment'
doubletCluster(x, clusters = collabels(x, onAbsence = "error"), ...)
```

Arguments

x	A numeric matrix-like object of count values, where each column corresponds to a cell and each row corresponds to an endogenous gene. Alternatively, a SummarizedExperiment or SingleCellExperiment object containing such a matrix.
...	For the generic, additional arguments to pass to specific methods. For the ANY method, additional arguments to pass to findMarkers . For the <code>SummarizedExperiment</code> method, additional arguments to pass to the ANY method. For the <code>SingleCellExperiment</code> method, additional arguments to pass to the <code>SummarizedExperiment</code> method.
clusters	A vector of length equal to <code>ncol(x)</code> , containing cluster identities for all cells. If <code>x</code> is a <code>SingleCellExperiment</code> , this is taken from <code>collabels(x)</code> by default.
subset.row	See <code>?"scran-gene-selection"</code> .
threshold	A numeric scalar specifying the FDR threshold with which to identify significant genes.
assay.type	A string specifying which assay values to use, e.g., "counts" or "logcounts".

Details

This function detects clusters of doublet cells in a manner similar to the method used by Bach et al. (2017). For each “query” cluster, we examine all possible pairs of “source” clusters, hypothesizing that the query consists of doublets formed from the two sources. If so, gene expression in the query cluster should be strictly intermediate between the two sources after library size normalization.

We apply pairwise t-tests to the normalized log-expression profiles (see [logNormCounts](#)) to reject this null hypothesis. This is done by identifying genes that are consistently up- or down-regulated

in the query compared to *both* of the sources. We count the number of genes that reject the null hypothesis at the specified FDR threshold. For each query cluster, the most likely pair of source clusters is that which minimizes the number of significant genes.

Potential doublet clusters are identified using the following characteristics:

- Low number of significant genes, i.e., `N` in the output `DataFrame`. The threshold can be identified by looking for small outliers in $\log(N)$ across all clusters, under the assumption that most clusters are *not* doublets (and thus should have high `N`).
- A reasonable proportion of cells in the cluster, i.e., `prop`. This requires some expectation of the doublet rate in the experimental protocol.
- Library sizes of the source clusters that are below that of the query cluster, i.e., `lib.size*` values below unity. This assumes that the doublet cluster will contain more RNA and have more counts than either of the two source clusters.

For each query cluster, the function will only report the pair of source clusters with the lowest `N`. It is possible that a source pair with slightly higher (but still low) value of `N` may have more appropriate `lib.size*` values. Thus, it may be valuable to examine `all.pairs` in the output, especially in over-clustered data sets with closely neighbouring clusters.

The reported `p.value` is of little use in a statistical sense, and is only provided for inspection. Technically, it could be treated as the Simes combined p-value against the doublet hypothesis for the query cluster. However, this does not account for the multiple testing across all pairs of clusters for each chosen cluster, especially as we are choosing the pair that is most concordant with the doublet null hypothesis.

We use library size normalization (via [librarySizeFactors](#)) even if existing size factors are present. This is because intermediate expression of the doublet cluster is not guaranteed for arbitrary size factors. For example, expression in the doublet cluster will be higher than that in the source clusters if normalization was performed with spike-in size factors.

Value

A `DataFrame` containing one row per query cluster with the following fields:

`source1`: String specifying the identity of the first source cluster.

`source2`: String specifying the identity of the second source cluster.

`N`: Integer, number of genes that are significantly non-intermediate in the query cluster compared to the two putative source clusters.

`best`: String specifying the identity of the top gene with the lowest p-value against the doublet hypothesis for this combination of query and source clusters.

`p.value`: Numeric, containing the adjusted p-value for the best gene.

`lib.size1`: Numeric, ratio of the median library sizes for the first source cluster to the query cluster.

`lib.size2`: Numeric, ratio of the median library sizes for the second source cluster to the query cluster.

`prop`: Numeric, proportion of cells in the query cluster.

`all.pairs`: A [SimpleList](#) object containing the above statistics for every pair of potential source clusters.

Each row is named according to its query cluster.

Author(s)

Aaron Lun

References

Bach K, Pensa S, Grzelak M, Hadfield J, Adams DJ, Marioni JC and Khaled WT (2017). Differentiation dynamics of mammary epithelial cells revealed by single-cell RNA sequencing. *Nat Commun.* 8, 1:2128.

Lun ATL (2018). Detecting clusters of doublet cells in *scraper*. https://lun.github.io/SingleCellThoughts/software/doublet_detection/bycluster.html

See Also

[doubletCells](#), which provides another approach for doublet detection.

[findMarkers](#), to detect DE genes between clusters.

Examples

```
# Mocking up an example.
ngenes <- 100
mu1 <- 2^rexp(ngenes)
mu2 <- 2^rnorm(ngenes)

counts.1 <- matrix(rpois(ngenes*100, mu1), nrow=ngenes)
counts.2 <- matrix(rpois(ngenes*100, mu2), nrow=ngenes)
counts.m <- matrix(rpois(ngenes*20, mu1+mu2), nrow=ngenes)

counts <- cbind(counts.1, counts.2, counts.m)
clusters <- rep(1:3, c(ncol(counts.1), ncol(counts.2), ncol(counts.m)))

# Compute doublet-ness of each cluster:
dbl <- doubletCluster(counts, clusters)
dbl

# Narrow this down to clusters with very low 'N':
library(scuttle)
isOutlier(dbl$N, log=TRUE, type="lower")

# Get help from "lib.size" below 1.
dbl$lib.size1 < 1 & dbl$lib.size2 < 1
```

doubletRecovery

Recover intra-sample doublets

Description

Recover intra-sample doublets that are neighbors to known inter-sample doublets in a multiplexed experiment. This function is now deprecated, use `recoverDoublets` from **scDblFinder** instead.

Usage

```
doubletRecovery(x, ...)

## S4 method for signature 'ANY'
doubletRecovery(
  x,
  doublets,
  samples,
  k = 50,
  transposed = FALSE,
  subset.row = NULL,
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam()
)

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

## S4 method for signature 'SingleCellExperiment'
doubletRecovery(x, ..., use.dimred = NULL)
```

Arguments

x	A log-expression matrix for all cells (including doublets) in columns and genes in rows. Alternatively, a SummarizedExperiment or SingleCellExperiment containing such a matrix. If transposed=TRUE, a matrix of low-dimensional coordinates where each row corresponds to a cell. This can also be in the reducedDims of a SingleCellExperiment if use.dimred is specified.
...	For the generic, additional arguments to pass to specific methods. For the SummarizedExperiment method, additional arguments to pass to the ANY method. For the SingleCellExperiment method, additional arguments to pass to the SummarizedExperiment method.
doublets	A logical, integer or character vector specifying which cells in x are known (inter-sample) doublets.
samples	A numeric vector containing the relative proportions of cells from each sample, used to determine how many cells are to be considered as intra-sample doublets.
k	Integer scalar specifying the number of nearest neighbors to use for computing the local doublet proportions.
transposed	Logical scalar indicating whether x is transposed, i.e., cells in the rows.
subset.row	See ?"scran-gene-selection" , specifying the genes to use for the neighbor search. Only used when transposed=FALSE.
BNPARAM	A BiocNeighborParam object specifying the algorithm to use for the nearest neighbor search.
BPPARAM	A BiocParallelParam object specifying the parallelization to use for the nearest neighbor search.
assay.type	A string specifying which assay values contain the log-expression matrix.
use.dimred	A string specifying whether existing values in reducedDims(x) should be used.

Details

In multiplexed single-cell experiments, we can detect doublets as libraries with labels for multiple samples. However, this approach fails to identify doublets consisting of two cells with the same label. Such cells may be problematic if they are still sufficiently abundant to drive formation of spurious clusters.

This function identifies intra-sample doublets based on the similarity in expression profiles to known inter-sample doublets. For each cell, we compute the proportion of the k neighbors that are known doublets. Of the “unmarked” cells that are not known doublets, those with top X largest proportions are considered to be intra-sample doublets.

To compute X , we assume that the formation of doublets is random with respect to their originating samples. This allows us to use `samples` to estimate the expected percentage of doublets that should occur within samples. We then convert into an absolute number X based on the number of known doublets in `doublets`.

A larger value of k provides more stable estimates of the doublet proportion in each cell. However, this comes at the cost of assuming that each cell actually has k neighboring cells of the same state. For example, if a doublet cluster has fewer than k members, its doublet proportions will be “diluted” by inclusion of unmarked cells in the next-closest cluster.

In principle, it is also possible to identify inter-sample doublets by applying a hard threshold on the doublet proportion. This threshold can be set close to the expected percentage from `samples` (i.e., the same one used to derive X). Unfortunately, in practice, the observed proportions are generally lower than expected, possibly due to contamination of doublet subpopulations by unmarked cells in noisy expression data. This motivates the use of a top X approach instead.

Value

A `DataFrame` containing one row per cell and the following fields:

- `proportion`, a numeric field containing the proportion of neighbors that are doublets.
- `known`, a logical field indicating whether this cell is a known inter-sample doublet.
- `predicted`, a logical field indicating whether this cell is a predicted intra-sample doublet.

The `metadata` contains `intra`, a numeric scalar containing the expected number of intra-sample doublets.

Author(s)

Aaron Lun

See Also

`doubletCells` and `doubletCluster`, for alternative methods of doublet detection when no prior doublet information is available.

`hashedDrops` from the **DropletUtils** package, to identify doublets from cell hashing experiments.

Examples

```
# Mocking up an example.
set.seed(100)
ngenest <- 1000
mu1 <- 2^rnorm(ngenest, sd=2)
mu2 <- 2^rnorm(ngenest, sd=2)
```

```

counts.1 <- matrix(rpois(ngenes*100, mu1), nrow=ngenes) # Pure type 1
counts.2 <- matrix(rpois(ngenes*100, mu2), nrow=ngenes) # Pure type 2
counts.m <- matrix(rpois(ngenes*20, mu1+mu2), nrow=ngenes) # Doublets (1 & 2)
all.counts <- cbind(counts.1, counts.2, counts.m)
lcounts <- scuttle::normalizeCounts(all.counts)

# Pretending that half of the doublets are known. Also pretending that
# the experiment involved two samples of equal size.
known <- 200 + seq_len(10)
out <- doubletRecovery(lcounts, doublets=known, k=10, samples=c(1, 1))
out

```

findMarkers

Find marker genes

Description

Find candidate marker genes for groups of cells (e.g., clusters) by testing for differential expression between pairs of groups.

Usage

```

findMarkers(x, ...)

## S4 method for signature 'ANY'
findMarkers(
  x,
  groups,
  test.type = c("t", "wilcox", "binom"),
  ...,
  pval.type = c("any", "some", "all"),
  min.prop = NULL,
  log.p = FALSE,
  full.stats = FALSE,
  sorted = TRUE,
  row.data = NULL,
  add.summary = FALSE,
  BPPARAM = SerialParam()
)

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

## S4 method for signature 'SingleCellExperiment'
findMarkers(x, groups = colLabels(x, onAbsence = "error"), ...)

```

Arguments

x A numeric matrix-like object of expression values, where each column corresponds to a cell and each row corresponds to an endogenous gene. This is expected to be normalized log-expression values for most tests - see Details.

Alternatively, a [SummarizedExperiment](#) or [SingleCellExperiment](#) object containing such a matrix.

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

- For `test.type="t"`, further arguments to pass to [pairwiseTTests](#).
- For `test.type="wilcox"`, further arguments to pass to [pairwiseWilcox](#).
- For `test.type="binom"`, further arguments to pass to [pairwiseBinom](#).

Common arguments for all testing functions include `gene.names`, `direction`, `block` and `BPPARAM`. Test-specific arguments are also supported for the appropriate `test.type`.

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

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

<code>groups</code>	A vector of length equal to <code>ncol(x)</code> , specifying the group to which each cell is assigned. If <code>x</code> is a SingleCellExperiment , this defaults to <code>colLabels(x)</code> if available.
<code>test.type</code>	String specifying the type of pairwise test to perform - a t-test with "t", a Wilcoxon rank sum test with "wilcox", or a binomial test with "binom".
<code>pval.type</code>	A string specifying how p-values are to be combined across pairwise comparisons for a given group/cluster.
<code>min.prop</code>	Numeric scalar specifying the minimum proportion of significant comparisons per gene, Defaults to 0.5 when <code>pval.type="some"</code> , otherwise defaults to zero.
<code>log.p</code>	A logical scalar indicating if log-transformed p-values/FDRs should be returned.
<code>full.stats</code>	A logical scalar indicating whether all statistics in <code>de.lists</code> should be stored in the output for each pairwise comparison.
<code>sorted</code>	Logical scalar indicating whether each output <code>DataFrame</code> should be sorted by a statistic relevant to <code>pval.type</code> .
<code>row.data</code>	A DataFrame containing additional row metadata for each gene in <code>x</code> , to be included in each of the output <code>DataFrames</code> . This should generally have row names identical to those of <code>x</code> . Alternatively, a list containing one such <code>DataFrame</code> per level of groups, where each <code>DataFrame</code> contains group-specific metadata for each gene to be included in the appropriate output <code>DataFrame</code> .
<code>add.summary</code>	Logical scalar indicating whether statistics from summaryMarkerStats should be added.
<code>BPPARAM</code>	A BiocParallelParam object indicating whether and how parallelization should be performed across genes.
<code>assay.type</code>	A string specifying which assay values to use, usually "logcounts".

Details

This function provides a convenience wrapper for marker gene identification between groups of cells, based on running [pairwiseTTests](#) or related functions and passing the result to [combineMarkers](#). All of the arguments above are supplied directly to one of these two functions - refer to the relevant function's documentation for more details.

If `x` contains log-normalized expression values generated with a pseudo-count of 1, it can be used in any of the pairwise testing procedures. If `x` is scale-normalized but not log-transformed, it can be

used with `test.type="wilcox"` and `test.type="binom"`. If `x` contains raw counts, it can only be used with `test.type="binom"`.

Note that `log.p` only affects the combined p-values and FDRs. If `full.stats=TRUE`, the p-values for each individual pairwise comparison will always be log-transformed, regardless of the value of `log.p`. Log-transformed p-values and FDRs are reported using the natural base.

The choice of `pval.type` determines whether the highly ranked genes are those that are DE between the current group and:

- any other group ("any")
- all other groups ("all")
- some other groups ("some")

See [?combineMarkers](#) for more details.

Value

A named list of [DataFrames](#), each of which contains a sorted marker gene list for the corresponding group. In each [DataFrame](#), the top genes are chosen to enable separation of that group from all other groups. See [?combineMarkers](#) for more details on the output format.

If `row.data` is provided, the additional fields are added to the front of the [DataFrame](#) for each cluster. If `add.summary=TRUE`, extra statistics for each cluster are also computed and added.

Any log-fold changes are reported as differences in average `x` between groups (usually in base 2, depending on the transformation applied to `x`).

Author(s)

Aaron Lun

See Also

[pairwiseTTests](#), [pairwiseWilcox](#), [pairwiseBinom](#), for the underlying functions that compute the pairwise DE statistics.

[combineMarkers](#), to combine pairwise statistics into a single marker list per cluster.

[summaryMarkerStats](#), to incorporate additional summary statistics per cluster.

[getMarkerEffects](#), to easily extract a matrix of effect sizes from each [DataFrame](#).

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Any clustering method is okay, only using k-means for convenience.
kout <- kmeans(t(logcounts(sce)), centers=4)

out <- findMarkers(sce, groups=kout$cluster)
names(out)
out[[1]]

# More customization of the tests:
out <- findMarkers(sce, groups=kout$cluster, test.type="wilcox")
out[[1]]
```

```

out <- findMarkers(sce, groups=kout$cluster, lfc=1, direction="up")
out[[1]]

out <- findMarkers(sce, groups=kout$cluster, pval.type="all")
out[[1]]

```

fitTrendCV2

Fit a trend to the CV2

Description

Fit a mean-dependent trend to the squared coefficient of variation, computed from count data after size factor normalization.

Usage

```

fitTrendCV2(
  means,
  cv2,
  ncells,
  min.mean = 0.1,
  nls.args = list(),
  simplified = TRUE,
  nmads = 6,
  max.iter = 50
)

```

Arguments

means	A numeric vector containing mean normalized expression values for all genes.
cv2	A numeric vector containing the squared coefficient of variation computed from normalized expression values for all genes.
ncells	Integer scalar specifying the number of cells used to compute cv2 and means.
min.mean	Numeric scalar specifying the minimum mean to use for trend fitting.
nls.args	A list of parameters to pass to nls .
simplified	Logical scalar indicating whether the function can automatically use a simpler trend if errors are encountered for the usual parameterization.
nmads	Numeric scalar specifying the number of MADs to use to compute the tricube bandwidth during robustification.
max.iter	Integer scalar specifying the maximum number of robustness iterations to perform.

Details

This function fits a mean-dependent trend to the CV2 of normalized expression values for the selected features. Specifically, it fits a trend of the form

$$y = A + \frac{B}{x}$$

using an iteratively reweighted least-squares approach implemented via `nls`. This trend is based on a similar formulation from **DESeq2** and generally captures the mean-CV2 trend well.

Trend fitting is performed after weighting each observation according to the inverse of the density of observations at the same mean. This avoids problems with differences in the distribution of means that would otherwise favor good fits in highly dense intervals at the expense of sparser intervals. Low-abundance genes with means below `min.mean` are also removed prior to fitting, to avoid problems with discreteness and the upper bound on the CV2 at low counts.

Robustness iterations are also performed to protect against outliers. An initial fit is performed and each observation is weighted using tricube-transformed standardized residuals (in addition to the existing inverse-density weights). The bandwidth of the tricube scheme is defined as `nmads` multiplied by the median standardized residual. Iterations are performed until convergence or `max.its` is reached.

Occasionally, there are not enough high-abundance points to uniquely determine the A parameter. In such cases, the function collapses back to fitting a simpler trend

$$y = \frac{B}{x}$$

to avoid errors about singular gradients in `nls`. If `simplified=FALSE`, this simplification is not allowed and the error is directly reported.

Value

A named list is returned containing:

`trend`: A function that returns the fitted value of the trend at any value of the mean.

`std.dev`: A numeric scalar containing the robust standard deviation of the ratio of `var` to the fitted value of the trend across all features used for trend fitting.

Author(s)

Aaron Lun

References

Brennecke P, Anders S, Kim JK et al. (2013). Accounting for technical noise in single-cell RNA-seq experiments. *Nat. Methods* 10:1093-95

See Also

`modelGeneCV2` and `modelGeneCV2WithSpikes`, where this function is used.

Examples

```

library(scuttle)
sce <- mockSCE()
normcounts <- normalizeCounts(sce, log=FALSE)

# Fitting a trend:
library(DelayedMatrixStats)
means <- rowMeans(normcounts)
cv2 <- rowVars(normcounts)/means^2
fit <- fitTrendCV2(means, cv2, ncol(sce))

# Examining the trend fit:
plot(means, cv2, pch=16, cex=0.5,
     xlab="Mean", ylab="CV2", log="xy")
curve(fit$trend(x), add=TRUE, col="dodgerblue", lwd=3)

```

fitTrendPoisson

Generate a trend for Poisson noise

Description

Create a mean-variance trend for log-normalized expression values derived from Poisson-distributed counts.

