

Package ‘scran’

April 15, 2020

Version 1.14.6

Date 2020-01-29

Title Methods for Single-Cell RNA-Seq Data Analysis

Description Implements functions for low-level analyses of single-cell RNA-seq data. Methods are provided for normalization of cell-specific biases, 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, scater, edgeR, limma, BiocNeighbors, igraph, statmod, DelayedArray, DelayedMatrixStats, BiocSingular, dqrng

Suggests testthat, BiocStyle, knitr, beachmat, HDF5Array, scRNAseq, dynamicTreeCut, DESeq2, monocle, Biobase, aroma.light, pheatmap

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

LinkingTo Rcpp, beachmat, BH, dqrng

License GPL-3

NeedsCompilation yes

VignetteBuilder knitr

SystemRequirements C++11

RoxygenNote 6.1.1

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

git_branch RELEASE_3_10

git_last_commit 240fad2

git_last_commit_date 2020-01-30

Date/Publication 2020-04-14

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:

bootstrapCluster	3
buildSNNGraph	4
cleanSizeFactors	8
clusterModularity	10
combineMarkers	11
combinePValues	15
combineVar	17
computeSpikeFactors	19
computeSumFactors	20
convertTo	25
correlateGenes	26
correlateNull	27
correlatePairs	29
cyclone	32
decomposeVar	35
denoisePCA	38
Distance-to-median	41
doubletCells	42
doubletCluster	45
findMarkers	48
fitTrendCV2	50
fitTrendPoisson	52
fitTrendVar	53
Gene selection	55
getClusteredPCs	56
getTopHVGs	58
getTopMarkers	59
improvedCV2	61
makeTechTrend	63
modelGeneCV2	65
modelGeneCV2WithSpikes	68
modelGeneVar	71
modelGeneVarByPoisson	74
modelGeneVarWithSpikes	77
multiBlockNorm	81
multiBlockVar	83
overlapExprs	85
pairwiseBinom	86
pairwiseTTests	89
pairwiseWilcox	92
Parallel analysis	95
quickCluster	97
quickSubCluster	101
sandbag	103
scaledColRanks	104
technicalCV2	106
testVar	108
trendVar	110

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.

Usage

```
bootstrapCluster(x, FUN, clusters = NULL, transposed = FALSE,  
  iterations = 20, ...)
```

Arguments

x	A two-dimensional object containing cells in columns. This is usually a numeric matrix of log-expression values, but can also be a SingleCellExperiment object. If transposed=TRUE, cells are expected to be in the rows, e.g., for precomputed PCs.
FUN	A function that accepts x and returns a vector of cluster identities.
clusters	A vector of cluster identities obtained by calling FUN(x, ...). This is provided as an additional argument in the case that the clusters have already been computed, in which case we can save a single round of computation.
transposed	Logical scalar indicating whether x is transposed with cells in the rows.
iterations	Integer scalar specifying the number of bootstrap iterations.
...	Further arguments to pass to FUN to control the clustering procedure.

Details

Bootstrapping is conventionally used to evaluate the precision of an estimator by applying it to an *in silico*-generated replicate dataset. We can (ab)use this framework to determine the stability of the clusters in the context of a scRNA-seq analysis. We sample cells with replacement from x, perform clustering with FUN and compare the new clusters to clusters.

The relevant statistic is the co-assignment probability for each pair of original clusters, i.e., the probability that a randomly chosen cells from each of the two original clusters will be put in the same bootstrap cluster. High co-assignment probabilities indicate that the two original clusters were not stably separated. We might then only trust separation between two clusters if their co-assignment probability was less than some threshold, e.g., 5%.

The co-assignment probability of each cluster to itself provides some measure of per-cluster stability. A probability of 1 indicates that all cells are always assigned to the same cluster across bootstrap iterations, while internal structure that encourages the formation of subclusters will lower this probability.

Value

A numeric matrix with upper triangular entries filled with the co-assignment probabilities for each pair of clusters in clusters.

Statistical notes

We use the co-assignment probability as it is more interpretable than, say, the Jaccard index (see the **fpc** package). It also focuses on the relevant differences between clusters, allowing us to determine which aspects of a clustering are stable. For example, A and B may be well separated but A and C may not be, which is difficult to represent in a single stability measure for A. If our main interest lies in the A/B separation, we do not want to be overly pessimistic about the stability of A, even though it might not be well-separated from all other clusters.