Usage

```

fitTrendPoisson(
  means,
  size.factors,
  npts = 1000,
  dispersion = 0,
  pseudo.count = 1,
  BPPARAM = SerialParam(),
  ...
)

```

Arguments

means	A numeric vector of length 2 or more, containing the range of mean counts observed in the dataset.
size.factors	A numeric vector of size factors for all cells in the dataset.
npts	An integer scalar specifying the number of interpolation points to use.
dispersion	A numeric scalar specifying the dispersion for the NB distribution. If zero, a Poisson distribution is used.
pseudo.count	A numeric scalar specifying the pseudo-count to be added to the scaled counts before log-transformation.
BPPARAM	A BiocParallelParam object indicating how parallelization should be performed across interpolation points.
...	Further arguments to pass to fitTrendVar for trend fitting.

Details

This function is useful for modelling technical noise in highly diverse datasets without spike-ins, where fitting a trend to the endogenous genes would not be appropriate given the strong biological heterogeneity. It is mostly intended for UMI datasets where the technical noise is close to Poisson-distributed.

This function operates by simulating Poisson or negative binomial-distributed counts, computing log-transformed normalized expression values from those counts, calculating the mean and variance and then passing those metrics to `fitTrendVar`. The log-transformation ensures that variance is modelled in the same space that is used for downstream analyses like PCA.

Simulations are performed across a range of values in means to achieve reliable interpolation, with the stability of the trend determined by the number of simulation points in `npts`. The number of cells is determined from the length of `size.factors`, which are used to scale the distribution means prior to sampling counts.

Value

A named list is returned containing:

`trend`: A function that returns the fitted value of the trend at any value of the mean.

`std.dev`: A numeric scalar containing the robust standard deviation of the ratio of var to the fitted value of the trend across all features used for trend fitting.

Author(s)

Aaron Lun

See Also

`fitTrendVar`, which is used to fit the trend.

Examples

```
# Mocking up means and size factors:
sf <- 2^rnorm(1000, sd=0.1)
sf <- sf/mean(sf)
means <- rexp(100, 0.1)

# Using these to construct a Poisson trend:
out <- fitTrendPoisson(means, sf)
curve(out$trend(x), xlim=c(0, 10))
```

fitTrendVar

Fit a trend to the variances of log-counts

Description

Fit a mean-dependent trend to the variances of the log-normalized expression values derived from count data.

Usage

```
fitTrendVar(
  means,
  vars,
  min.mean = 0.1,
  parametric = TRUE,
  lowess = TRUE,
  density.weights = TRUE,
  nls.args = list(),
  ...
)
```

Arguments

<code>means</code>	A numeric vector containing the mean log-expression value for each gene.
<code>vars</code>	A numeric vector containing the variance of log-expression values for each gene.
<code>min.mean</code>	A numeric scalar specifying the minimum mean to use for trend fitting.
<code>parametric</code>	A logical scalar indicating whether a parametric fit should be attempted.
<code>lowess</code>	A logical scalar indicating whether a LOWESS fit should be attempted.
<code>density.weights</code>	A logical scalar indicating whether to use inverse density weights.
<code>nls.args</code>	A list of parameters to pass to <code>nls</code> if <code>parametric=TRUE</code> .
<code>...</code>	Further arguments to pass to <code>weightedLowess</code> for LOWESS fitting.

Details

This function fits a mean-dependent trend to the variance of the log-normalized expression for the selected features. The fitted trend can then be used to decompose the variance of each gene into biological and technical components, as done in `modelGeneVar` and `modelGeneVarWithSpikes`.

The default fitting process follows a two-step procedure when `parametric=TRUE` and `lowess=TRUE`:

1. A non-linear curve of the form

$$y = \frac{ax}{x^n + b}$$

is fitted to the variances against the means using `nls`. Starting values and the number of iterations are automatically set if not explicitly specified in `nls.args`.

2. `weightedLowess` is applied to the log-ratios of the variance of each gene to the corresponding fitted value from the non-linear curve. The final trend is defined as the product of the fitted values from the non-linear curve and the exponential of the LOWESS fitted value. If any tuning is necessary, the most important parameter here is `span`, which can be passed in the `...` arguments.

The aim is to use the parametric curve to reduce the sharpness of the expected mean-variance relationship for easier smoothing. Conversely, the parametric form is not exact, so the smoothers will model any remaining trends in the residuals.

If `parametric=FALSE`, LOWESS smoothing is performed directly on the log-variances using `weightedLowess`. This may be helpful in situations where the data does not follow the expected parametric curve, e.g., for transformed data that spans negative values where the expression is not defined. (Indeed, the expression above is purely empirical, chosen simply as it matched the shape of the observed trend in many datasets.)

If `lowess=FALSE`, the LOWESS smoothing step is omitted and the parametric fit is used directly. This may be necessary in situations where the LOWESS overfits, e.g., because of very few points at high abundances.

Value

A named list is returned containing:

`trend`: A function that returns the fitted value of the trend at any value of the mean.

`std.dev`: A numeric scalar containing the robust standard deviation of the ratio of `var` to the fitted value of the trend across all features used for trend fitting.

Filtering by mean

Genes with mean log-expression below `min.mean` are not used in trend fitting. This aims to remove the majority of low-abundance genes and preserve the sensitivity of span-based smoothers at moderate-to-high abundances. It also protects against discreteness, which can interfere with estimation of the variability of the variance estimates and accurate scaling of the trend.

Filtering is applied on the mean log-expression to avoid introducing spurious trends at the filter boundary. The default threshold is chosen based on the point at which discreteness is observed in variance estimates from Poisson-distributed counts. For heterogeneous droplet data, a lower threshold of 0.001-0.01 may be more appropriate, though this usually does not matter all too much.

When extrapolating to values below the smallest observed mean (or `min.mean`), the output function will approach zero as the mean approaches zero. This reflects the fact that the variance should be zero at a log-expression of zero (assuming a pseudo-count of 1 was used). When extrapolating to values above the largest observed mean, the output function will be set to the fitted value of the trend at the largest mean.

Weighting by density

All fitting (with `nls` and `weightedLowess`) is performed by weighting each observation according to the inverse of the density of observations at the same mean. This ensures that parts of the curve with few points are fitted as well as parts of the trend with many points. Otherwise, differences in the distribution of means would favor good fits in highly dense intervals at the expense of sparser intervals. (Note that these densities are computed after filtering on `min.mean`, so the high density of points at zero has no effect.)

That being said, the density weighting can give inappropriate weight to very sparse intervals, especially those at high abundance. This results in overfitting where the trend is compelled to pass through each point at these intervals. For most part, this is harmless as most high-abundance genes are not highly variable so an overfitted trend is actually appropriate. However, if high-abundance genes are variable, it may be better to set `density.weights=FALSE` to avoid this overfitting effect.

Author(s)

Aaron Lun

See Also

[modelGeneVar](#) and [modelGeneVarWithSpikes](#), where this function is used.

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Fitting a trend:
library(DelayedMatrixStats)
means <- rowMeans(logcounts(sce))
vars <- rowVars(logcounts(sce))
fit <- fitTrendVar(means, vars)

# Comparing the two trend fits:
plot(means, vars, pch=16, cex=0.5, xlab="Mean", ylab="Variance")
curve(fit$trend(x), add=TRUE, col="dodgerblue", lwd=3)
```

Gene selection

Gene selection

Description

Details on how gene selection is performed in almost all **scran** functions.

Subsetting by row

For functions accepting some gene-by-cell matrix x , we can choose to perform calculations only on a subset of rows (i.e., genes) with the `subset.row` argument. This can be a logical, integer or character vector indicating the rows of x to use. If a character vector, it must contain the names of the rows in x . Future support will be added for more esoteric subsetting vectors like the Bioconductor [Rle](#) classes.

The output of running a function with `subset.row` will *always* be the same as the output of subsetting x beforehand and passing it into the function. However, it is often more efficient to use `subset.row` as we can avoid constructing an intermediate subsetting matrix. The same reasoning applies for any x that is a [SingleCellExperiment](#) object.

Filtering by mean

Some functions will have a `min.mean` argument to filter out low-abundance genes prior to processing. Depending on the function, the filter may be applied to the average library size-adjusted count computed by [calculateAverage](#), the average log-count, or some other measure of abundance - see the documentation for each function for details.

Any filtering on `min.mean` is automatically intersected with a specified `subset.row`. For example, only subsetting genes that pass the filter are retained if `subset.row` is specified alongside `min.mean`.

Author(s)

Aaron Lun

getClusteredPCs *Use clusters to choose the number of PCs*

Description

Cluster cells after using varying number of PCs, and pick the number of PCs using a heuristic based on the number of clusters.

Usage

```
getClusteredPCs(
  pcs,
  FUN = NULL,
  ...,
  min.rank = 5,
  max.rank = ncol(pcs),
  by = 1
)
```

Arguments

pcs	A numeric matrix of PCs, where rows are cells and columns are dimensions representing successive PCs.
FUN	A clustering function that takes a numeric matrix with rows as cells and returns a vector containing a cluster label for each cell.
...	Further arguments to pass to FUN.
min.rank	Integer scalar specifying the minimum number of PCs to use.
max.rank	Integer scalar specifying the maximum number of PCs to use.
by	Integer scalar specifying what intervals should be tested between min.rank and max.rank.

Details

Assume that the data contains multiple subpopulations, each of which is separated from the others on a different axis. For example, each subpopulation could be defined by a unique set of marker genes that drives separation on its own PC. If we had x subpopulations, we would need at least $x - 1$ PCs to successfully distinguish all of them. This motivates the choice of the number of PCs provided we know the number of subpopulations in the data.

In practice, we do not know the number of subpopulations so we use the number of clusters as a proxy instead. We apply a clustering function FUN on the first d PCs, and only consider the values of d that yield no more than $d + 1$ clusters. If we see more clusters with fewer dimensions, we consider this to represent overclustering rather than distinct subpopulations, as multiple subpopulations should not be distinguishable on the same axes (based on the assumption above).

We choose d that satisfies the constraint above and maximizes the number of clusters. The idea is that more PCs should include more biological signal, allowing FUN to detect more distinct subpopulations; until the point that the extra signal outweighs the added noise at high dimensions, such that resolution decreases and it becomes more difficult for FUN to distinguish between subpopulations.

Any FUN can be used that automatically chooses the number of clusters based on the data. The default is a graph-based clustering method using `buildSNNGraph` and `cluster_walktrap`, where

arguments in ... are passed to the former. Users should not supply FUN where the number of clusters is fixed in advance, (e.g., k-means, hierarchical clustering with known k in [cutree](#)).

The identities of the output clusters are returned at each step for comparison, e.g., using methods like **clustree**.

Value

A [DataFrame](#) with one row per tested number of PCs. This contains the fields:

n.pcs: Integer scalar specifying the number of PCs used.

n.clusters: Integer scalar specifying the number of clusters identified.

clusters: A [List](#) containing the cluster identities for this number of PCs.

The metadata of the DataFrame contains chosen, an integer scalar specifying the “ideal” number of PCs to use.

Author(s)

Aaron Lun

See Also

[runPCA](#), to compute the PCs in the first place.

[buildSNNGraph](#), for arguments to use in with default FUN.

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

sce <- scater::runPCA(sce)
output <- getClusteredPCs(reducedDim(sce))
output

metadata(output)$chosen
```

getMarkerEffects *Get marker effect sizes*

Description

Utility function to extract the marker effect sizes as a matrix from the output of [findMarkers](#).

Usage

```
getMarkerEffects(x, prefix = "logFC", strip = TRUE, remove.na.col = FALSE)
```

Arguments

x	A DataFrame containing marker statistics for a given group/cluster, usually one element of the List returned by findMarkers .
prefix	String containing the prefix for the columns containing the effect size.
strip	Logical scalar indicating whether the prefix should be removed from the output column names.
remove.na.col	Logical scalar indicating whether to remove columns containing any NAs.

Details

Setting `remove.na.col=TRUE` may be desirable in applications involving blocked comparisons, where some pairwise comparisons are not possible if the relevant levels occur in different blocks. In such cases, the resulting column is filled with NAs that may interfere with downstream steps like clustering.

Value

A numeric matrix containing the effect sizes for the comparison to every other group/cluster.

Author(s)

Aaron Lun

See Also

[findMarkers](#) and [combineMarkers](#), to generate the DataFrames.

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

kout <- kmeans(t(logcounts(sce)), centers=4)
out <- findMarkers(sce, groups=kout$cluster)

eff1 <- getMarkerEffects(out[[1]])
str(eff1)
```

getTopHVGs

Identify HVGs

Description

Define a set of highly variable genes, based on variance modelling statistics from [modelGeneVar](#) or related functions.

Usage

```

getTopHVGs(
  stats,
  var.field = "bio",
  n = NULL,
  prop = NULL,
  var.threshold = 0,
  fdr.field = "FDR",
  fdr.threshold = NULL,
  row.names = !is.null(rownames(stats))
)

```

Arguments

<code>stats</code>	A DataFrame of variance modelling statistics with one row per gene.
<code>var.field</code>	String specifying the column of <code>stats</code> containing the relevant metric of variation.
<code>n</code>	Integer scalar specifying the number of top HVGs to report.
<code>prop</code>	Numeric scalar specifying the proportion of genes to report as HVGs.
<code>var.threshold</code>	Numeric scalar specifying the minimum threshold on the metric of variation.
<code>fdr.field</code>	String specifying the column of <code>stats</code> containing the adjusted p-values. If <code>NULL</code> , no filtering is performed on the FDR.
<code>fdr.threshold</code>	Numeric scalar specifying the FDR threshold.
<code>row.names</code>	Logical scalar indicating whether row names should be reported.

Details

This function will identify all genes where the relevant metric of variation is greater than `var.threshold`. By default, this means that we retain all genes with positive values in the `var.field` column of `stats`. If `var.threshold=NULL`, the minimum threshold on the value of the metric is not applied.

If `fdr.threshold` is specified, we further subset to genes that have FDR less than or equal to `fdr.threshold`. By default, FDR thresholding is turned off as [modelGeneVar](#) and related functions determine significance of large variances *relative* to other genes. This can be overly conservative if many genes are highly variable.

If `n=NULL` and `prop=NULL`, the resulting subset of genes is directly returned. Otherwise, the top set of genes with the largest values of the variance metric are returned, where the size of the set is defined as the larger of `n` and `prop*nrow(stats)`.

Value

A character vector containing the names of the most variable genes, if `row.names=TRUE`.

Otherwise, an integer vector specifying the indices of `stats` containing the most variable genes.

Author(s)

Aaron Lun

See Also

[modelGeneVar](#) and friends, to generate `stats`.

[modelGeneCV2](#) and friends, to also generate `stats`.

Examples

```

library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

stats <- modelGeneVar(sce)
str(getTopHVGs(stats))
str(getTopHVGs(stats, fdr.threshold=0.05)) # more stringent

stats2 <- modelGeneCV2(sce)
str(getTopHVGs(stats2, var.field="ratio"))

```

getTopMarkers	<i>Get top markers</i>
---------------	------------------------

Description

Obtain the top markers for each pairwise comparison between clusters, or for each cluster.

Usage

```

getTopMarkers(
  de.lists,
  pairs,
  n = 10,
  pval.field = "p.value",
  fdr.field = "FDR",
  pairwise = TRUE,
  pval.type = c("any", "some", "all"),
  fdr.threshold = 0.05,
  ...
)

```

Arguments

de.lists	A list-like object where each element is a data.frame or DataFrame . Each element should represent the results of a pairwise comparison between two groups/clusters, in which each row should contain the statistics for a single gene/feature. Rows should be named by the feature name in the same order for all elements.
pairs	A matrix, data.frame or DataFrame with two columns and number of rows equal to the length of de.lists. Each row should specify the pair of clusters being compared for the corresponding element of de.lists.
n	Integer scalar specifying the number of markers to obtain from each pairwise comparison, if pairwise=FALSE. Otherwise, the number of top genes to take from each cluster's combined marker set, see Details.
pval.field	String specifying the column of each DataFrame in de.lists to use to identify top markers. Smaller values are assigned higher rank.

<code>fdr.field</code>	String specifying the column containing the adjusted p-values.
<code>pairwise</code>	Logical scalar indicating whether top markers should be returned for every pairwise comparison. If <code>FALSE</code> , one marker set is returned for every cluster.
<code>pval.type</code>	String specifying how markers from pairwise comparisons are to be combined if <code>pairwise=FALSE</code> . This has the same effect as <code>pval.type</code> in <code>combineMarkers</code> .
<code>fdr.threshold</code>	Numeric scalar specifying the FDR threshold for filtering. If <code>NULL</code> , no filtering is performed on the FDR.
<code>...</code>	Further arguments to pass to <code>combineMarkers</code> if <code>pairwise=FALSE</code> .

Details

This is a convenience utility that converts the results of pairwise comparisons into a marker list that can be used in downstream functions, e.g., as the marker sets in **SingleR**. By default, it returns a list of lists containing the top genes for every pairwise comparison, which is useful for feature selection to select genes distinguishing between closely related clusters. The top `n` genes are chosen with adjusted p-values below `fdr.threshold`.

If `pairwise=FALSE`, `combineMarkers` is called on `de.lists` and `pairs` to obtain a per-cluster ranking of genes from all pairwise comparisons involving that cluster. If `pval.type="any"`, the top genes with Top values no greater than `n` are retained; this is equivalent to taking the union of the top `n` genes from each pairwise comparison for each cluster. Otherwise, the top `n` genes with the smallest p-values are retained. In both cases, genes are further filtered by `fdr.threshold`.

Value

If `pairwise=TRUE`, a [List](#) of Lists of character vectors is returned. Each element of the outer list corresponds to cluster X, each element of the inner list corresponds to another cluster Y, and each character vector specifies the marker genes that distinguish X from Y.

If `pairwise=FALSE`, a List of character vectors is returned. Each character vector contains the marker genes that distinguish X from any, some or all other clusters, depending on `combine.type`.

Author(s)

Aaron Lun

See Also

[pairwiseTTests](#) and friends, to obtain `de.lists` and `pairs`.

[combineMarkers](#), for another function that consolidates pairwise DE comparisons.

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Any clustering method is okay.
kout <- kmeans(t(logcounts(sce)), centers=3)

out <- pairwiseTTests(logcounts(sce),
  groups=paste0("Cluster", kout$cluster))

# Getting top pairwise markers:
```

```

top <- getTopMarkers(out$statistics, out$pairs)
top[[1]]
top[[1]][[2]]

# Getting top per-cluster markers:
top <- getTopMarkers(out$statistics, out$pairs, pairwise=FALSE)
top[[1]]

```

modelGeneCV2

Model the per-gene CV2

Description

Model the squared coefficient of variation (CV2) of the normalized expression profiles for each gene, fitting a trend to account for the mean-variance relationship across genes.

Usage

```

modelGeneCV2(x, ...)

## S4 method for signature 'ANY'
modelGeneCV2(
  x,
  size.factors = NULL,
  block = NULL,
  subset.row = NULL,
  subset.fit = NULL,
  ...,
  equiweight = TRUE,
  method = "fisher",
  BPPARAM = SerialParam()
)

## S4 method for signature 'SummarizedExperiment'
modelGeneCV2(x, ..., assay.type = "counts")

## S4 method for signature 'SingleCellExperiment'
modelGeneCV2(x, size.factors = NULL, ...)

```

Arguments

x	A numeric matrix of counts where rows are genes and columns are cells. Alternatively, a SummarizedExperiment or SingleCellExperiment containing such a matrix.
...	For the generic, further arguments to pass to each method. For the ANY method, further arguments to pass to fitTrendCV2 . 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 of size factors for each cell in x.

block	A factor specifying the blocking levels for each cell in <code>x</code> . If specified, variance modelling is performed separately within each block and statistics are combined across blocks.
subset.row	See <code>?"scran-gene-selection"</code> , specifying the rows for which to model the variance. Defaults to all genes in <code>x</code> .
subset.fit	An argument similar to <code>subset.row</code> , specifying the rows to be used for trend fitting. Defaults to <code>subset.row</code> .
equiweight	A logical scalar indicating whether statistics from each block should be given equal weight. Otherwise, each block is weighted according to its number of cells. Only used if <code>block</code> is specified.
method	String specifying how p-values should be combined when <code>block</code> is specified, see <code>combinePValues</code> .
BPPARAM	A <code>BiocParallelParam</code> object indicating whether parallelization should be performed across genes.
assay.type	String or integer scalar specifying the assay containing the counts.

Details

For each gene, we compute the CV2 and mean of the counts after scaling them by `size.factors`. A trend is fitted to the CV2 against the mean for all genes using `fitTrendCV2`. The fitted value for each gene is used as a proxy for the technical noise, assuming that most genes exhibit a low baseline level of variation that is not biologically interesting. The ratio of the total CV2 to the trend is used as a metric to rank interesting genes, with larger ratios being indicative of strong biological heterogeneity.

By default, the trend is fitted using all of the genes in `x`. If `subset.fit` is specified, the trend is fitted using only the specified subset, and the values of `trend` for all other genes are determined by extrapolation or interpolation. This could be used to perform the fit based on genes that are known to have low variance, thus weakening the assumption above. Note that this does not refer to spike-in transcripts, which should be handled via `modelGeneCV2WithSpikes`.

If no `size.factors` are supplied, they are automatically computed depending on the input type:

- If `size.factors=NULL` for the ANY method, the sum of counts for each cell in `x` is used to compute a size factor via the `librarySizeFactors` function.
- If `size.factors=NULL` for the `SingleCellExperiment` method, `sizeFactors(x)` is used if available. Otherwise, it defaults to library size-derived size factors.

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

Value

A `DataFrame` is returned where each row corresponds to a gene in `x` (or in `subset.row`, if specified). This contains the numeric fields:

`mean`: Mean normalized expression per gene.

`total`: Squared coefficient of variation of the normalized expression per gene.

`trend`: Fitted value of the trend.

`ratio`: Ratio of total to trend.

`p.value`, `FDR`: Raw and adjusted p-values for the test against the null hypothesis that `ratio` ≤ 1.

If `block` is not specified, the metadata of the DataFrame contains the output of running `fitTrendCV2` on the specified features, along with the mean and `cv2` used to fit the trend.

If `block` is specified, the output contains another `.block` field. This field is itself a DataFrame of DataFrames, where each internal DataFrame contains statistics for the variance modelling within each block and has the same format as described above. Each internal DataFrame's metadata contains the output of `fitTrendCV2` for the cells of that block.

Computing p-values

The p-value for each gene is computed by assuming that the CV2 estimates are normally distributed around the trend, and that the standard deviation of the CV2 distribution is proportional to the value of the trend. This is used to construct a one-sided test for each gene based on its `ratio`, under the null hypothesis that the ratio is equal to or less than 1. The proportionality constant for the standard deviation is set to the `std.dev` returned by `fitTrendCV2`. This is estimated from the spread of per-gene CV2 values around the trend, so the null hypothesis effectively becomes “is this gene *more* variable than other genes of the same abundance?”

Handling uninteresting factors

Setting `block` will estimate the mean and variance of each gene for cells in each level of `block` separately. The trend is fitted separately for each level, and the variance decomposition is also performed separately. Per-level statistics are then combined to obtain a single value per gene:

- For means and CV2 values, this is done by taking the geometric mean across blocking levels. If `equiweight=FALSE`, a weighted average is used where the value for each level is weighted by the number of cells. By default, all levels are equally weighted when combining statistics.
- Per-level p-values are combined using `combinePValues` according to `method`. By default, Fisher's method is used to identify genes that are highly variable in any batch. Whether or not this is responsive to `equiweight` depends on the chosen method.
- Blocks with fewer than 2 cells are completely ignored and do not contribute to the combined mean, variance component or p-value.

Author(s)

Aaron Lun

Examples

```
library(scuttle)
sce <- mockSCE()

# Simple case:
spk <- modelGeneCV2(sce)
spk

plot(spk$mean, spk$total, pch=16, log="xy")
curve(metadata(spk)$trend(x), add=TRUE, col="dodgerblue")

# With blocking:
block <- sample(LETTERS[1:2], ncol(sce), replace=TRUE)
blk <- modelGeneCV2(sce, block=block)
blk

par(mfrow=c(1,2))
```

```

for (i in colnames(blk$per.block)) {
  current <- blk$per.block[[i]]
  plot(current$mean, current$total, pch=16, log="xy")
  curve(metadata(current)$trend(x), add=TRUE, col="dodgerblue")
}

```

modelGeneCV2WithSpikes

Model the per-gene CV2 with spike-ins

Description

Model the squared coefficient of variation (CV2) of the normalized expression profiles for each gene, using spike-ins to estimate the baseline level of technical noise at each abundance.

Usage

```

modelGeneCV2WithSpikes(x, ...)

## S4 method for signature 'ANY'
modelGeneCV2WithSpikes(
  x,
  spikes,
  size.factors = NULL,
  spike.size.factors = NULL,
  block = NULL,
  subset.row = NULL,
  ...,
  equiweight = TRUE,
  method = "fisher",
  BPPARAM = SerialParam()
)

## S4 method for signature 'SummarizedExperiment'
modelGeneCV2WithSpikes(x, ..., assay.type = "counts")

## S4 method for signature 'SingleCellExperiment'
modelGeneCV2WithSpikes(
  x,
  spikes,
  size.factors = NULL,
  spike.size.factors = NULL,
  ...,
  assay.type = "counts"
)

```

Arguments

x A numeric matrix of counts where rows are (usually endogenous) genes and columns are cells.

	Alternatively, a SummarizedExperiment or SingleCellExperiment containing such a matrix.
...	For the generic, further arguments to pass to each method. For the ANY method, further arguments to pass to fitTrendCV2 . For the SummarizedExperiment method, further arguments to pass to the ANY method. For the SingleCellExperiment method, further arguments to pass to the SummarizedExperiment method.
spikes	A numeric matrix of counts where each row corresponds to a spike-in transcript. This should have the same number of columns as <code>x</code> . Alternatively, for the SingleCellExperiment method, this can be a string or an integer scalar specifying the altExp containing the spike-in count matrix.
size.factors	A numeric vector of size factors for each cell in <code>x</code> , to be used for scaling gene expression.
spike.size.factors	A numeric vector of size factors for each cell in <code>spikes</code> , to be used for scaling spike-ins.
block	A factor specifying the blocking levels for each cell in <code>x</code> . If specified, variance modelling is performed separately within each block and statistics are combined across blocks.
subset.row	See <code>?“scran-gene-selection”</code> , specifying the rows for which to model the variance. Defaults to all genes in <code>x</code> .
equiweight	A logical scalar indicating whether statistics from each block should be given equal weight. Otherwise, each block is weighted according to its number of cells. Only used if <code>block</code> is specified.
method	String specifying how p-values should be combined when <code>block</code> is specified, see combinePValues .
BPPARAM	A BiocParallelParam object indicating whether parallelization should be performed across genes.
assay.type	String or integer scalar specifying the assay containing the counts. For the SingleCellExperiment method, this is used to retrieve both the endogenous and spike-in counts.

Details

For each gene and spike-in transcript, we compute the variance and CV2 of the normalized expression values. A trend is fitted to the CV2 against the mean for spike-in transcripts using [fitTrendCV2](#). The value of the trend at the abundance of each gene is used to define the variation attributable to technical noise. The ratio to the trend is then used to define overdispersion corresponding to interesting biological heterogeneity.

This function is almost the same as [modelGeneCV2](#), with the only theoretical difference being that the trend is fitted on spike-in CV2 rather than using the means and CV2 of endogenous genes. This is because there are certain requirements for how normalization is performed when comparing spike-in transcripts with endogenous genes - see comments in “Explanation for size factor rescaling”. We enforce this by centering the size factors for both sets of features and recomputing normalized expression values.

Value

A `DataFrame` is returned where each row corresponds to a gene in `x` (or in `subset.row`, if specified). This contains the numeric fields:

`mean`: Mean normalized expression per gene.

`total`: Squared coefficient of variation of the normalized expression per gene.

`trend`: Fitted value of the trend.

`ratio`: Ratio of `total` to `trend`.

`p.value`, `FDR`: Raw and adjusted p-values for the test against the null hypothesis that `ratio` ≤ 1.

If `block` is not specified, the metadata of the `DataFrame` contains the output of running `fitTrendCV2` on the spike-in transcripts, along with the `mean` and `cv2` used to fit the trend.

If `block` is specified, the output contains another `per.block` field. This field is itself a `DataFrame` of `DataFrames`, where each internal `DataFrame` contains statistics for the variance modelling within each block and has the same format as described above. Each internal `DataFrame`'s metadata contains the output of `fitTrendCV2` for the cells of that block.

Computing p-values

The p-value for each gene is computed by assuming that the CV2 estimates are normally distributed around the trend, and that the standard deviation of the CV2 distribution is proportional to the value of the trend. This is used to construct a one-sided test for each gene based on its `ratio`, under the null hypothesis that the ratio is equal to 1. The proportionality constant for the standard deviation is set to the `std.dev` returned by `fitTrendCV2`. This is estimated from the spread of CV2 values for spike-in transcripts, so the null hypothesis effectively becomes “is this gene *more* variable than spike-in transcripts of the same abundance?”

Default size factor choices

If no size factors are supplied, they are automatically computed depending on the input type:

- If `size.factors=NULL` for the ANY method, the sum of counts for each cell in `x` is used to compute a size factor via the `librarySizeFactors` function.
- If `spike.size.factors=NULL` for the ANY method, the sum of counts for each cell in `spikes` is used to compute a size factor via the `librarySizeFactors` function.
- If `size.factors=NULL` for the `SingleCellExperiment` method, `sizeFactors(x)` is used if available. Otherwise, it defaults to library size-derived size factors.
- If `spike.size.factors=NULL` for the `SingleCellExperiment` method and `spikes` is not a matrix, `sizeFactors(altExp(x, spikes))` is used if available. Otherwise, it defaults to library size-derived size factors.

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

Explanation for size factor rescaling

The use of a spike-in-derived trend makes several assumptions. The first is that a constant amount of spike-in RNA was added to each cell, such that any differences in observed expression of the spike-in transcripts can be wholly attributed to technical noise. The second is that endogenous genes and spike-in transcripts follow the same mean-variance relationship, i.e., a spike-in transcript captures the technical noise of an endogenous gene at the same mean count.

Here, the spike-in size factors across all cells are scaled so that their mean is equal to that of the gene-based size factors for the same set of cells. This ensures that the average normalized abundances of the spike-in transcripts are comparable to those of the endogenous genes, allowing the trend fitted to the former to be used to determine the biological component of the latter. Otherwise, differences in scaling of the size factors would shift the normalized expression values of the former away from the latter, violating the second assumption.

If `block` is specified, rescaling is performed separately for all cells in each block. This aims to avoid problems from (frequent) violations of the first assumption where there are differences in the quantity of spike-in RNA added to each batch. Without scaling, these differences would lead to systematic shifts in the spike-in abundances from the endogenous genes when fitting a batch-specific trend (even if there is no global difference in scaling across all batches).

Handling uninteresting factors

Setting `block` will estimate the mean and variance of each gene for cells in each level of `block` separately. The trend is fitted separately for each level, and the variance decomposition is also performed separately. Per-level statistics are then combined to obtain a single value per gene:

- For means and CV2 values, this is done by taking the geometric mean across blocking levels. If `equiweight=FALSE`, a weighted average is used where the value for each level is weighted by the number of cells. By default, all levels are equally weighted when combining statistics.
- Per-level p-values are combined using `combinePValues` according to `method`. By default, Fisher's method is used to identify genes that are highly variable in any batch. Whether or not this is responsive to `equiweight` depends on the chosen method.
- Blocks with fewer than 2 cells are completely ignored and do not contribute to the combined mean, variance component or p-value.

Author(s)

Aaron Lun

See Also

[fitTrendCV2](#), for the trend fitting options.

[modelGeneCV2](#), for modelling variance without spike-in controls.

Examples

```
library(scuttle)
sce <- mockSCE()

# Using spike-ins.
spk <- modelGeneCV2WithSpikes(sce, "Spikes")
spk

plot(spk$mean, spk$total)
points(metadata(spk)$mean, metadata(spk)$var, col="red", pch=16)
curve(metadata(spk)$trend(x), add=TRUE, col="dodgerblue")

# With blocking (and spike-ins).
block <- sample(LETTERS[1:2], ncol(sce), replace=TRUE)
blk <- modelGeneCV2WithSpikes(sce, "Spikes", block=block)
blk
```

```

par(mfrow=c(1,2))
for (i in colnames(blk$per.block)) {
  current <- blk$per.block[[i]]
  plot(current$mean, current$total)
  points(metadata(current)$mean, metadata(current)$var, col="red", pch=16)
  curve(metadata(current)$trend(x), add=TRUE, col="dodgerblue")
}

```

modelGeneVar

Model the per-gene variance

Description

Model the variance of the log-expression profiles for each gene, decomposing it into technical and biological components based on a fitted mean-variance trend.

Usage

```

## S4 method for signature 'ANY'
modelGeneVar(
  x,
  block = NULL,
  design = NULL,
  subset.row = NULL,
  subset.fit = NULL,
  ...,
  equiweight = TRUE,
  method = "fisher",
  BPPARAM = SerialParam()
)

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

```

Arguments

x	A numeric matrix of log-normalized expression values where rows are genes and columns are cells. Alternatively, a SummarizedExperiment containing such a matrix.
block	A factor specifying the blocking levels for each cell in x. If specified, variance modelling is performed separately within each block and statistics are combined across blocks.
design	A numeric matrix containing blocking terms for uninteresting factors of variation.
subset.row	See ?" scran-gene-selection ", specifying the rows for which to model the variance. Defaults to all genes in x.
subset.fit	An argument similar to subset.row, specifying the rows to be used for trend fitting. Defaults to subset.row.

...	For the generic, further arguments to pass to each method. For the ANY method, further arguments to pass to <code>fitTrendVar</code> . For the <code>SummarizedExperiment</code> method, further arguments to pass to the ANY method.
<code>equiweight</code>	A logical scalar indicating whether statistics from each block should be given equal weight. Otherwise, each block is weighted according to its number of cells. Only used if <code>block</code> is specified.
<code>method</code>	String specifying how p-values should be combined when <code>block</code> is specified, see <code>combinePValues</code> .
<code>BPPARAM</code>	A <code>BiocParallelParam</code> object indicating whether parallelization should be performed across genes.
<code>assay.type</code>	String or integer scalar specifying the assay containing the log-expression values.

Details

For each gene, we compute the variance and mean of the log-expression values. A trend is fitted to the variance against the mean for all genes using `fitTrendVar`. The fitted value for each gene is used as a proxy for the technical component of variation for each gene, under the assumption that most genes exhibit a low baseline level of variation that is not biologically interesting. The biological component of variation for each gene is defined as the the residual from the trend.

Ranking genes by the biological component enables identification of interesting genes for downstream analyses in a manner that accounts for the mean-variance relationship. We use log-transformed expression values to blunt the impact of large positive outliers and to ensure that large variances are driven by strong log-fold changes between cells rather than differences in counts. Log-expression values are also used in downstream analyses like PCA, so modelling them here avoids inconsistencies with different quantifications of variation across analysis steps.

By default, the trend is fitted using all of the genes in `x`. If `subset.fit` is specified, the trend is fitted using only the specified subset, and the technical components for all other genes are determined by extrapolation or interpolation. This could be used to perform the fit based on genes that are known to have low variance, thus weakening the assumption above. Note that this does not refer to spike-in transcripts, which should be handled via `modelGeneVarWithSpikes`.

Value

A `DataFrame` is returned where each row corresponds to a gene in `x` (or in `subset.row`, if specified). This contains the numeric fields:

`mean`: Mean normalized log-expression per gene.

`total`: Variance of the normalized log-expression per gene.

`bio`: Biological component of the variance.

`tech`: Technical component of the variance.

`p.value`, `FDR`: Raw and adjusted p-values for the test against the null hypothesis that $\text{bio} \leq 0$.

If `block` is not specified, the metadata of the `DataFrame` contains the output of running `fitTrendVar` on the specified features, along with the `mean` and `var` used to fit the trend.

If `block` is specified, the output contains another `per.block` field. This field is itself a `DataFrame` of `DataFrames`, where each internal `DataFrame` contains statistics for the variance modelling within each block and has the same format as described above. Each internal `DataFrame`'s metadata contains the output of `fitTrendVar` for the cells of that block.

Handling uninteresting factors

Setting `block` will estimate the mean and variance of each gene for cells in each level of `block` separately. The trend is fitted separately for each level, and the variance decomposition is also performed separately. Per-level statistics are then combined to obtain a single value per gene:

- For means and variance components, this is done by averaging values across levels. If `equiweight=FALSE`, a weighted average is used where the value for each level is weighted by the number of cells. By default, all levels are equally weighted when combining statistics.
- Per-level p-values are combined using `combinePValues` according to `method`. By default, Fisher's method is used to identify genes that are highly variable in any batch. Whether or not this is responsive to `equiweight` depends on the chosen method.
- Blocks with fewer than 2 cells are completely ignored and do not contribute to the combined mean, variance component or p-value.

Use of `block` is the recommended approach for accounting for any uninteresting categorical factor of variation. In addition to accounting for systematic differences in expression between levels of the blocking factor, it also accommodates differences in the mean-variance relationships.

Alternatively, uninteresting factors can be used to construct a design matrix to pass to the function via `design`. In this case, a linear model is fitted to the expression profile for each gene and the residual variance is calculated. This approach is useful for covariates or additive models that cannot be expressed as a one-way layout for use in `block`. However, it assumes that the error is normally distributed with equal variance for all observations of a given gene.

Use of `block` and `design` together is currently not supported and will lead to an error.

Computing p-values

The p-value for each gene is computed by assuming that the variance estimates are normally distributed around the trend, and that the standard deviation of the variance distribution is proportional to the value of the trend. This is used to construct a one-sided test for each gene based on its `bio`, under the null hypothesis that the biological component is equal to zero. The proportionality constant for the standard deviation is set to the `std.dev` returned by `fitTrendVar`. This is estimated from the spread of per-gene variance estimates around the trend, so the null hypothesis effectively becomes “is this gene *more* variable than other genes of the same abundance?”

Author(s)

Aaron Lun

See Also

[fitTrendVar](#), for the trend fitting options.

[modelGeneVarWithSpikes](#), for modelling variance with spike-in controls.

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Fitting to all features.
allf <- modelGeneVar(sce)
allf
```

```

plot(allf$mean, allf$total)
curve(metadata(allf)$trend(x), add=TRUE, col="dodgerblue")

# Using a subset of features for fitting.
subf <- modelGeneVar(sce, subset.fit=1:100)
subf

plot(subf$mean, subf$total)
curve(metadata(subf)$trend(x), add=TRUE, col="dodgerblue")
points(metadata(subf)$mean, metadata(subf)$var, col="red", pch=16)

# With blocking.
block <- sample(LETTERS[1:2], ncol(sce), replace=TRUE)
blk <- modelGeneVar(sce, block=block)
blk

par(mfrow=c(1,2))
for (i in colnames(blk$per.block)) {
  current <- blk$per.block[[i]]
  plot(current$mean, current$total)
  curve(metadata(current)$trend(x), add=TRUE, col="dodgerblue")
}

```

modelGeneVarByPoisson *Model the per-gene variance with Poisson noise*

Description

Model the variance of the log-expression profiles for each gene, decomposing it into technical and biological components based on a mean-variance trend corresponding to Poisson noise.

Usage

```

modelGeneVarByPoisson(x, ...)