Technically speaking, some mental gymnastics are required to compare the original and bootstrap clusters in this manner. After bootstrapping, the sampled cells represent distinct entities from the original dataset (otherwise it would be difficult to treat them as independent replicates) for which the original clusters do not immediately apply. Instead, we assume that we perform label transfer using a nearest-neighbors approach - which, in this case, is the same as using the original label for each cell, as the nearest neighbor of each resampled cell to the original dataset is itself.

Needless to say, bootstrapping will only generate replicates that differ by sampling noise. Real replicates will differ due to composition differences, variability in expression across individuals, etc. Thus, any stability inferences from bootstrapping are likely to be overly optimistic.

Author(s)

Aaron Lun

See Also

[quickCluster](#), to get a quick and dirty function to use as FUN. It is often more computationally efficient to define your own function, though.

Examples

```
library(scater)
sce <- mockSCE(ncells=200)
bootstrapCluster(sce, FUN=quickCluster, min.size=10)

# Defining your own function:
sce <- logNormCounts(sce)
sce <- runPCA(sce)
bootstrapCluster(reducedDim(sce), transposed=TRUE, FUN=function(x) {
  kmeans(x, 2)$cluster
})
```

buildSNNGraph

Build a nearest-neighbor graph

Description

Build a shared or k-nearest-neighbors graph for cells based on their expression profiles.

Usage

```

buildSNNGraph(x, ...)

## S4 method for signature 'ANY'
buildSNNGraph(x, k = 10, d = 50, type = c("rank",
  "number", "jaccard"), transposed = FALSE, subset.row = NULL,
  BNPARAM = KmknnParam(), BSPARAM = bsparam(),
  BPPARAM = SerialParam())

## S4 method for signature 'SingleCellExperiment'
buildSNNGraph(x, ..., subset.row = NULL,
  assay.type = "logcounts", get.spikes = FALSE, use.dimred = NULL)

buildKNNGraph(x, ...)

## S4 method for signature 'ANY'
buildKNNGraph(x, k = 10, d = 50, directed = FALSE,
  transposed = FALSE, subset.row = NULL, BNPARAM = KmknnParam(),
  BSPARAM = bsparam(), BPPARAM = SerialParam())

## S4 method for signature 'SingleCellExperiment'
buildKNNGraph(x, ..., subset.row = NULL,
  assay.type = "logcounts", get.spikes = FALSE, use.dimred = NULL)

neighborsToSNNGraph(indices, type = c("rank", "number", "jaccard"))

neighborsToKNNGraph(indices, directed = FALSE)

```

Arguments

x	For the ANY method, 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> . For the SingleCellExperiment method, a <code>SingleCellExperiment</code> containing such an expression matrix. Alternatively, graph building will be performed from its reducedDims if <code>use.dimred</code> is set.
...	For the generics, additional arguments to pass to the specific methods. For the <code>SingleCellExperiment</code> methods, additional arguments to pass to the corresponding ANY method.
k	An integer scalar specifying the number of nearest neighbors to consider during graph construction.
d	An integer scalar specifying the number of dimensions to use for the search.
type	A string specifying the type of weighting scheme to use for shared neighbors.
transposed	A logical scalar indicating whether x is transposed (i.e., rows are cells).
subset.row	See <code>?"scran-gene-selection"</code> .
BNPARAM	A BiocNeighborParam object specifying the nearest neighbor algorithm.
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.
get.spikes	See ?"scran-gene-selection".
use.dimred	A string specifying whether existing values in reducedDims(x) should be used.
directed	A logical scalar indicating whether the output of buildKNNGraph should be a directed graph.
indices	An integer matrix where each row corresponds to a cell and contains the indices of the k nearest neighbors (by increasing distance) from that cell.