## S4 method for signature 'ANY'
modelGeneVarByPoisson(
  x,
  size.factors = NULL,
  block = NULL,
  design = NULL,
  subset.row = NULL,
  npts = 1000,
  dispersion = 0,
  pseudo.count = 1,
  ...,
  equiweight = TRUE,
  method = "fisher",
  BPPARAM = SerialParam()
)

## S4 method for signature 'SummarizedExperiment'

```

```
modelGeneVarByPoisson(x, ..., assay.type = "counts")

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

Arguments

x	A numeric matrix of counts where rows are (usually endogenous) genes and columns are cells. Alternatively, a SummarizedExperiment or SingleCellExperiment containing such a matrix.
...	For the generic, further arguments to pass to each method. For the ANY method, further arguments to pass to fitTrendVar . 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 of size factors for each cell in x, to be used for scaling gene expression.
block	A factor specifying the blocking levels for each cell in x. If specified, variance modelling is performed separately within each block and statistics are combined across blocks.
design	A numeric matrix containing blocking terms for uninteresting factors of variation.
subset.row	See ?"scrans-gene-selection" , specifying the rows for which to model the variance. Defaults to all genes in x.
npts	An integer scalar specifying the number of interpolation points to use.
dispersion	A numeric scalar specifying the dispersion for the NB distribution. If zero, a Poisson distribution is used.
pseudo.count	Numeric scalar specifying the pseudo-count to add prior to log-transformation.
equiweight	A logical scalar indicating whether statistics from each block should be given equal weight. Otherwise, each block is weighted according to its number of cells. Only used if block is specified.
method	String specifying how p-values should be combined when block is specified, see combinePValues .
BPPARAM	A BiocParallelParam object indicating whether parallelization should be performed across genes.
assay.type	String or integer scalar specifying the assay containing the log-expression values.

Details

For each gene, we compute the variance and mean of the log-expression values. A trend is fitted to the variance against the mean for simulated Poisson counts as described in [fitTrendPoisson](#). The technical component for each gene is defined as the value of the trend at that gene's mean abundance. The biological component is then defined as the residual from the trend.

This function is similar to [modelGeneVarWithSpikes](#), with the only difference being that the trend is fitted on simulated Poisson count-derived variances rather than spike-ins. The assumption is

that the technical component is Poisson-distributed, or at least negative binomial-distributed with a known constant dispersion. This is useful for UMI count data sets that do not have spike-ins and are too heterogeneous to assume that most genes exhibit negligible biological variability.

If no size factors are supplied, they are automatically computed depending on the input type:

- If `size.factors=NULL` for the ANY method, the sum of counts for each cell in `x` is used to compute a size factor via the `librarySizeFactors` function.
- If `size.factors=NULL` for the `SingleCellExperiment` method, `sizeFactors(x)` is used if available. Otherwise, it defaults to library size-derived size factors.

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

Value

A `DataFrame` is returned where each row corresponds to a gene in `x` (or in `subset.row`, if specified). This contains the numeric fields:

`mean`: Mean normalized log-expression per gene.

`total`: Variance of the normalized log-expression per gene.

`bio`: Biological component of the variance.

`tech`: Technical component of the variance.

`p.value`, `FDR`: Raw and adjusted p-values for the test against the null hypothesis that $\text{bio} \leq 0$.

If `block` is not specified, the metadata of the `DataFrame` contains the output of running `fitTrendVar` on the simulated counts, along with the mean and var used to fit the trend.

If `block` is specified, the output contains another `per.block` field. This field is itself a `DataFrame` of `DataFrames`, where each internal `DataFrame` contains statistics for the variance modelling within each block and has the same format as described above. Each internal `DataFrame`'s metadata contains the output of `fitTrendVar` for the cells of that block.

Computing p-values

The p-value for each gene is computed by assuming that the variance estimates are normally distributed around the trend, and that the standard deviation of the variance distribution is proportional to the value of the trend. This is used to construct a one-sided test for each gene based on its `bio`, under the null hypothesis that the biological component is equal to zero. The proportionality constant for the standard deviation is set to the `std.dev` returned by `fitTrendVar`. This is estimated from the spread of variance estimates for the simulated Poisson-distributed counts, so the null hypothesis effectively becomes “is this gene *more* variable than a hypothetical gene with only Poisson noise?”

Handling uninteresting factors

Setting `block` will estimate the mean and variance of each gene for cells in each level of `block` separately. The trend is fitted separately for each level, and the variance decomposition is also performed separately. Per-level statistics are then combined to obtain a single value per gene:

- For means and variance components, this is done by averaging values across levels. If `equiweight=FALSE`, a weighted average is used where the value for each level is weighted by the number of cells. By default, all levels are equally weighted when combining statistics.
- Per-level p-values are combined using `combinePValues` according to `method`. By default, Fisher's method is used to identify genes that are highly variable in any batch. Whether or not this is responsive to `equiweight` depends on the chosen method.

- Blocks with fewer than 2 cells are completely ignored and do not contribute to the combined mean, variance component or p-value.

Use of `block` is the recommended approach for accounting for any uninteresting categorical factor of variation. In addition to accounting for systematic differences in expression between levels of the blocking factor, it also accommodates differences in the mean-variance relationships.

Alternatively, uninteresting factors can be used to construct a design matrix to pass to the function via `design`. In this case, a linear model is fitted to the expression profile for each gene and the residual variance is calculated. This approach is useful for covariates or additive models that cannot be expressed as a one-way layout for use in `block`. However, it assumes that the error is normally distributed with equal variance for all observations of a given gene.

Use of `block` and `design` together is currently not supported and will lead to an error.

Author(s)

Aaron Lun

See Also

[fitTrendVar](#), for the trend fitting options.

[modelGeneVar](#), for modelling variance without spike-in controls.

Examples

```
library(scuttle)
sce <- mockSCE()

# Using spike-ins.
pois <- modelGeneVarByPoisson(sce)
pois

plot(pois$mean, pois$total, ylim=c(0, 10))
points(metadata(pois)$mean, metadata(pois)$var, col="red", pch=16)
curve(metadata(pois)$trend(x), add=TRUE, col="dodgerblue")

# With blocking.
block <- sample(LETTERS[1:2], ncol(sce), replace=TRUE)
blk <- modelGeneVarByPoisson(sce, block=block)
blk

par(mfrow=c(1,2))
for (i in colnames(blk$per.block)) {
  current <- blk$per.block[[i]]
  plot(current$mean, current$total, ylim=c(0, 10))
  points(metadata(current)$mean, metadata(current)$var, col="red", pch=16)
  curve(metadata(current)$trend(x), add=TRUE, col="dodgerblue")
}
```

 modelGeneVarWithSpikes

Model the per-gene variance with spike-ins

Description

Model the variance of the log-expression profiles for each gene, decomposing it into technical and biological components based on a mean-variance trend fitted to spike-in transcripts.

Usage

```

modelGeneVarWithSpikes(x, ...)

## S4 method for signature 'ANY'
modelGeneVarWithSpikes(
  x,
  spikes,
  size.factors = NULL,
  spike.size.factors = NULL,
  block = NULL,
  design = NULL,
  subset.row = NULL,
  pseudo.count = 1,
  ...,
  equiweight = TRUE,
  method = "fisher",
  BPPARAM = SerialParam()
)

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

## S4 method for signature 'SingleCellExperiment'
modelGeneVarWithSpikes(
  x,
  spikes,
  size.factors = NULL,
  spike.size.factors = NULL,
  ...,
  assay.type = "counts"
)

```

Arguments

x	A numeric matrix of counts where rows are (usually endogenous) genes and columns are cells. Alternatively, a SummarizedExperiment or SingleCellExperiment containing such a matrix.
...	For the generic, further arguments to pass to each method. For the ANY method, further arguments to pass to fitTrendVar .

	For the SummarizedExperiment method, further arguments to pass to the ANY method.
	For the SingleCellExperiment method, further arguments to pass to the SummarizedExperiment method.
spikes	A numeric matrix of counts where each row corresponds to a spike-in transcript. This should have the same number of columns as <code>x</code> . Alternatively, for the SingleCellExperiment method, this can be a string or an integer scalar specifying the altExp containing the spike-in count matrix.
size.factors	A numeric vector of size factors for each cell in <code>x</code> , to be used for scaling gene expression.
spike.size.factors	A numeric vector of size factors for each cell in <code>spikes</code> , to be used for scaling spike-ins.
block	A factor specifying the blocking levels for each cell in <code>x</code> . If specified, variance modelling is performed separately within each block and statistics are combined across blocks.
design	A numeric matrix containing blocking terms for uninteresting factors of variation.
subset.row	See ?" scrn-gene-selection ", specifying the rows for which to model the variance. Defaults to all genes in <code>x</code> .
pseudo.count	Numeric scalar specifying the pseudo-count to add prior to log-transformation.
equiweight	A logical scalar indicating whether statistics from each block should be given equal weight. Otherwise, each block is weighted according to its number of cells. Only used if <code>block</code> is specified.
method	String specifying how p-values should be combined when <code>block</code> is specified, see combinePValues .
BPPARAM	A BiocParallelParam object indicating whether parallelization should be performed across genes.
assay.type	String or integer scalar specifying the assay containing the counts. For the SingleCellExperiment method, this is used to retrieve both the endogenous and spike-in counts.

Details

For each gene and spike-in transcript, we compute the variance and mean of the log-expression values. A trend is fitted to the variance against the mean for spike-in transcripts using [fitTrendVar](#). The technical component for each gene is defined as the value of the trend at that gene's mean abundance. The biological component is then defined as the residual from the trend.

This function is almost the same as [modelGeneVar](#), with the only difference being that the trend is fitted on spike-in variances rather than using the means and variances of endogenous genes. It assumes that a constant amount of spike-in RNA was added to each cell, such that any differences in observed expression of the spike-in transcripts can be wholly attributed to technical noise; and that endogenous genes and spike-in transcripts follow the same mean-variance relationship.

Unlike [modelGeneVar](#), [modelGeneVarWithSpikes](#) starts from a count matrix (for both genes and spike-ins) and requires size factors and a pseudo-count specification to compute the log-expression values. This is because there are certain requirements for how normalization is performed when comparing spike-in transcripts with endogenous genes - see comments in "Explanation for size factor rescaling". We enforce this by centering the size factors for both sets of features and recomputing the log-expression values prior to computing means and variances.

Value

A `DataFrame` is returned where each row corresponds to a gene in `x` (or in `subset.row`, if specified). This contains the numeric fields:

`mean`: Mean normalized log-expression per gene.

`total`: Variance of the normalized log-expression per gene.

`bio`: Biological component of the variance.

`tech`: Technical component of the variance.

`p.value`, `FDR`: Raw and adjusted p-values for the test against the null hypothesis that $\text{bio} \leq 0$.

If `block` is not specified, the metadata of the `DataFrame` contains the output of running `fitTrendVar` on the spike-in transcripts, along with the `mean` and `var` used to fit the trend.

If `block` is specified, the output contains another `per.block` field. This field is itself a `DataFrame` of `DataFrames`, where each internal `DataFrame` contains statistics for the variance modelling within each block and has the same format as described above. Each internal `DataFrame`'s metadata contains the output of `fitTrendVar` for the cells of that block.

Default size factor choices

If no size factors are supplied, they are automatically computed depending on the input type:

- If `size.factors=NULL` for the ANY method, the sum of counts for each cell in `x` is used to compute a size factor via the `librarySizeFactors` function.
- If `spike.size.factors=NULL` for the ANY method, the sum of counts for each cell in `spikes` is used to compute a size factor via the `librarySizeFactors` function.
- If `size.factors=NULL` for the `SingleCellExperiment` method, `sizeFactors(x)` is used if available. Otherwise, it defaults to library size-derived size factors.
- If `spike.size.factors=NULL` for the `SingleCellExperiment` method and `spikes` is not a matrix, `sizeFactors(altExp(x,spikes))` is used if available. Otherwise, it defaults to library size-derived size factors.

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

Explanation for size factor rescaling

The use of a spike-in-derived trend makes several assumptions. The first is that a constant amount of spike-in RNA was added to each cell, such that any differences in observed expression of the spike-in transcripts can be wholly attributed to technical noise. The second is that endogenous genes and spike-in transcripts follow the same mean-variance relationship, i.e., a spike-in transcript captures the technical noise of an endogenous gene at the same mean count.

Here, the spike-in size factors across all cells are scaled so that their mean is equal to that of the gene-based size factors for the same set of cells. This ensures that the average normalized abundances of the spike-in transcripts are comparable to those of the endogenous genes, allowing the trend fitted to the former to be used to determine the biological component of the latter. Otherwise, differences in scaling of the size factors would shift the normalized expression values of the former away from the latter, violating the second assumption.

If `block` is specified, rescaling is performed separately for all cells in each block. This aims to avoid problems from (frequent) violations of the first assumption where there are differences in the quantity of spike-in RNA added to each batch. Without scaling, these differences would lead to systematic shifts in the spike-in abundances from the endogenous genes when fitting a batch-specific trend (even if there is no global difference in scaling across all batches).

Computing p-values

The p-value for each gene is computed by assuming that the variance estimates are normally distributed around the trend, and that the standard deviation of the variance distribution is proportional to the value of the trend. This is used to construct a one-sided test for each gene based on its `bio`, under the null hypothesis that the biological component is equal to zero. The proportionality constant for the standard deviation is set to the `std.dev` returned by `fitTrendVar`. This is estimated from the spread of variance estimates for spike-in transcripts, so the null hypothesis effectively becomes “is this gene *more* variable than spike-in transcripts of the same abundance?”

Handling uninteresting factors

Setting `block` will estimate the mean and variance of each gene for cells in each level of `block` separately. The trend is fitted separately for each level, and the variance decomposition is also performed separately. Per-level statistics are then combined to obtain a single value per gene:

- For means and variance components, this is done by averaging values across levels. If `equiweight=FALSE`, a weighted average is used where the value for each level is weighted by the number of cells. By default, all levels are equally weighted when combining statistics.
- Per-level p-values are combined using `combinePValues` according to `method`. By default, Fisher’s method is used to identify genes that are highly variable in any batch. Whether or not this is responsive to `equiweight` depends on the chosen method.
- Blocks with fewer than 2 cells are completely ignored and do not contribute to the combined mean, variance component or p-value.

Use of `block` is the recommended approach for accounting for any uninteresting categorical factor of variation. In addition to accounting for systematic differences in expression between levels of the blocking factor, it also accommodates differences in the mean-variance relationships.

Alternatively, uninteresting factors can be used to construct a design matrix to pass to the function via `design`. In this case, a linear model is fitted to the expression profile for each gene and the residual variance is calculated. This approach is useful for covariates or additive models that cannot be expressed as a one-way layout for use in `block`. However, it assumes that the error is normally distributed with equal variance for all observations of a given gene.

Use of `block` and `design` together is currently not supported and will lead to an error.

Author(s)

Aaron Lun

See Also

[fitTrendVar](#), for the trend fitting options.

[modelGeneVar](#), for modelling variance without spike-in controls.

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Using spike-ins.
spk <- modelGeneVarWithSpikes(sce, "Spikes")
spk
```

```

plot(spik$mean, spik$total, log="xy")
points(metadata(spik)$mean, metadata(spik)$cv2, col="red", pch=16)
curve(metadata(spik)$trend(x), add=TRUE, col="dodgerblue")

# With blocking (and spike-ins).
block <- sample(LETTERS[1:2], ncol(sce), replace=TRUE)
blk <- modelGeneVarWithSpikes(sce, "Spikes", block=block)
blk

par(mfrow=c(1,2))
for (i in colnames(blk$per.block)) {
  current <- blk$per.block[[i]]
  plot(current$mean, current$total)
  points(metadata(current)$mean, metadata(current)$cv2, col="red", pch=16)
  curve(metadata(current)$trend(x), add=TRUE, col="dodgerblue")
}

```

multiMarkerStats

Combine multiple sets of marker statistics

Description

Combine multiple sets of marker statistics, typically from different tests, into a single [DataFrame](#) for convenient inspection.

Usage

```
multiMarkerStats(..., repeated = NULL, sorted = TRUE)
```

Arguments

...	Two or more lists or Lists produced by findMarkers or combineMarkers . Each list should contain DataFrames of results, one for each group/cluster of cells. The names of each List should be the same; the universe of genes in each DataFrame should be the same; and the same number of columns in each DataFrame should be named. All elements in ... are also expected to be named.
repeated	Character vector of columns that are present in one or more DataFrames but should only be reported once. Typically used to avoid reporting redundant copies of annotation-related columns.
sorted	Logical scalar indicating whether each output DataFrame should be sorted by some relevant statistic.

Details

The combined statistics are designed to favor a gene that is highly ranked in each of the individual test results. This is highly conservative and aims to identify robust DE that is significant under all testing schemes.

A combined Top value of T indicates that the gene is among the top T genes of one or more pairwise comparisons in each of the testing schemes. (We can be even more aggressive if the individual results were generated with a larger min.prop value.) In effect, a gene can only achieve a low Top

value if it is consistently highly ranked in each test. If `sorted=TRUE`, this is used to order the genes in the output `DataFrame`.

The combined `p.value` is effectively the result of applying an intersection-union test to the per-test results. This will only be low if the gene has a low `p-value` in each of the test results. If `sorted=TRUE` and `Top` is not present, this will be used to order the genes in the output `DataFrame`.

Value

A named List of `DataFrames` with one `DataFrame` per group/cluster. Each `DataFrame` contains statistics from the corresponding entry of each List in `...`, prefixed with the name of the List. In addition, several combined statistics are reported:

- `Top`, the largest rank of each gene across all `DataFrames` for that group. This is only reported if each list in `...` was generated with `pval.type="any"` in `combineMarkers`.
- `p.value`, the largest `p-value` of each gene across all `DataFrames` for that group. This is replaced by `log.p.value` if `p-values` in `...` are log-transformed.
- `FDR`, the BH-adjusted value of `p.value`. This is replaced by `log.FDR` if `p-values` in `...` are log-transformed.

Author(s)

Aaron Lun

See Also

[findMarkers](#) and [combineMarkers](#), to generate elements in `...`

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Any clustering method is okay, only using k-means for convenience.
kout <- kmeans(t(logcounts(sce)), centers=4)

tout <- findMarkers(sce, groups=kout$cluster, direction="up")
wout <- findMarkers(sce, groups=kout$cluster, direction="up", test="wilcox")

combined <- multiMarkerStats(t=tout, wilcox=wout)
colnames(combined[[1]])
```

pairwiseBinom

Perform pairwise binomial tests

Description

Perform pairwise binomial tests between groups of cells, possibly after blocking on uninteresting factors of variation.

Usage

```
pairwiseBinom(x, ...)

## S4 method for signature 'ANY'
pairwiseBinom(
  x,
  groups,
  block = NULL,
  restrict = NULL,
  exclude = NULL,
  direction = c("any", "up", "down"),
  threshold = 1e-08,
  lfc = 0,
  log.p = FALSE,
  gene.names = NULL,
  subset.row = NULL,
  BPPARAM = SerialParam()
)

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

## S4 method for signature 'SingleCellExperiment'
pairwiseBinom(x, groups = colLabels(x, onAbsence = "error"), ...)
```

Arguments

x	A numeric matrix-like object of counts, where each column corresponds to a cell and each row corresponds to a gene.
...	For the generic, further 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.
groups	A vector of length equal to ncol(x), specifying the group assignment for each cell. If x is a SingleCellExperiment, this is automatically derived from colLabels .
block	A factor specifying the blocking level for each cell.
restrict	A vector specifying the levels of groups for which to perform pairwise comparisons.
exclude	A vector specifying the levels of groups for which <i>not</i> to perform pairwise comparisons.
direction	A string specifying the direction of effects to be considered for the alternative hypothesis.
threshold	Numeric scalar specifying the value below which a gene is presumed to be not expressed.
lfc	Numeric scalar specifying the minimum absolute log-ratio in the proportion of expressing genes between groups.
log.p	A logical scalar indicating if log-transformed p-values/FDRs should be returned.
gene.names	Deprecated, use row.data in findMarkers instead to add custom annotation.

subset.row	See ?"scran-gene-selection".
BPPARAM	A BiocParallelParam object indicating whether and how parallelization should be performed across genes.
assay.type	A string specifying which assay values to use, usually "logcounts".

Details

This function performs exact binomial tests to identify marker genes between pairs of groups of cells. Here, the null hypothesis is that the proportion of cells expressing a gene is the same between groups. A list of tables is returned where each table contains the statistics for all genes for a comparison between each pair of groups. This can be examined directly or used as input to [combineMarkers](#) for marker gene detection.

Effect sizes for each comparison are reported as log₂-fold changes in the proportion of expressing cells in one group over the proportion in another group. We add a pseudo-count that squeezes the log-FCs towards zero to avoid undefined values when one proportion is zero. This is closely related to but somewhat more interpretable than the log-odds ratio, which would otherwise be the more natural statistic for a proportion-based test.

If `restrict` is specified, comparisons are only performed between pairs of groups in `restrict`. This can be used to focus on DEGs distinguishing between a subset of the groups (e.g., closely related cell subtypes). Similarly, if any entries of groups are NA, the corresponding cells are ignored.

`x` can be a count matrix or any transformed counterpart where zeroes remain zero and non-zeroes remain non-zero. This is true of any scaling normalization and monotonic transformation like the log-transform. If the transformation breaks this rule, some adjustment of `threshold` is necessary.

A consequence of the transformation-agnostic behaviour of this function is that it will not respond to normalization. Differences in library size will not be considered by this function. However, this is not necessarily problematic for marker gene detection - users can treat this as *retaining* information about the total RNA content, analogous to spike-in normalization.

Value

A list is returned containing `statistics` and `pairs`.

The `statistics` element is itself a list of [DataFrames](#). Each `DataFrame` contains the statistics for a comparison between a pair of groups, including the overlap proportions, p-values and false discovery rates.

The `pairs` element is a `DataFrame` with one row corresponding to each entry of `statistics`. This contains the fields `first` and `second`, specifying the two groups under comparison in the corresponding `DataFrame` in `statistics`.

In each `DataFrame` in `statistics`, the log-fold change represents the log-ratio of the proportion of expressing cells in the first group compared to the expressing proportion in the second group.

Direction and magnitude of the effect

If `direction="any"`, two-sided binomial tests will be performed for each pairwise comparisons between groups of cells. For other `direction`, one-sided tests in the specified direction will be used instead. This can be used to focus on genes that are upregulated in each group of interest, which is often easier to interpret.

In practice, the two-sided test is approximated by combining two one-sided tests using a Bonferroni correction. This is done for various logistical purposes; it is also the only way to combine p-values across blocks in a direction-aware manner. As a result, the two-sided p-value reported here will not

be the same as that from `binom.test`. In practice, they are usually similar enough that this is not a major concern.

To interpret the setting of `direction`, consider the DataFrame for group X, in which we are comparing to another group Y. If `direction="up"`, genes will only be significant in this DataFrame if they are upregulated in group X compared to Y. If `direction="down"`, genes will only be significant if they are downregulated in group X compared to Y. See `?binom.test` for more details on the interpretation of one-sided Wilcoxon rank sum tests.

The magnitude of the log-fold change in the proportion of expressing cells can also be tested by setting `lfc`. By default, `lfc=0` meaning that we will reject the null upon detecting any difference in proportions. If this is set to some other positive value, the null hypothesis will change depending on `direction`:

- If `direction="any"`, the null hypothesis is that the true log-fold change in proportions is either $-lfc$ or lfc with equal probability. A two-sided p-value is computed against this composite null.
- If `direction="up"`, the null hypothesis is that the true log-fold change is lfc , and a one-sided p-value is computed.
- If `direction="down"`, the null hypothesis is that the true log-fold change is $-lfc$, and a one-sided p-value is computed.

Blocking on uninteresting factors

If `block` is specified, binomial tests are performed between groups of cells within each level of `block`. For each pair of groups, the p-values for each gene across all levels of `block` are combined using Stouffer's weighted Z-score method.

The weight for the p-value in a particular level of `block` is defined as $N_x + N_y$, where N_x and N_y are the number of cells in groups X and Y, respectively, for that level. This means that p-values from blocks with more cells will have a greater contribution to the combined p-value for each gene.

When combining across batches, one-sided p-values in the same direction are combined first. Then, if `direction="any"`, the two combined p-values from both directions are combined. This ensures that a gene only receives a low overall p-value if it changes in the same direction across batches.

When comparing two groups, blocking levels are ignored if no p-value was reported, e.g., if there were insufficient cells for a group in a particular level. If all levels are ignored in this manner, the entire comparison will only contain NA p-values and a warning will be emitted.

Author(s)

Aaron Lun

References

Whitlock MC (2005). Combining probability from independent tests: the weighted Z-method is superior to Fisher's approach. *J. Evol. Biol.* 18, 5:1368-73.

See Also

`binom.test` and `binomTest`, on which this function is based.

`combineMarkers`, to combine pairwise comparisons into a single DataFrame per group.

`getTopMarkers`, to obtain the top markers from each pairwise comparison.

Examples

```

library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Any clustering method is okay.
kout <- kmeans(t(logcounts(sce)), centers=2)

# Vanilla application:
out <- pairwiseBinom(logcounts(sce), groups=kout$cluster)
out

# Directional and with a minimum log-fold change:
out <- pairwiseBinom(logcounts(sce), groups=kout$cluster,
  direction="up", lfc=1)
out

```

pairwiseTTests	<i>Perform pairwise t-tests</i>
----------------	---------------------------------

Description

Perform pairwise Welch t-tests between groups of cells, possibly after blocking on uninteresting factors of variation.

Usage

```

pairwiseTTests(x, ...)

## S4 method for signature 'ANY'
pairwiseTTests(
  x,
  groups,
  block = NULL,
  design = NULL,
  restrict = NULL,
  exclude = NULL,
  direction = c("any", "up", "down"),
  lfc = 0,
  std.lfc = FALSE,
  log.p = FALSE,
  gene.names = NULL,
  subset.row = NULL,
  BPPARAM = SerialParam()
)

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

## S4 method for signature 'SingleCellExperiment'
pairwiseTTests(x, groups = colLabels(x, onAbsence = "error"), ...)

```

Arguments

x	A numeric matrix-like object of normalized log-expression values, where each column corresponds to a cell and each row corresponds to an endogenous gene. Alternatively, a SummarizedExperiment or SingleCellExperiment object containing such a matrix.
...	For the generic, further 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.
groups	A vector of length equal to <code>ncol(x)</code> , specifying the group assignment for each cell. If x is a SingleCellExperiment , this is automatically derived from collLabels .
block	A factor specifying the blocking level for each cell.
design	A numeric matrix containing blocking terms for uninteresting factors. Note that these factors should not be confounded with groups.
restrict	A vector specifying the levels of groups for which to perform pairwise comparisons.
exclude	A vector specifying the levels of groups for which <i>not</i> to perform pairwise comparisons.
direction	A string specifying the direction of log-fold changes to be considered in the alternative hypothesis.
lfc	A positive numeric scalar specifying the log-fold change threshold to be tested against.
std.lfc	A logical scalar indicating whether log-fold changes should be standardized.
log.p	A logical scalar indicating if log-transformed p-values/FDRs should be returned.
gene.names	Deprecated, use <code>row.data</code> in findMarkers instead to add custom annotation.
subset.row	See ?" scran-gene-selection ".
BPPARAM	A BiocParallelParam object indicating whether and how parallelization should be performed across genes.
assay.type	A string specifying which assay values to use, usually "logcounts".

Details

This function performs t-tests to identify differentially expressed genes (DEGs) between pairs of groups of cells. A typical aim is to use the DEGs to determine cluster identity based on expression of marker genes with known biological activity. A list of tables is returned where each table contains per-gene statistics for a comparison between one pair of groups. This can be examined directly or used as input to [combineMarkers](#) for marker gene detection.

We use t-tests as they are simple, fast and perform reasonably well for single-cell data (Soneson and Robinson, 2018). However, if one of the groups contains fewer than two cells, no p-value will be reported for comparisons involving that group. A warning will also be raised about insufficient degrees of freedom (d.f.) in such cases.

When `log.p=TRUE`, the log-transformed p-values and FDRs are reported using the natural base. This is useful in cases with many cells such that reporting the p-values directly would lead to double-precision underflow.

If `restrict` is specified, comparisons are only performed between pairs of groups in `restrict`. This can be used to focus on DEGs distinguishing between a subset of the groups (e.g., closely related cell subtypes).

If `exclude` is specified, comparisons are not performed between groups in `exclude`. Similarly, if any entries of groups are NA, the corresponding cells are ignored.

Value

A list is returned containing `statistics` and `pairs`.

The `statistics` element is itself a list of `DataFrames`. Each `DataFrame` contains the statistics for a comparison between a pair of groups, including the log-fold changes, p-values and false discovery rates.

The `pairs` element is a `DataFrame` where each row corresponds to an entry of `statistics`. This contains the fields `first` and `second`, specifying the two groups under comparison in the corresponding `DataFrame` in `statistics`.

In each `DataFrame` in `statistics`, the log-fold change represents the change in the first group compared to the second group.

Direction and magnitude of the log-fold change

Log-fold changes are reported as differences in the values of `x`. Thus, all log-fold changes have the same base as whatever was used to perform the log-transformation in `x`. If `logNormCounts` was used, this would be base 2.

If `direction="any"`, two-sided tests will be performed for each pairwise comparisons between groups. Otherwise, one-sided tests in the specified direction will be used instead. This can be used to focus on genes that are upregulated in each group of interest, which is often easier to interpret when assigning cell type to a cluster.

To interpret the setting of `direction`, consider the `DataFrame` for group `X`, in which we are comparing to another group `Y`. If `direction="up"`, genes will only be significant in this `DataFrame` if they are upregulated in group `X` compared to `Y`. If `direction="down"`, genes will only be significant if they are downregulated in group `X` compared to `Y`.

The magnitude of the log-fold changes can also be tested by setting `lfc`. By default, `lfc=0` meaning that we will reject the null upon detecting any differential expression. If this is set to some other positive value, the null hypothesis will change depending on `direction`:

- If `direction="any"`, the null hypothesis is that the true log-fold change is either $-lfc$ or lfc with equal probability. A two-sided p-value is computed against this composite null.
- If `direction="up"`, the null hypothesis is that the true log-fold change is lfc , and a one-sided p-value is computed.
- If `direction="down"`, the null hypothesis is that the true log-fold change is $-lfc$, and a one-sided p-value is computed.

This is similar to the approach used in `treat` and allows users to focus on genes with strong log-fold changes.

If `std.lfc=TRUE`, the log-fold change for each gene is standardized by the variance. When the Welch t-test is being used, this is equivalent to Cohen's `d`. Standardized log-fold changes may be more appealing for visualization as it avoids large fold changes due to large variance. The choice of `std.lfc` does not affect the calculation of the p-values.

Blocking on uninteresting factors

If `block` is specified, Welch t-tests are performed between groups within each level of `block`. For each pair of groups, the p-values for each gene across all levels of `block` are combined using Stouffer's weighted Z-score method. The reported log-fold change for each gene is also a weighted average of log-fold changes across levels.

The weight for a particular level is defined as $(1/N_x + 1/N_y)^{-1}$, where N_x and N_y are the number of cells in groups X and Y, respectively, for that level. This is inversely proportional to the expected variance of the log-fold change, provided that all groups and blocking levels have the same variance.

When comparing two groups, blocking levels are ignored if no p-value was reported, e.g., if there were insufficient cells for a group in a particular level. This includes levels that contain fewer than two cells for either group, as this cannot yield a p-value from the Welch t-test. If all levels are ignored in this manner, the entire comparison will only contain NA p-values and a warning will be emitted.

Regressing out unwanted factors

If `design` is specified, a linear model is instead fitted to the expression profile for each gene. This linear model will include the groups as well as any blocking factors in `design`. A t-test is then performed to identify DEGs between pairs of groups, using the values of the relevant coefficients and the gene-wise residual variance.

Note that `design` must be full rank when combined with the groups terms, i.e., there should not be any confounding variables. We make an exception for the common situation where `design` contains an "(Intercept)" column, which is automatically detected and removed (emitting a warning along the way).

We recommend using `block` instead of `design` for uninteresting categorical factors of variation. The former accommodates differences in the variance of expression in each group via Welch's t-test. As a result, it is more robust to misspecification of the groups, as misspecified groups (and inflated variances) do not affect the inferences for other groups. Use of `block` also avoids assuming additivity of effects between the blocking factors and the groups.

Nonetheless, use of `design` is unavoidable when blocking on real-valued covariates. It is also useful for ensuring that log-fold changes/p-values are computed for comparisons between all pairs of groups (assuming that `design` is not confounded with the groups). This may not be the case with `block` if a pair of groups never co-occur in a single blocking level.

Author(s)

Aaron Lun

References

Whitlock MC (2005). Combining probability from independent tests: the weighted Z-method is superior to Fisher's approach. *J. Evol. Biol.* 18, 5:1368-73.

Soneson C and Robinson MD (2018). Bias, robustness and scalability in single-cell differential expression analysis. *Nat. Methods*

Lun ATL (2018). Comments on marker detection in *scran*. https://l1tla.github.io/SingleCellThoughts/software/marker_detection/comments.html

See Also

`t.test`, on which this function is based.

`combineMarkers`, to combine pairwise comparisons into a single DataFrame per group.
`getTopMarkers`, to obtain the top markers from each pairwise comparison.

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Any clustering method is okay.
kout <- kmeans(t(logcounts(sce)), centers=3)

# Vanilla application:
out <- pairwiseTTests(logcounts(sce), groups=kout$cluster)
out

# Directional with log-fold change threshold:
out <- pairwiseTTests(logcounts(sce), groups=kout$cluster,
  direction="up", lfc=0.2)
out
```

pairwiseWilcox

Perform pairwise Wilcoxon rank sum tests

Description

Perform pairwise Wilcoxon rank sum tests between groups of cells, possibly after blocking on uninteresting factors of variation.

Usage

```
pairwiseWilcox(x, ...)

## S4 method for signature 'ANY'
pairwiseWilcox(
  x,
  groups,
  block = NULL,
  restrict = NULL,
  exclude = NULL,
  direction = c("any", "up", "down"),
  lfc = 0,
  log.p = FALSE,
  gene.names = NULL,
  subset.row = NULL,
  BPPARAM = SerialParam()
)

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

## S4 method for signature 'SingleCellExperiment'
pairwiseWilcox(x, groups = colLabels(x, onAbsence = "error"), ...)
```

Arguments

x	A numeric matrix-like object of normalized (and possibly log-transformed) expression values, where each column corresponds to a cell and each row corresponds to an endogenous gene. Alternatively, a SummarizedExperiment or SingleCellExperiment object containing such a matrix.
...	For the generic, further 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.
groups	A vector of length equal to <code>ncol(x)</code> , specifying the group assignment for each cell. If x is a SingleCellExperiment , this is automatically derived from colLabels .
block	A factor specifying the blocking level for each cell.
restrict	A vector specifying the levels of groups for which to perform pairwise comparisons.
exclude	A vector specifying the levels of groups for which <i>not</i> to perform pairwise comparisons.
direction	A string specifying the direction of differences to be considered in the alternative hypothesis.
lfc	Numeric scalar specifying the minimum log-fold change for one observation to be considered to be “greater” than another.
log.p	A logical scalar indicating if log-transformed p-values/FDRs should be returned.
gene.names	Deprecated, use <code>row.data</code> in findMarkers instead to add custom annotation.
subset.row	See ?“ scrans-gene-selection ”.
BPPARAM	A BiocParallelParam object indicating whether and how parallelization should be performed across genes.
assay.type	A string specifying which assay values to use, usually “logcounts”.

Details

This function performs Wilcoxon rank sum tests to identify differentially expressed genes (DEGs) between pairs of groups of cells. A list of tables is returned where each table contains the statistics for all genes for a comparison between each pair of groups. This can be examined directly or used as input to [combineMarkers](#) for marker gene detection.

The effect size for each gene in each comparison is reported as the area under the curve (AUC). Consider the distribution of expression values for gene X within each of two groups A and B. The AUC is the probability that a randomly selected cell in A has a greater expression of X than a randomly selected cell in B. (Ties are assumed to be randomly broken.) Concordance probabilities near 0 indicate that most observations in A are lower than most observations in B; conversely, probabilities near 1 indicate that most observations in A are higher than those in B. The Wilcoxon rank sum test effectively tests for significant deviations from an AUC of 0.5.

Wilcoxon rank sum tests are more robust to outliers and insensitive to non-normality, in contrast to t-tests in [pairwiseTTests](#). However, they take longer to run, the effect sizes are less interpretable, and there are more subtle violations of its assumptions in real data. For example, the i.i.d. assumptions are unlikely to hold after scaling normalization due to differences in variance. Also note that we approximate the distribution of the Wilcoxon rank sum statistic to deal with large numbers of cells and ties.

If `restrict` is specified, comparisons are only performed between pairs of groups in `restrict`. This can be used to focus on DEGs distinguishing between a subset of the groups (e.g., closely related cell subtypes).

If `exclude` is specified, comparisons are not performed between groups in `exclude`. Similarly, if any entries of groups are NA, the corresponding cells are ignored.

Value

A list is returned containing statistics and pairs.

The `statistics` element is itself a list of [DataFrames](#). Each DataFrame contains the statistics for a comparison between a pair of groups, including the AUCs, p-values and false discovery rates.

The `pairs` element is a DataFrame with one row corresponding to each entry of `statistics`. This contains the fields `first` and `second`, specifying the two groups under comparison in the corresponding DataFrame in `statistics`.

In each DataFrame in `statistics`, the AUC represents the probability of sampling a value in the first group greater than a random value from the second group.

Direction and magnitude of the effect

If `direction="any"`, two-sided Wilcoxon rank sum tests will be performed for each pairwise comparisons between groups of cells. Otherwise, one-sided tests in the specified direction will be used instead. This can be used to focus on genes that are upregulated in each group of interest, which is often easier to interpret.

To interpret the setting of `direction`, consider the DataFrame for group X, in which we are comparing to another group Y. If `direction="up"`, genes will only be significant in this DataFrame if they are upregulated in group X compared to Y. If `direction="down"`, genes will only be significant if they are downregulated in group X compared to Y. See [?wilcox.test](#) for more details on the interpretation of one-sided Wilcoxon rank sum tests.

Users can also specify a log-fold change threshold in `lfc` to focus on genes that exhibit large shifts in location. This is equivalent to specifying the `mu` parameter in [wilcox.test](#), with some additional subtleties depending on `direction`:

- If `direction="any"`, the null hypothesis is that the true shift is either $-lfc$ or lfc with equal probability. A two-sided p-value is computed against this composite null. The AUC is computed by averaging the AUCs obtained when X's expression values are shifted to the left or right by lfc . This can be very roughly interpreted as considering an observation of Y to be tied with an observation of X if they differ by less than lfc .
- If `direction="up"`, the null hypothesis is that the true shift is lfc , and a one-sided p-value is computed. The AUC is computed after shifting X's expression values to the left by lfc .
- If `direction="down"`, the null hypothesis is that the true shift is $-lfc$, and a one-sided p-value is computed. The AUC is computed after shifting X's expression values to the right by lfc .

The fact that the AUC depends on `lfc` is necessary to preserve its relationship with the p-value.

Blocking on uninteresting factors

If `block` is specified, Wilcoxon tests are performed between groups of cells within each level of `block`. For each pair of groups, the p-values for each gene across all levels of `block` are combined using Stouffer's Z-score method. The reported AUC is also a weighted average of the AUCs across all levels.

The weight for a particular level of block is defined as $N_x N_y$, where N_x and N_y are the number of cells in groups X and Y, respectively, for that level. This means that p-values from blocks with more cells will have a greater contribution to the combined p-value for each gene.

When combining across batches, one-sided p-values in the same direction are combined first. Then, if `direction="any"`, the two combined p-values from both directions are combined. This ensures that a gene only receives a low overall p-value if it changes in the same direction across batches.

When comparing two groups, blocking levels are ignored if no p-value was reported, e.g., if there were insufficient cells for a group in a particular level. If all levels are ignored in this manner, the entire comparison will only contain NA p-values and a warning will be emitted.

Author(s)

Aaron Lun

References

Whitlock MC (2005). Combining probability from independent tests: the weighted Z-method is superior to Fisher's approach. *J. Evol. Biol.* 18, 5:1368-73.

Soneson C and Robinson MD (2018). Bias, robustness and scalability in single-cell differential expression analysis. *Nat. Methods*

See Also

[wilcox.test](#), on which this function is based.

[combineMarkers](#), to combine pairwise comparisons into a single DataFrame per group.

[getTopMarkers](#), to obtain the top markers from each pairwise comparison.

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Any clustering method is okay.
kout <- kmeans(t(logcounts(sce)), centers=2)

# Vanilla application:
out <- pairwiseWilcox(logcounts(sce), groups=kout$cluster)
out

# Directional and with a minimum log-fold change:
out <- pairwiseWilcox(logcounts(sce), groups=kout$cluster,
  direction="up", lfc=0.2)
out
```