Details

The buildSNNGraph method builds a shared nearest-neighbour graph using cells as nodes. For each cell, its k nearest neighbours are identified using the `findKNN` function, based on distances between their expression profiles (Euclidean by default). An edge is drawn between all pairs of cells that share at least one neighbour, weighted by the characteristics of the shared nearest neighbors:

- If type="rank", the weighting scheme defined by Xu and Su (2015) is used. The weight between two nodes is $k - r/2$ where r is the smallest sum of ranks for any shared neighboring node. For example, if one node was the closest neighbor of each of two nodes, the weight between the two latter nodes would be $k - 1$. For the purposes of this ranking, each node has a rank of zero in its own nearest-neighbor set. More shared neighbors, or shared neighbors that are close to both cells, will generally yield larger weights.
- If type="number", the weight between two nodes is simply the number of shared nearest neighbors between them. The weight can range from zero to $k + 1$, as the node itself is included in its own nearest-neighbor set. This is a simpler scheme that is also slightly faster but does not account for the ranking of neighbors within each set.
- If type="jaccard", the weight between two nodes is the Jaccard similarity between the two sets of neighbors for those nodes. This weight can range from zero to 1, and is a monotonic transformation of the weight used by type="number". It is provided for consistency with other clustering algorithms such as those in **seurat**.

The aim is to use the SNN graph to perform clustering of cells via community detection algorithms in the **igraph** package. This is faster and more memory efficient than hierarchical clustering for large numbers of cells. In particular, it avoids the need to construct a distance matrix for all pairs of cells. Only the identities of nearest neighbours are required, which can be obtained quickly with methods in the **BiocNeighbors** package.

The choice of k controls the connectivity of the graph and the resolution of community detection algorithms. Smaller values of k will generally yield smaller, finer clusters, while increasing k will increase the connectivity of the graph and make it more difficult to resolve different communities. The value of k can be roughly interpreted as the anticipated size of the smallest subpopulation. If a subpopulation in the data has fewer than k+1 cells, buildSNNGraph and buildKNNGraph will forcibly construct edges between cells in that subpopulation and cells in other subpopulations. This increases the risk that the subpopulation will not form its own cluster as it is more interconnected with the rest of the cells in the dataset.

Note that the setting of k here is slightly different from that used in SNN-Cliq. The original implementation considers each cell to be its first nearest neighbor that contributes to k. In buildSNNGraph, the k nearest neighbours refers to the number of *other* cells.

The buildKNNGraph method builds a simpler k-nearest neighbour graph. Cells are again nodes, and edges are drawn between each cell and its k-nearest neighbours. No weighting of the edges is performed. In theory, these graphs are directed as nearest neighbour relationships may not be reciprocal. However, by default, directed=FALSE such that an undirected graph is returned.

Value

A [graph](#) where nodes are cells and edges represent connections between nearest neighbors. For `buildSNNGraph`, these edges are weighted by the number of shared nearest neighbors. For `buildKNNGraph`, edges are not weighted but may be directed if `directed=TRUE`.

Choice of input data

In practice, PCA is performed on a matrix-like `x` to obtain the first `d` principal components. The `k`-NN search can then be rapidly performed on the PCs rather than on the full expression matrix. By default, the first 50 components are chosen, which should retain most of the substructure in the data set. If `d` is `NA` or greater than or equal to the number of cells, no dimensionality reduction is performed.

The PCA is performed using methods the `runSVD` function from the **BiocSingular** package. To improve speed, this can be done using approximate algorithms by modifying `BSPARAM`, e.g., to `Ir1baParam()`. Approximate algorithms will converge towards the correct result but often involve some random initialization and thus are technically dependent on the session seed. For full reproducibility, users are advised to call `set.seed` beforehand when using this option.

Expression values in `x` should typically be on the log-scale, e.g., log-transformed counts. Ranks can also be used for greater robustness, e.g., from `scaledColRanks`. (Dimensionality reduction is still okay when ranks are provided - running PCA on ranks is equivalent to running MDS on the distance matrix derived from Spearman's rho.)

If the input matrix `x` is already transposed for the ANY method, `transposed=TRUE` avoids an unnecessary internal transposition. A typical use case is when `x` contains some reduced dimension coordinates with cells in the rows. In such cases, setting `transposed=TRUE` and `d=NA` will use the input coordinates directly for graph-building.

The same principles apply when `x` is a [SingleCellExperiment](#) object, except that the relevant matrix is now retrieved from the assays using `assay.type`. If `use.dimred` is not `NULL`, existing PCs are used from the specified entry of `reducedDims(x)`, and any setting of `d`, `subset.row` and `get.spikes` are ignored.

The `neighborsToSNNGraph` and `neighborsToKNNGraph` functions operate directly on a matrix of nearest neighbor indices, obtained using functions like `findKNN`. This may be useful for constructing a graph from precomputed nearest-neighbor search results. Note that the user is responsible for ensuring that the indices are valid (i.e., `range(indices)` is positive and no greater than `max(indices)`).