pseudoBulkDGE

*Quickly perform pseudo-bulk DE analyses***Description**

A wrapper function around **edgeR**'s quasi-likelihood methods to conveniently perform differential expression analyses on pseudo-bulk profiles, allowing detection of cell type-specific changes between conditions in replicated studies.

Usage

```
pseudoBulkDGE(x, ...)
```

```
## S4 method for signature 'ANY'
```

```
pseudoBulkDGE(
  x,
  col.data,
  label,
  design,
  coef,
  contrast = NULL,
  condition = NULL,
  lfc = 0,
  include.intermediates = TRUE,
  row.data = NULL,
  sorted = FALSE,
  method = c("edgeR", "voom"),
  qualities = TRUE,
  robust = TRUE,
  sample = NULL
)
```

```
## S4 method for signature 'SummarizedExperiment'
```

```
pseudoBulkDGE(x, col.data = colData(x), ..., assay.type = 1)
```

Arguments

<code>x</code>	A numeric matrix of counts where rows are genes and columns are pseudo-bulk profiles. Alternatively, a <code>SummarizedExperiment</code> object containing such a matrix in its assays.
<code>...</code>	For the generic, additional arguments to pass to individual methods. For the <code>SummarizedExperiment</code> method, additional arguments to pass to the ANY method.
<code>col.data</code>	A data.frame or DataFrame containing metadata for each column of <code>x</code> .
<code>label</code>	A vector of factor of length equal to <code>ncol(x)</code> , specifying the cluster or cell type assignment for each column of <code>x</code> .
<code>design</code>	A formula to be used to construct a design matrix from variables in <code>col.data</code> . Alternatively, a function that accepts a data.frame with the same fields as <code>col.data</code> and returns a design matrix.

<code>coef</code>	String or character vector containing the coefficients to drop from the design matrix to form the null hypothesis. Can also be an integer scalar or vector specifying the indices of the relevant columns.
<code>contrast</code>	Numeric vector or matrix containing the contrast of interest. Alternatively, a character vector to be passed to <code>makeContrasts</code> to create this numeric vector/matrix. If specified, this takes precedence over <code>coef</code> .
<code>condition</code>	A vector or factor of length equal to <code>ncol(x)</code> , specifying the experimental condition for each column of <code>x</code> . Only used for abundance-based filtering of genes.
<code>lfc</code>	Numeric scalar specifying the log-fold change threshold to use in <code>glmTreat</code> or <code>treat</code> .
<code>include.intermediates</code>	Logical scalar indicating whether the intermediate edgeR objects should be returned.
<code>row.data</code>	A DataFrame containing additional row metadata for each gene in <code>x</code> , to be included in each of the output DataFrames .
<code>sorted</code>	Logical scalar indicating whether the output tables should be sorted by p-value.
<code>method</code>	String specifying the DE analysis framework to use.
<code>qualities</code>	Logical scalar indicating whether quality weighting should be used when <code>method="voom"</code> , see <code>voomWithQualityWeights</code> for more details.
<code>robust</code>	Logical scalar indicating whether robust empirical Bayes shrinkage should be performed.
<code>sample</code>	Deprecated.
<code>assay.type</code>	String or integer scalar specifying the assay to use from <code>x</code> .

Details

In replicated multi-condition scRNA-seq experiments, we often have clusters comprised of cells from different samples of different experimental conditions. It is often desirable to check for differential expression between conditions within each cluster, allowing us to identify cell-type-specific responses to the experimental perturbation.

Given a set of pseudo-bulk profiles (usually generated by `sumCountsAcrossCells`), this function loops over the labels and uses **edgeR** or `voom` to detect DE genes between conditions. The DE analysis for each label is largely the same as a standard analysis for bulk RNA-seq data, using design and `coef` or `contrast` as described in the **edgeR** or **limma** user guides. Generally speaking, **edgeR** handles low counts better via its count-based model but `method="voom"` supports variable sample precision when `quality=TRUE`.

Performing pseudo-bulk DGE enables us to reuse well-tested methods developed for bulk RNA-seq data analysis. Each pseudo-bulk profile can be treated as an *in silico* mimicry of a real bulk RNA-seq sample (though in practice, it tends to be much more variable due to the lower numbers of cells). This also models the relevant variability between experimental replicates (i.e., across samples) rather than that between cells in the same sample, without resorting to expensive mixed-effects models.

The DE analysis for each label is independent of that for any other label. This aims to minimize problems due to differences in abundance and variance between labels, at the cost of losing the ability to share information across labels.

In some cases, it will be impossible to perform a DE analysis for a label. The most obvious reason is if there are no residual degrees of freedom; other explanations include impossible contrasts or a failure to construct an appropriate design matrix (e.g., if a cell type only exists in one condition).

Note that we assume that `x` has already been filtered to remove unstable pseudo-bulk profiles generated from few cells.

Value

A `List` with one `DataFrame` of DE results per unique (non-failed) level of `cluster`. This contains columns from `topTags` if `method="edgeR"` or `topTable` if `method="voom"`. All `DataFrames` have row names equal to `rownames(x)`.

The `metadata` of the `List` contains `failed`, a character vector with the names of the labels for which the comparison could not be performed - see `Details`.

The `metadata` of the individual `DataFrames` contains `design`, the final design matrix for that label. If `include.intermediates`, the `metadata` will also contain `y`, the `DGEList` used for the analysis; and `fit`, the `DGEGLM` object after GLM fitting.

Comments on abundance filtering

For each label, abundance filtering is performed using `filterByExpr` prior to further analysis. Genes that are filtered out will still show up in the `DataFrame` for that label, but with all statistics set to `NA`. As this is done separately for each label, a different set of genes may be filtered out for each label, which is largely to be expected if there is any label-specific expression.

By default, the minimum group size for `filterByExpr` is determined using the design matrix. However, this may not be optimal if the design matrix contains additional terms (e.g., blocking factors) in which case it is not easy to determine the minimum size of the groups relevant to the comparison of interest. To overcome this, users can specify `condition.field` to specify the group to which each sample belongs, which is used by `filterByExpr` to obtain a more appropriate minimum group size.

Author(s)

Aaron Lun

References

Tung P-Y et al. (2017). Batch effects and the effective design of single-cell gene expression studies. *Sci. Rep.* 7, 39921

Lun ATL and Marioni JC (2017). Overcoming confounding plate effects in differential expression analyses of single-cell RNA-seq data. *Biostatistics* 18, 451-464

Crowell HL et al. (2019). On the discovery of population-specific state transitions from multi-sample multi-condition single-cell RNA sequencing data. *bioRxiv*

See Also

`sumCountsAcrossCells`, to easily generate the pseudo-bulk count matrix.

`decideTestsPerLabel`, to generate a summary of the DE results across all labels.

`pseudoBulkSpecific`, to look for label-specific DE genes.

`pbDS` from the `muscat` package, which uses a similar approach.

Examples

```
set.seed(10000)
library(scuttle)
sce <- mockSCE(ncells=1000)
sce$samples <- gl(8, 125) # Pretending we have 8 samples.

# Making up some clusters.
```

```

sce <- logNormCounts(sce)
clusters <- kmeans(t(logcounts(sce)), centers=3)$cluster

# Creating a set of pseudo-bulk profiles:
info <- DataFrame(sample=sce$samples, cluster=clusters)
pseudo <- sumCountsAcrossCells(sce, info)

# Making up an experimental design for our 8 samples.
pseudo$DRUG <- gl(2,4)[pseudo$sample]

# DGE analysis:
out <- pseudoBulkDGE(pseudo,
  label=pseudo$cluster,
  condition=pseudo$DRUG,
  design=~DRUG,
  coef="DRUG2"
)
out[[1]]
metadata(out[[1]])$design

```

pseudoBulkSpecific *Label-specific pseudo-bulk DE*

Description

Detect label-specific DE genes in a pseudo-bulk analysis, by testing whether the log-fold change is more extreme than the average log-fold change of other labels.

Usage

```

pseudoBulkSpecific(x, ...)

## S4 method for signature 'ANY'
pseudoBulkSpecific(
  x,
  label,
  condition = NULL,
  ...,
  method = c("edgeR", "voom"),
  sorted = FALSE,
  average = c("median", "mean"),
  missing.as.zero = FALSE,
  reference = NULL
)

## S4 method for signature 'SummarizedExperiment'
pseudoBulkSpecific(x, ..., assay.type = 1)

```

Arguments

x A numeric matrix of counts where rows are genes and columns are pseudo-bulk profiles. Alternatively, a SummarizedExperiment object containing such a matrix in its assays.

...	For the generic, further arguments to pass to individual methods. For the ANY method, further arguments to pass to <code>pseudoBulkDGE</code> . For the SummarizedExperiment method, further arguments to pass to the ANY method.
label	A vector of factor of length equal to <code>ncol(x)</code> , specifying the cluster or cell type assignment for each column of <code>x</code> .
condition	A vector or factor of length equal to <code>ncol(x)</code> , specifying the experimental condition for each column of <code>x</code> . Only used for abundance-based filtering of genes.
method	String specifying the DE analysis framework to use.
sorted	Logical scalar indicating whether the output tables should be sorted by p-value.
average	String specifying the method to compute the average log-fold change of all other labels.
missing.as.zero	Logical scalar indicating whether missing log-fold changes should be set to zero.
reference	A List containing the (unsorted) output of <code>pseudoBulkDGE</code> . This can be supplied to avoid redundant calculations but is automatically computed if NULL.
assay.type	String or integer scalar specifying the assay to use from <code>x</code> .

Details

This function implements a quick and dirty method for detecting label-specific DE genes. For a given label and gene, the null hypothesis is that the log-fold change lies between zero and the average log-fold change for that gene across all other labels. Genes that reject this null either have log-fold changes in the opposite sign or are significantly more extreme than the average.

To implement this, we test each gene against the two extremes and taking the larger of the two p-values. The p-value is set to 1 if the log-fold change lies between the extremes. This is somewhat similar to how `treat` might behave if the null interval was not centered at zero; however, our approach is more conservative than the `treat` as the p-value calculations are not quite correct.

It is worth stressing that there are no guarantees that the DE genes detected in this manner are truly label-specific. For any label and DEG, there may be one or more labels with stronger log-fold changes, but the average may be pulled towards zero by other labels with weaker or opposing effects. The use of the average is analogous to recommendations in the `edgeR` user's guide for testing against multiple groups. However, a more stringent selection can be achieved by applying gates on `decideTestsPerLabel`.

Note that, if `lfc` is specified in the arguments to `pseudoBulkDGE`, the null interval is expanded in both directions by the specified value.

Value

A List of DataFrames where each DataFrame contains DE statistics for one label. This is equivalent to the output of `pseudoBulkDGE`; if `reference` is supplied, most of the statistics will be identical to those reported there.

The main differences are that the p-values and FDR values are changed. Each DataFrame also has an `OtherAverage` field containing the average log-fold change across all other labels.

Computing the average

The average log-fold change for each gene is computed by taking the median or mean (depending on `average`) of the corresponding log-fold changes in each of the DE analyses for the other labels.

We use the median by default as this means that at least half of all other labels should have weaker or opposing effects.

By default, low-abundance genes that were filtered out in a comparison do not contribute to the average. Any log-fold changes that could be computed from them are considered to be too unstable. If the gene is filtered out in all other labels, the average is set to zero for testing but is reported as NA.

If `missing.as.zero=TRUE`, the log-fold changes for all filtered genes are set to zero. This is useful when a gene is only expressed in the subset of labels and is consistently DEG in each comparison of the subset. Testing against the average computed from only those labels in the subset would fail to detect this DEG as subset-specific.

Author(s)

Aaron Lun

See Also

[pseudoBulkDGE](#), for the underlying function that does all the heavy lifting.

Examples

```
set.seed(10000)
library(scuttle)
sce <- mockSCE(ncells=1000)
sce$samples <- gl(8, 125) # Pretending we have 8 samples.

# Making up some clusters.
sce <- logNormCounts(sce)
clusters <- kmeans(t(logcounts(sce)), centers=3)$cluster

# Creating a set of pseudo-bulk profiles:
info <- DataFrame(sample=sce$samples, cluster=clusters)
pseudo <- sumCountsAcrossCells(sce, info)

# Making up an experimental design for our 8 samples
# and adding a common DEG across all labels.
pseudo$DRUG <- gl(2,4)[pseudo$sample]
assay(pseudo)[1,pseudo$DRUG==1] <- assay(pseudo)[1,pseudo$DRUG==1] * 10

# Label-specific analysis (note behavior of the first gene).
out <- pseudoBulkSpecific(pseudo,
  label=pseudo$cluster,
  condition=pseudo$DRUG,
  design=~DRUG,
  coef="DRUG2"
)

out[[1]]
```

quickCluster *Quick clustering of cells*

Description

Cluster similar cells based on their expression profiles, using either log-expression values or ranks.

Usage

```
quickCluster(x, ...)

## S4 method for signature 'ANY'
quickCluster(
  x,
  min.size = 100,
  method = c("igraph", "hclust"),
  use.ranks = FALSE,
  d = NULL,
  subset.row = NULL,
  min.mean = NULL,
  graph.fun = "walktrap",
  BSPARAM = bsparam(),
  BPPARAM = SerialParam(),
  block = NULL,
  block.BPPARAM = SerialParam(),
  ...
)

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

Arguments

x	A numeric count matrix where rows are genes and columns are cells. Alternatively, a SummarizedExperiment object containing such a matrix.
...	For the generic, further arguments to pass to specific methods. For the ANY method, additional arguments to be passed to NNGraphParam for method="igraph" or to HclustParam for method="hclust". For the SummarizedExperiment method, additional arguments to pass to the ANY method.
min.size	An integer scalar specifying the minimum size of each cluster.
method	String specifying the clustering method to use. "hclust" uses hierarchical clustering while "igraph" uses graph-based clustering.
use.ranks	A logical scalar indicating whether clustering should be performed on the rank matrix, i.e., based on Spearman's rank correlation.
d	An integer scalar specifying the number of principal components to retain. If d=NULL and use.ranks=TRUE, this defaults to 50. If d=NULL and use.rank=FALSE, the number of PCs is chosen by denoisePCA . If d=NA, no dimensionality reduction is performed and the gene expression values (or their rank equivalents) are directly used in clustering.

subset.row	See ?" scran-gene-selection ".
min.mean	A numeric scalar specifying the filter to be applied on the average count for each filter prior to computing ranks. Only used when use.ranks=TRUE, see ? scaledColRanks for details.
graph.fun	A function specifying the community detection algorithm to use on the nearest neighbor graph when method="igraph". Usually obtained from the igraph package.
BSPARAM	A BiocSingularParam object specifying the algorithm to use for PCA, if d is not NA.
BPPARAM	A BiocParallelParam object to use for parallel processing within each block.
block	A factor of length equal to ncol(x) specifying whether clustering should be performed within pre-specified blocks. By default, all columns in x are treated as a single block.
block.BPPARAM	A BiocParallelParam object specifying whether and how parallelization should be performed across blocks, if block is non-NULL and has more than one level.
assay.type	A string specifying which assay values to use.

Details

This function provides a convenient wrapper to quickly define clusters of a minimum size `min.size`. Its intended use is to generate “quick and dirty” clusters for use in [computeSumFactors](#). Two clustering strategies are available:

- If `method="hclust"`, a distance matrix is constructed; hierarchical clustering is performed using Ward’s criterion; and [cutreeDynamic](#) is used to define clusters of cells.
- If `method="igraph"`, a shared nearest neighbor graph is constructed using the [buildSNNGraph](#) function. This is used to define clusters based on highly connected communities in the graph, using the `graph.fun` function.

By default, `quickCluster` will apply these clustering algorithms on the principal component (PC) scores generated from the log-expression values. These are obtained by running [denoisePCA](#) on HVGs detected using the trend fitted to endogenous genes with [modelGeneVar](#). If `d` is specified, the PCA is directly performed on the entire `x` and the specified number of PCs is retained.

It is also possible to use the clusters from this function for actual biological interpretation. In such cases, users should set `min.size=0` to avoid aggregation of small clusters. However, it is often better to call the relevant functions ([modelGeneVar](#), [denoisePCA](#) and [buildSNNGraph](#)) manually as this provides more opportunities for diagnostics when the meaning of the clusters is important.

Value

A character vector of cluster identities for each cell in `x`.

Clustering within blocks

We can break up the dataset by specifying `block` to cluster cells, usually within each batch or run. This generates clusters within each level of `block`, which is entirely adequate for applications like [computeSumFactors](#) where the aim of clustering is to separate dissimilar cells rather than group together all similar cells. Blocking reduces computational work considerably by allowing each level to be processed independently, without compromising performance provided that there are enough cells within each batch.

Indeed, for applications like [computeSumFactors](#), we can use `block` even in the absence of any known batch structure. Specifically, we can set it to an arbitrary factor such as `block=cut(seq_len(ncol(x)),10)` to split the cells into ten batches of roughly equal size. This aims to improve speed, especially when combined with `block.PARAM` to parallelize processing of the independent levels.

Using ranks

If `use.ranks=TRUE`, clustering is instead performed on PC scores obtained from scaled and centred ranks generated by [scaledColRanks](#). This effectively means that clustering uses distances based on the Spearman's rank correlation between two cells. In addition, if `x` is a [dgCMatrix](#) and `BSPARAM` has `deferred=TRUE`, ranks will be computed without loss of sparsity to improve speed and memory efficiency during PCA.

When `use.ranks=TRUE`, the function will filter out genes with average counts (as defined by [calculateAverage](#)) below `min.mean` prior to computing ranks. This removes low-abundance genes with many tied ranks, especially due to zeros, which may reduce the precision of the clustering. We suggest setting `min.mean` to 1 for read count data and 0.1 for UMI data - the function will automatically try to determine this from the data if `min.mean=NULL`.

Setting `use.ranks=TRUE` is invariant to scaling normalization and avoids circularity between normalization and clustering, e.g., in [computeSumFactors](#). However, the default is to use the log-expression values with `use.ranks=FALSE`, as this yields finer and more precise clusters.

Enforcing cluster sizes

With `method="hclust"`, [cutreeDynamic](#) is used to ensure that all clusters contain a minimum number of cells. However, some cells may not be assigned to any cluster and are assigned identities of " \emptyset " in the output vector. In most cases, this is because those cells belong in a separate cluster with fewer than `min.size` cells. The function will not be able to call this as a cluster as the minimum threshold on the number of cells has not been passed. Users are advised to check that the unassigned cells do indeed form their own cluster. Otherwise, it may be necessary to use a different clustering algorithm.

When using `method="igraph"`, clusters are first identified using the specified `graph.fun`. If the smallest cluster contains fewer cells than `min.size`, it is merged with the closest neighbouring cluster. In particular, the function will attempt to merge the smallest cluster with each other cluster. The merge that maximizes the modularity score is selected, and a new merged cluster is formed. This process is repeated until all (merged) clusters are larger than `min.size`.

Author(s)

Aaron Lun and Karsten Bach

References

van Dongen S and Enright AJ (2012). Metric distances derived from cosine similarity and Pearson and Spearman correlations. *arXiv* 1208.3145

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

[computeSumFactors](#), where the clustering results can be used as `clusters=`.

[buildSNNGraph](#), for additional arguments to customize the clustering when `method="igraph"`.

[cutreeDynamic](#), for additional arguments to customize the clustering when `method="hclust"`.

[scaledColRanks](#), to get the rank matrix that was used with `use.rank=TRUE`.
[quickSubCluster](#), for a related function that uses a similar approach for subclustering.

Examples

```
library(scuttle)
sce <- mockSCE()

# Basic application (lowering min.size for this small demo):
clusters <- quickCluster(sce, min.size=50)
table(clusters)

# Operating on ranked expression values:
clusters2 <- quickCluster(sce, min.size=50, use.ranks=TRUE)
table(clusters2)

# Using hierarchical clustering:
clusters <- quickCluster(sce, min.size=50, method="hclust")
table(clusters)
```

quickPseudotime	<i>Quick MST-based pseudotime</i>
-----------------	-----------------------------------

Description

A convenience wrapper to quickly compute a minimum spanning tree (MST) on the cluster centroids to obtain a pseudotime ordering of the cells. This function is now deprecated as it has been moved to the **TSCAN** package itself.

Usage

```
quickPseudotime(
  x,
  clusters,
  use = 1,
  outgroup = FALSE,
  outscale = 3,
  start = NULL
)
```

Arguments

x	A named list of numeric matrices containing dimensionality reduction results. All matrices should have the same number of cells, i.e., rows. Alternatively, a SingleCellExperiment containing such results in its <code>reducedDims</code> .
clusters	A factor of length equal to the number of cells in x, specifying the cluster assignment for each cell.
use	Integer scalar or string specifying the entry of x to use for MST construction and pseudotime calculations.
outgroup, outscale	Arguments passed to createClusterMST .
start	Arguments passed to orderClusterMST .

Details

This function simply calls, in order:

- `rowsum`, to compute the average low-dimensional coordinates for each cluster.
- `createClusterMST` on the average coordinates specified by `use`.
- `orderClusterMST` on the average and per-cell coordinates specified by `use`.
- `connectClusterMST` on the average coordinates for all entries of `x`.

Value

A `List` containing:

- `centered`, a list of numeric matrices containing the averaged coordinates for each cluster. Each matrix corresponds to a dimensionality reduction result in `x`.
- `mst`, a `graph` object containing the cluster-level MST computed on the coordinates from `use`.
- `ordering`, a numeric matrix of pseudotimes for various paths through the MST computed from `use`.
- `connected`, a list of `data.frames` containing the edge coordinates between centers. Each `data.frame` corresponds to a dimensionality reduction result in `x`.

Author(s)

Aaron Lun

See Also

`createClusterMST` and friends, for the functions that do the actual work.

Examples

```
# Mocking up some data:
library(scuttle)
sce <- mockSCE(ncells=500)
sce <- logNormCounts(sce)
sce <- scater::runPCA(sce)
clusters <- clusterSNNGraph(sce, use.dimred="PCA")

# Quickly computing the pseudotime:
out <- quickPseudotime(sce, clusters, use="PCA")
out$mst
head(out$ordering)
```

quickSubCluster	<i>Quick and dirty subclustering</i>
-----------------	--------------------------------------

Description

Performs a quick subclustering for all cells within each group.

Usage

```
quickSubCluster(x, ...)

## S4 method for signature 'ANY'
quickSubCluster(x, normalize = TRUE, ...)

## S4 method for signature 'SummarizedExperiment'
quickSubCluster(x, ...)

## S4 method for signature 'SingleCellExperiment'
quickSubCluster(
  x,
  groups,
  normalize = TRUE,
  prepFUN = NULL,
  min.ncells = 50,
  clusterFUN = NULL,
  BLUSPARAM = NNGraphParam(),
  format = "%s.%s",
  assay.type = "counts",
  simplify = FALSE
)
```

Arguments

x	A matrix of counts or log-normalized expression values (if <code>normalize=FALSE</code>), where each row corresponds to a gene and each column corresponds to a cell. Alternatively, a SummarizedExperiment or SingleCellExperiment object containing such a matrix.
...	For the generic, further arguments to pass to specific methods. For the ANY and SummarizedExperiment methods, further arguments to pass to the SingleCellExperiment method.
normalize	Logical scalar indicating whether each subset of x should be log-transformed prior to further analysis.
groups	A vector of group assignments for all cells, usually corresponding to cluster identities.
prepFUN	A function that accepts a single SingleCellExperiment object and returns another SingleCellExperiment containing any additional elements required for clustering (e.g., PCA results).
min.ncells	An integer scalar specifying the minimum number of cells in a group to be considered for subclustering.

clusterFUN	A function that accepts a single SingleCellExperiment object and returns a vector of cluster assignments for each cell in that object.
BLUSPARAM	A BlusterParam object that is used to specify the clustering via clusterRows . Only used when clusterFUN=NULL.
format	A string to be passed to sprintf , specifying how the subclusters should be named with respect to the parent level in groups and the level returned by clusterFUN.
assay.type	String or integer scalar specifying the relevant assay.
simplify	Logical scalar indicating whether just the subcluster assignments should be returned.

Details

quickSubCluster is a simple convenience function that loops over all levels of groups to perform subclustering. It subsets `x` to retain all cells in one level and then runs `prepFUN` and `clusterFUN` to cluster them. Levels with fewer than `min.ncells` are not subclustered and have "subcluster" set to the name of the level.

The distinction between `prepFUN` and `clusterFUN` is that the former's calculations are preserved in the output. For example, we would put the PCA in `prepFUN` so that the PCs are returned in the [reducedDims](#) for later use. In contrast, `clusterFUN` is only used to obtain the subcluster assignments so any intermediate objects are lost.

By default, `prepFUN` will run [modelGeneVar](#), take the top 10 `clusterFUN` will then perform clustering on the PC matrix with [clusterRows](#) and [BLUSPARAM](#). Either or both of these functions can be replaced with custom functions.

The default behavior of this function is the same as running [quickCluster](#) on each subset with default parameters except for `min.size=0`.

Value

By default, a named [List](#) of [SingleCellExperiment](#) objects. Each object corresponds to a level of groups and contains a "subcluster" column metadata field with the subcluster identities for each cell. The [metadata](#) of the [List](#) also contains `index`, a list of integer vectors specifying the cells in `x` in each returned [SingleCellExperiment](#) object; and `subcluster`, a character vector of subcluster identities (see next).

If `simplify=TRUE`, the character vector of subcluster identities is returned. This is of length equal to `ncol(x)` and each entry follows the format defined in `format`. (Unless the number of cells in the parent cluster is less than `min.cells`, in which case the parent cluster's name is used.)

Author(s)

Aaron Lun

See Also

[quickCluster](#), for a related function to quickly obtain clusters.

Examples

```
library(scuttle)
sce <- mockSCE(ncells=200)
```

```

# Lowering min.size for this small demo:
clusters <- quickCluster(sce, min.size=50)

# Getting subclusters:
out <- quickSubCluster(sce, clusters)

# Defining custom prep functions:
out2 <- quickSubCluster(sce, clusters,
  prepFUN=function(x) {
    dec <- modelGeneVarWithSpikes(x, "Spikes")
    top <- getTopHVGs(dec, prop=0.2)
    scater::runPCA(x, subset_row=top, ncomponents=25)
  }
)

# Defining custom cluster functions:
out3 <- quickSubCluster(sce, clusters,
  clusterFUN=function(x) {
    kmeans(reducedDim(x, "PCA"), sqrt(ncol(x)))$cluster
  }
)

```

sandbag

Cell cycle phase training

Description

Use gene expression data to train a classifier for cell cycle phase.

Usage

```

sandbag(x, ...)

## S4 method for signature 'ANY'
sandbag(x, phases, gene.names = rownames(x), fraction = 0.5, subset.row = NULL)

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

```

Arguments

x	A numeric matrix of gene expression values where rows are genes and columns are cells. Alternatively, a SummarizedExperiment object containing such a matrix.
...	For the generic, additional arguments to pass to specific methods. For the SummarizedExperiment method, additional arguments to pass to the ANY method.
phases	A list of subsetting vectors specifying which cells are in each phase of the cell cycle. This should typically be of length 3, with elements named as "G1", "S" and "G2M".
gene.names	A character vector of gene names.

<code>fraction</code>	A numeric scalar specifying the minimum fraction to define a marker gene pair.
<code>subset.row</code>	See ?" scran-gene-selection ".
<code>assay.type</code>	A string specifying which assay values to use, e.g., "counts" or "logcounts".

Details

This function implements the training step of the pair-based prediction method described by Scialdone et al. (2015). Pairs of genes (A, B) are identified from a training data set where in each pair, the fraction of cells in phase G1 with expression of $A > B$ (based on expression values in `training.data`) and the fraction with $B > A$ in each other phase exceeds `fraction`. These pairs are defined as the marker pairs for G1. This is repeated for each phase to obtain a separate marker pair set.

Pre-defined sets of marker pairs are provided for mouse and human (see Examples). The mouse set was generated as described by Scialdone et al. (2015), while the human training set was generated with data from Leng et al. (2015). Classification from test data can be performed using the [cyclone](#) function. For each cell, this involves comparing expression values between genes in each marker pair. The cell is then assigned to the phase that is consistent with the direction of the difference in expression in the majority of pairs.

Value

A named list of data.frames, where each data frame corresponds to a cell cycle phase and contains the names of the genes in each marker pair.

Author(s)

Antonio Scialdone, with modifications by Aaron Lun

References

Scialdone A, Natarajana KN, Saraiva LR et al. (2015). Computational assignment of cell-cycle stage from single-cell transcriptome data. *Methods* 85:54–61

Leng N, Chu LF, Barry C et al. (2015). Oscope identifies oscillatory genes in unsynchronized single-cell RNA-seq experiments. *Nat. Methods* 12:947–50

See Also

[cyclone](#), to perform the classification on a test dataset.

Examples

```
library(scuttle)
sce <- mockSCE(ncells=50, ngenes=200)

is.G1 <- 1:20
is.S <- 21:30
is.G2M <- 31:50
out <- sandbag(sce, list(G1=is.G1, S=is.S, G2M=is.G2M))
str(out)

# Getting pre-trained marker sets
mm.pairs <- readRDS(system.file("exdata", "mouse_cycle_markers.rds", package="scran"))
hs.pairs <- readRDS(system.file("exdata", "human_cycle_markers.rds", package="scran"))
```

scaledColRanks *Compute scaled column ranks*

Description

Compute scaled column ranks from each cell's expression profile for distance calculations based on rank correlations.

Usage

```
scaledColRanks(
  x,
  subset.row = NULL,
  min.mean = NULL,
  transposed = FALSE,
  as.sparse = FALSE,
  withDimnames = TRUE,
  BPPARAM = SerialParam()
)
```

Arguments

x	A numeric matrix-like object containing cells in columns and features in the rows.
subset.row	A logical, integer or character scalar indicating the rows of x to use, see ?"scran-gene-selection".
min.mean	A numeric scalar specifying the filter to be applied on the average normalized count for each feature prior to computing ranks. Disabled by setting to NULL.
transposed	A logical scalar specifying whether the output should be transposed.
as.sparse	A logical scalar indicating whether the output should be sparse.
withDimnames	A logical scalar specifying whether the output should contain the dimnames of x.
BPPARAM	A BiocParallelParam object specifying whether and how parallelization should be performed. Currently only used for filtering if min.mean is not provided.

Details

Euclidean distances computed based on the output rank matrix are equivalent to distances computed from Spearman's rank correlation. This can be used in clustering, nearest-neighbour searches, etc. as a robust alternative to Euclidean distances computed directly from x.

If as.sparse=TRUE, the most common average rank is set to zero in the output. This can be useful for highly sparse input data where zeroes have the same rank and are themselves returned as zeroes. Obviously, this means that the ranks are not centred, so this will have to be done manually prior to any downstream distance calculations.

Value

A matrix of the same dimensions as x, where each column contains the centred and scaled ranks of the expression values for each cell. If transposed=TRUE, this matrix is transposed so that rows correspond to cells. If as.sparse, the columns are not centered to preserve sparsity.

Author(s)

Aaron Lun

See Also[quickCluster](#), where this function is used.**Examples**

```
library(scuttle)
sce <- mockSCE()
rout <- scaledColRanks(counts(sce), transposed=TRUE)

# For use in clustering:
d <- dist(rout)
table(cutree(hclust(d), 4))

g <- buildSNNGraph(rout, transposed=TRUE)
table(igraph::cluster_walktrap(g)$membership)
```

summaryMarkerStats *Summary marker statistics*

Description

Compute additional gene-level statistics for each group to assist in identifying marker genes, to complement the formal test statistics generated by [findMarkers](#).

Usage

```
summaryMarkerStats(x, ...)
```

```
## S4 method for signature 'ANY'
summaryMarkerStats(
  x,
  groups,
  row.data = NULL,
  average = "mean",
  BPPARAM = SerialParam()
)
```

```
## S4 method for signature 'SummarizedExperiment'
summaryMarkerStats(x, ..., assay.type = "logcounts")
```

Arguments

x A numeric matrix-like object of expression values, where each column corresponds to a cell and each row corresponds to an endogenous gene. This is generally expected to be normalized log-expression values unless one knows better. Alternatively, a [SummarizedExperiment](#) or [SingleCellExperiment](#) object containing such a matrix.

...	For the generic, further arguments to pass to specific methods. For the SummarizedExperiment method, further arguments to pass to the ANY method.
groups	A vector of length equal to <code>ncol(x)</code> , specifying the group to which each cell is assigned. If <code>x</code> is a SingleCellExperiment , this defaults to <code>colLabels(x)</code> if available.
row.data	A DataFrame containing additional row metadata for each gene in <code>x</code> , to be included in each of the output DataFrames. This should generally have row names identical to those of <code>x</code> . Alternatively, a list containing one such DataFrame per level of groups, where each DataFrame contains group-specific metadata for each gene to be included in the appropriate output DataFrame.
average	String specifying the type of average, to be passed to sumCountsAcrossCells .
BPPARAM	A BiocParallelParam object indicating whether and how parallelization should be performed across genes.
assay.type	A string specifying which assay values to use, usually "logcounts".

Details

This function only generates descriptive statistics for each gene to assist marker selection. It does not consider blocking factors or covariates that would otherwise be available from comparisons between groups. For the sake of brevity, statistics for the “other” groups are summarized into a single value.

Value

A named [List](#) of [DataFrames](#), with one entry per level of groups. Each DataFrame has number of rows corresponding to the rows in `x` and contains the fields:

- `self.average`, the average (log-)expression across all cells in the current group.
- `other.average`, the grand average of the average (log-)expression across cells in the other groups.
- `self.detected`, the proportion of cells with detected expression in the current group.
- `other.detected`, the average proportion of cells with detected expression in the other groups.

Author(s)

Aaron Lun

See Also

[findMarkers](#), where the output of this function can be used in `row.data=`.

Examples

```
library(scuttle)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Any clustering method is okay.
kout <- kmeans(t(logcounts(sce)), centers=3)
sum.out <- summaryMarkerStats(sce, kout$cluster)
```

```

sum.out[["1"]]

# Add extra rowData if you like.
rd <- DataFrame(Symbol=sample(LETTERS, nrow(sce), replace=TRUE),
  row.names=rownames(sce))
sum.out <- summaryMarkerStats(sce, kout$cluster, row.data=rd)
sum.out[["1"]]

```

testLinearModel

Hypothesis tests with linear models

Description

Perform basic hypothesis tests with linear models in an efficient manner.

Usage

```

testLinearModel(x, ...)

## S4 method for signature 'ANY'
testLinearModel(
  x,
  design,
  coefs = ncol(design),
  contrasts = NULL,
  block = NULL,
  equiweight = FALSE,
  method = "z",
  subset.row = NULL,
  BPPARAM = SerialParam()
)

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

```

Arguments

x	A numeric matrix-like object containing log-expression values for cells (columns) and genes (rows). Alternatively, a SummarizedExperiment containing such a matrix.
...	For the generic, further arguments to pass to specific methods. For the SummarizedExperiment method, further arguments to pass to the ANY method.
design	A numeric design matrix with number of rows equal to <code>ncol(x)</code> .
coefs	An integer vector specifying the coefficients to drop to form the null model. Only used if <code>contrasts</code> is not specified.
contrasts	A numeric vector or matrix specifying the contrast of interest. This should have length (if vector) or number of rows (if matrix) equal to <code>ncol(x)</code> .

block	A factor specifying the blocking levels for each cell in <i>x</i> . If specified, variance modelling is performed separately within each block and statistics are combined across blocks.
equiweight	A logical scalar indicating whether statistics from each block should be given equal weight. Otherwise, each block is weighted according to its number of cells. Only used if <i>block</i> is specified.
method	String specifying how p-values should be combined when <i>block</i> is specified, see combinePValues .
subset.row	See ?" scrn-gene-selection ", specifying the rows for which to model the variance. Defaults to all genes in <i>x</i> .
BPPARAM	A BiocParallelParam object indicating whether parallelization should be performed across genes.
assay.type	String or integer scalar specifying the assay containing the log-expression values.

Details

This function can be considered a more efficient version of [lmFit](#) that works on a variety of matrix representations (see [fitLinearModel](#)). It also omits the empirical Bayes shrinkage step, which is acceptable given the large number of residual d.f. in typical single-cell studies.

If *contrasts* is specified, the null hypothesis is defined by the contrast matrix or vector in the same manner that is used in the **limma** and **edgeR** packages. Briefly, the contrast vector specifies a linear combination of coefficients that sums to zero under the null. For contrast matrices, the joint null consists of the intersection of the nulls defined by each column vector.

Otherwise, if only *coefs* is specified, the null model is formed by simply dropping all of the specified coefficients from *design*.

If *block* is specified, a linear model is fitted separately to the cells in each level. The results are combined across levels by averaging coefficients and combining p-values with [combinePValues](#). By default, the contribution from each level is weighted by its number of cells; if *equiweight*=TRUE, each level is given equal weight instead.

Value

A [DataFrame](#) containing test results with one row per row of *x*. It contains the estimated values of the contrasted coefficients as well as the p-value and FDR for each gene.

Author(s)

Aaron Lun

See Also

[fitLinearModel](#), which performs the hard work of fitting the linear models.

Examples

```
y <- matrix(rnorm(10000), ncol=100)

# Example with categorical factors:
A <- gl(2, 50)
design <- model.matrix(~A)
testLinearModel(y, design, contrasts=c(0, 1))
```

```
# Example with continuous variables:
u <- runif(100)
design <- model.matrix(~u)
testLinearModel(y, design, contrasts=c(0, 1))

# Example with multiple variables:
B <- gl(4, 25)
design <- model.matrix(~B)
testLinearModel(y, design, contrasts=cbind(c(0,1,0,0), c(0,0,1,-1)))
```

testPseudotime	<i>Test for differences along pseudotime</i>
----------------	--

Description

Implements a simple method of testing for significant differences with respect to pseudotime, based on fitting linear models with a spline basis matrix. This function is now deprecated as it has been moved to the **TSCAN** package itself.

Usage

```
testPseudotime(x, ...)
```

```
## S4 method for signature 'ANY'
testPseudotime(
  x,
  pseudotime,
  df = 5,
  get.lfc = TRUE,
  get.spline.coef = FALSE,
  trend.only = TRUE
)
```

```
## S4 method for signature 'SummarizedExperiment'
testPseudotime(x, ..., assay.type = "logcounts")
```

Arguments

x	A numeric matrix-like object containing log-expression values for cells (columns) and genes (rows). Alternatively, a SummarizedExperiment containing such a matrix.
...	For the generic, further arguments to pass to specific method. For the ANY method, further arguments to pass to fitLinearModel . For the SummarizedExperiment method, further arguments to pass to the ANY method.
pseudotime	A numeric matrix with one row per cell in x and one column per path (i.e., lineage). A vector is treated the same as a 1-column matrix.
df	Integer scalar specifying the degrees of freedom for the splines.

<code>get.lfc</code>	Logical scalar indicating whether to return an overall log-fold change along each path.
<code>get.spline.coef</code>	Logical scalar indicating whether to return the estimates of the spline coefficients.
<code>trend.only</code>	Logical scalar indicating whether only differences in the trend should be considered when testing for differences between paths.
<code>assay.type</code>	String or integer scalar specifying the assay containing the log-expression matrix.

Details

For a single path in `pseudotime`, this function fits a natural spline to the expression of each gene with respect to pseudotime. It then does an ANOVA to test whether any of the spline coefficients are non-zero. In this manner, genes exhibiting a significant (and potentially non-linear) trend with respect to the pseudotime can be detected as those with low p-values.

For multiple paths in `pseudotime`, the null hypothesis is that all paths have the same trend (if `trend.only=TRUE`) or the same trend and intercept (if `FALSE`). This is done by effectively fitting a separate trend to each path and performing an ANOVA to detect differences in the trend alone or in the trend and intercept. In this manner, genes exhibiting differences in behavior between paths can be detected.

The expected format of `pseudotime` is the same as that returned by `orderClusterMST`. Each cell is assigned to a path if it has a non-NA value in the corresponding column. For single path testing, cells with NA values in `pseudotime` are ignored; for multiple path testing, cells assigned to multiple paths are ignored.

By default, estimates of the spline coefficients are not returned as they are difficult to interpret. Rather, a log-fold change of expression along each path is estimated to provide some indication of the overall magnitude and direction of any change.

Value

A `DataFrame` is returned containing the statistics for each gene (row), including the p-value and its BH-adjusted equivalent. If `get.lfc=TRUE`, an overall log-fold change is returned for each path.

If `get.spline.coef=TRUE`, the estimated spline coefficients are also returned (single path) or the differences in the spline fits to the first path are returned (multiple paths).

Author(s)

Aaron Lun

See Also

[orderClusterMST](#), to generate the pseudotime matrix.

[testLinearModel](#), which performs the tests under the hood.

Examples

```
y <- matrix(rnorm(10000), ncol=100)

# Testing for a difference in a single path:
u <- runif(100)
```

```
testPseudotime(y, u)

# Testing for differences in multiple paths
# by mocking up a pseudotime matrix.
p <- cbind(path1=u, path2=u)
path1 <- rbinom(length(u), 1, 0.5)==0
p[!path1,1] <- NA
p[path1,2] <- NA

testPseudotime(y, p)
```


Index

- * **variance**
 - Distance-to-median, [50](#)
- [.logBH](#), [3](#)

- [aggregateAcrossCells](#), [39](#)
- [altExp](#), [25](#), [79](#), [80](#), [90](#), [91](#)
- [altExps](#), [24](#), [25](#)
- [approxSilhouette](#), [10](#)

- [binom.test](#), [97](#)
- [binomTest](#), [97](#)
- [BiocNeighborParam](#), [52](#), [57](#)
- [BiocParallelParam](#), [5](#), [17](#), [33](#), [35](#), [42](#), [47](#), [52](#), [57](#), [60](#), [64](#), [76](#), [79](#), [83](#), [86](#), [90](#), [96](#), [99](#), [103](#), [113](#), [121](#), [123](#), [125](#)
- [BiocSingularParam](#), [5](#), [47](#), [52](#), [113](#)
- [BlusterParam](#), [118](#)
- [bootstrapCluster](#), [4](#), [14](#)
- [bootstrapStability](#), [4](#)
- [buildKNNGraph](#), [11](#)
- [buildKNNGraph \(buildSNNGraph\)](#), [4](#)
- [buildKNNGraph, ANY-method \(buildSNNGraph\)](#), [4](#)
- [buildKNNGraph, SingleCellExperiment-method \(buildSNNGraph\)](#), [4](#)
- [buildSNNGraph](#), [4](#), [11](#), [69](#), [70](#), [113](#), [114](#)
- [buildSNNGraph, ANY-method \(buildSNNGraph\)](#), [4](#)
- [buildSNNGraph, SingleCellExperiment-method \(buildSNNGraph\)](#), [4](#)
- [buildSNNGraph, SummarizedExperiment-method \(buildSNNGraph\)](#), [4](#)

- [calculateAverage](#), [29](#), [68](#), [114](#)
- [calculateSumFactors](#)
 - ([computeSumFactors](#)), [25](#)
- [calculateSumFactors, ANY-method \(computeSumFactors\)](#), [25](#)
- [calculateSumFactors, SummarizedExperiment-method \(computeSumFactors\)](#), [25](#)
- [cleanSizeFactors](#), [6](#), [29](#)
- [cluster_walktrap](#), [6](#), [69](#)
- [clusterKNNGraph \(clusterSNNGraph\)](#), [10](#)
- [clusterKNNGraph, ANY-method \(clusterSNNGraph\)](#), [10](#)
- [clusterKNNGraph, SingleCellExperiment-method \(clusterSNNGraph\)](#), [10](#)
- [clusterKNNGraph, SummarizedExperiment-method \(clusterSNNGraph\)](#), [10](#)
- [clusterModularity](#), [8](#)
- [clusterPurity](#), [9](#)
- [clusterPurity, ANY-method \(clusterPurity\)](#), [9](#)
- [clusterPurity, SingleCellExperiment-method \(clusterPurity\)](#), [9](#)
- [clusterPurity, SummarizedExperiment-method \(clusterPurity\)](#), [9](#)
- [clusterRows](#), [10](#), [12](#), [118](#)
- [clusterSNNGraph](#), [10](#)
- [clusterSNNGraph, ANY-method \(clusterSNNGraph\)](#), [10](#)
- [clusterSNNGraph, SingleCellExperiment-method \(clusterSNNGraph\)](#), [10](#)
- [clusterSNNGraph, SummarizedExperiment-method \(clusterSNNGraph\)](#), [10](#)
- [coassignProb](#), [13](#)
- [colLabels](#), [54](#), [60](#), [95](#), [99](#), [103](#), [123](#)
- [combineBlocks](#), [14](#)
- [combineCV2 \(combineVar\)](#), [22](#)
- [combineMarkers](#), [16](#), [60](#), [61](#), [71](#), [74](#), [93](#), [94](#), [96](#), [97](#), [99](#), [102](#), [103](#), [105](#)
- [combinePValues](#), [15](#), [18](#), [20](#), [23](#), [24](#), [76](#), [77](#), [79](#), [81](#), [83](#), [84](#), [86](#), [87](#), [90](#), [92](#), [125](#)
- [combineVar](#), [15](#), [22](#)
- [communities](#), [11](#)
- [computeSpikeFactors](#), [24](#)
- [computeSumFactors](#), [7](#), [8](#), [25](#), [113](#), [114](#)
- [connectClusterMST](#), [116](#)
- [connectClusterMST \(createClusterMST\)](#), [38](#)
- [convertTo](#), [30](#)
- [cor](#), [37](#), [38](#)
- [correlateGenes](#), [32](#), [38](#)
- [correlateNull](#), [33](#), [35](#), [36](#)
- [correlatePairs](#), [32–34](#), [34](#)
- [correlatePairs, ANY-method \(correlatePairs\)](#), [34](#)

- correlatePairs, SummarizedExperiment-method (correlatePairs), 34
- createClusterMST, 38, 40, 115, 116
- cutree, 70
- cutreeDynamic, 113, 114
- cyclone, 41, 120
- cyclone, ANY-method (cyclone), 41
- cyclone, SummarizedExperiment-method (cyclone), 41

- DataFrame, 9, 12, 14–17, 23, 32, 36, 47, 55, 58, 60, 61, 70–73, 76, 80, 83, 87, 91, 93, 96, 100, 104, 106–108, 110, 123, 125, 127
- decideTests, 44, 45
- decideTestsPerLabel, 44, 108, 110
- decomposeVar (defunct), 45
- defunct, 45
- denoisePCA, 46, 112, 113
- denoisePCANumber (denoisePCA), 46
- DESeqDataSetFromMatrix, 31
- dgCMatrix, 114
- DGEList, 31
- Distance-to-median, 50
- DM (Distance-to-median), 50
- doubletCells, 51, 56, 58
- doubletCells, ANY-method (doubletCells), 51
- doubletCells, SingleCellExperiment-method (doubletCells), 51
- doubletCells, SummarizedExperiment-method (doubletCells), 51
- doubletCluster, 53, 54, 58
- doubletCluster, ANY-method (doubletCluster), 54
- doubletCluster, SingleCellExperiment-method (doubletCluster), 54
- doubletCluster, SummarizedExperiment-method (doubletCluster), 54
- doubletRecovery, 56
- doubletRecovery, ANY-method (doubletRecovery), 56
- doubletRecovery, SingleCellExperiment-method (doubletRecovery), 56
- doubletRecovery, SummarizedExperiment-method (doubletRecovery), 56

- filterByExpr, 108
- findKNN, 6
- findMarkers, 20, 46, 54, 56, 59, 70, 71, 93–95, 99, 103, 122, 123
- findMarkers, ANY-method (findMarkers), 59
- findMarkers, SingleCellExperiment-method (findMarkers), 59
- findMarkers, SummarizedExperiment-method (findMarkers), 59
- findNeighbors, 52
- fitLinearModel, 125, 126
- fitTrendCV2, 46, 62, 75–77, 79–81
- fitTrendPoisson, 64, 86
- fitTrendVar, 46, 47, 64, 65, 65, 83, 84, 86–92

- Gene selection, 68
- getClusteredPCs, 69
- getDenoisedPCs (denoisePCA), 46
- getDenoisedPCs, ANY-method (denoisePCA), 46
- getDenoisedPCs, SummarizedExperiment-method (denoisePCA), 46
- getMarkerEffects, 61, 70
- getTopHVGs, 71
- getTopMarkers, 73, 97, 102, 105
- glmTreat, 107
- graph, 6, 11, 12, 39, 116

- HclustParam, 112

- improvedCV2 (defunct), 45

- kmeans, 12

- librarySizeFactors, 25, 27, 52, 55, 76, 80, 87, 91
- List, 17, 44, 70, 74, 93, 108, 110, 116, 118, 123
- lmFit, 125
- logNormCounts, 30, 54, 100

- makeContrasts, 107
- makeKNNGraph, 4–6
- makeSNNGraph, 4–6
- makeTechTrend (defunct), 45
- metadata, 12, 58, 108, 118
- modelGeneCV2, 23, 24, 46, 63, 72, 75, 79, 81
- modelGeneCV2, ANY-method (modelGeneCV2), 75
- modelGeneCV2, SingleCellExperiment-method (modelGeneCV2), 75
- modelGeneCV2, SummarizedExperiment-method (modelGeneCV2), 75
- modelGeneCV2WithSpikes, 23, 63, 76, 78
- modelGeneCV2WithSpikes, ANY-method (modelGeneCV2WithSpikes), 78
- modelGeneCV2WithSpikes, SingleCellExperiment-method (modelGeneCV2WithSpikes), 78

- modelGeneCV2WithSpikes, SummarizedExperiment-method (modelGeneCV2WithSpikes), 78
- modelGeneVar, 15, 23, 24, 46, 49, 66, 67, 71, 72, 82, 88, 90, 92, 113, 118
- modelGeneVar, ANY-method (modelGeneVar), 82
- modelGeneVar, SingleCellExperiment-method (modelGeneVar), 82
- modelGeneVar, SummarizedExperiment-method (modelGeneVar), 82
- modelGeneVarByPoisson, 46, 48, 49, 85
- modelGeneVarByPoisson, ANY-method (modelGeneVarByPoisson), 85
- modelGeneVarByPoisson, SingleCellExperiment-method (modelGeneVarByPoisson), 85
- modelGeneVarByPoisson, SummarizedExperiment-method (modelGeneVarByPoisson), 85
- modelGeneVarWithSpikes, 23, 25, 47–49, 66, 67, 83, 84, 86, 89
- modelGeneVarWithSpikes, ANY-method (modelGeneVarWithSpikes), 89
- modelGeneVarWithSpikes, SingleCellExperiment-method (modelGeneVarWithSpikes), 89
- modelGeneVarWithSpikes, SummarizedExperiment-method (modelGeneVarWithSpikes), 89
- multiBlockNorm (defunct), 45
- multiBlockVar (defunct), 45
- multiMarkerStats, 93

- neighborPurity, 9
- neighborsToKNNGraph (buildSNNGraph), 4
- neighborsToSNNGraph (buildSNNGraph), 4
- newCellDataSet, 31
- nls, 7, 8, 62, 63, 66, 67
- nls.control, 7
- NNGraphParam, 10, 12, 112

- orderClusterMST, 115, 116, 127
- orderClusterMST (createClusterMST), 38
- overlapExprs, 46
- overlapExprs (defunct), 45

- pairwiseBinom, 17, 60, 61, 94
- pairwiseBinom, ANY-method (pairwiseBinom), 94
- pairwiseBinom, SingleCellExperiment-method (pairwiseBinom), 94
- pairwiseBinom, SummarizedExperiment-method (pairwiseBinom), 94
- pairwiseModularity, 8
- pairwiseRand, 13, 14
- pairwiseTTests, 17, 20, 60, 61, 74, 98, 103
- pairwiseTTests, ANY-method (pairwiseTTests), 98
- pairwiseTTests, SingleCellExperiment-method (pairwiseTTests), 98
- pairwiseTTests, SummarizedExperiment-method (pairwiseTTests), 98
- pairwiseWilcox, 17, 20, 60, 61, 102
- pairwiseWilcox, ANY-method (pairwiseWilcox), 102
- pairwiseWilcox, SingleCellExperiment-method (pairwiseWilcox), 102
- pairwiseWilcox, SummarizedExperiment-method (pairwiseWilcox), 102

- parallelPCA, 46
- parallelPCA (defunct), 45
- pseudoBulkDGE, 44, 45, 106, 110, 111
- pseudoBulkDGE, ANY-method (pseudoBulkDGE), 106
- pseudoBulkDGE, SummarizedExperiment-method (pseudoBulkDGE), 106
- pseudoBulkSpecific, 108, 109
- pseudoBulkSpecific, ANY-method (pseudoBulkSpecific), 109
- pseudoBulkSpecific, SummarizedExperiment-method (pseudoBulkSpecific), 109

- quickCluster, 28, 30, 112, 118, 122
- quickCluster, ANY-method (quickCluster), 112
- quickCluster, SummarizedExperiment-method (quickCluster), 112
- quickPseudotime, 40, 115
- quickSubCluster, 115, 117
- quickSubCluster, ANY-method (quickSubCluster), 117
- quickSubCluster, SingleCellExperiment-method (quickSubCluster), 117
- quickSubCluster, SummarizedExperiment-method (quickSubCluster), 117

- reducedDims, 5, 9, 11, 48, 57, 115, 118
- Rle, 68
- rowsum, 116
- runmed, 50
- runPCA, 70
- runSVD, 49

- sandbag, 42, 43, 119
- sandbag, ANY-method (sandbag), 119
- sandbag, SummarizedExperiment-method (sandbag), 119
- scaledColRanks, 113–115, 121

scran-gene-selection (Gene selection),
68

segments, 39

SimpleList, 55

SingleCellExperiment, 4, 5, 9, 11, 24, 26,
30, 31, 47, 51, 54, 57, 60, 68, 75, 76,
79, 80, 86, 87, 89–91, 99, 103, 115,
117, 118, 122, 123

sizeFactors, 25, 27, 52, 76, 80, 87, 91

sprintf, 118

sumCountsAcrossCells, 107, 108, 123

SummarizedExperiment, 5, 9, 11, 26, 35, 42,
47, 51, 54, 57, 60, 75, 79, 82, 83, 86,
89, 90, 99, 103, 112, 117, 119, 122,
124, 126

summarizeTestsPerLabel
(decideTestsPerLabel), 44

summaryMarkerStats, 60, 61, 122

summaryMarkerStats, ANY-method
(summaryMarkerStats), 122

summaryMarkerStats, SummarizedExperiment-method
(summaryMarkerStats), 122

t.test, 101

technicalCV2 (defunct), 45

testLinearModel, 15, 124, 127

testLinearModel, ANY-method
(testLinearModel), 124

testLinearModel, SummarizedExperiment-method
(testLinearModel), 124

testPseudotime, 126

testPseudotime, ANY-method
(testPseudotime), 126

testPseudotime, SummarizedExperiment-method
(testPseudotime), 126

testVar (defunct), 45

topTable, 108

topTags, 108

treat, 100, 107, 110

trendVar (defunct), 45

TwoStepParam, 10, 12

voom, 107

voomWithQualityWeights, 107

weightedLowess, 66, 67

wilcox.test, 104, 105