Author(s)

Aaron Lun, with KNN code contributed by Jonathan Griffiths.

References

Xu C and Su Z (2015). Identification of cell types from single-cell transcriptomes using a novel clustering method. *Bioinformatics* 31:1974-80

See Also

See `make_graph` for details on the graph output object.

See `cluster_walktrap`, `cluster_louvain` and related functions in **igraph** for clustering based on the produced graph.

Also see `findKNN` for specifics of the nearest-neighbor search.

Examples

```

library(scater)
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 <- runPCA(sce)
g <- buildSNNGraph(sce, use.dimred="PCA")
clusters <- igraph::cluster_fast_greedy(g)$membership
table(clusters)

```

cleanSizeFactors	<i>Clean size factors</i>
------------------	---------------------------

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

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

Details

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

$$y = \frac{ax}{1 + bx}$$

where `y` is `num.detected` and `x` is `size.factors` for all positive size factors. This is a purely empirical expression, chosen because it 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(scater)
sf <- librarySizeFactors(counts)
ngenes <- nexprs(counts, byrow=FALSE)

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

clusterModularity *Compute the cluster-wise modularity*

Description

Calculate the modularity of each cluster from a graph, based on a null model of random connections between nodes.

Usage

```
clusterModularity(graph, clusters, get.weights = FALSE,  
  get.values = NULL, as.ratio = FALSE)
```

Arguments

graph	A graph object from igraph , usually where each node represents a cell.
clusters	Factor specifying the cluster identity for each node.
get.weights	Logical scalar indicating whether the observed and expected edge weights should be returned, rather than the modularity.
get.values	Deprecated, same as <code>get.weights</code> .
as.ratio	Logical scalar indicating whether the log-ratio of observed to expected weights should be returned.

Details

This function computes a modularity score in the same manner as that from [modularity](#). The modularity is defined as the (scaled) difference between the observed and expected number of edges between nodes in the same cluster. The expected number of edges is defined by a null model where edges are randomly distributed among nodes. The same logic applies for weighted graphs, replacing the number of edges with the summed weight of edges.

Whereas [modularity](#) returns a modularity score for the entire graph, `clusterModularity` provides scores for the individual clusters. The sum of the diagonal elements of the output matrix should be equal to the output of [modularity](#) (after supplying weights to the latter, if necessary). A well-separated cluster should have mostly intra-cluster edges and a high modularity score on the corresponding diagonal entry, while two closely related clusters that are weakly separated will have many inter-cluster edges and a high off-diagonal score.

In practice, the modularity may not be the most effective metric for evaluating cluster separatedness. This is because the modularity is proportional to the number of cells, so larger clusters will naturally have a large score regardless of separation. An alternative approach is to set `as.ratio=TRUE`, which returns the ratio of the observed to expected weights for each entry of the matrix. This adjusts for differences in cluster size and improves resolution of differences between clusters.

Directed graphs are treated as undirected inputs with `mode="each"` in [as.undirected](#). In the rare case that self-loops are present, these will also be handled correctly.

Value

By default, an upper triangular numeric matrix of order equal to the number of clusters is returned. Each entry corresponds to a pair of clusters and is proportional to the difference between the observed and expected edge weights between those clusters.

If `as.ratio=TRUE`, an upper triangular numeric matrix is again returned. Here, each entry is equal to the ratio between the observed and expected edge weights.

If `get.weights=TRUE`, a list is returned containing two upper triangular numeric matrices. The observed matrix contains the observed sum of edge weights between and within clusters, while the expected matrix contains the expected sum of edge weights under the random model.

Author(s)

Aaron Lun

See Also

[buildSNNGraph](#), for one method to construct graph.

[modularity](#), for the calculation of the entire graph modularity.

Examples

```
library(scater)
sce <- mockSCE()
sce <- logNormCounts(sce)
g <- buildSNNGraph(sce)
clusters <- igraph::cluster_walktrap(g)$membership

# Examining the modularity values directly.
out <- clusterModularity(g, clusters)
out

# Compute the ratio instead, for visualization
# (log-transform to improve range of colors).
out <- clusterModularity(g, clusters, as.ratio=TRUE)
image(log2(out+1))

# Alternatively, get the edge weights directly:
out <- clusterModularity(g, clusters, get.weights=TRUE)
out
```

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)
```

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 NULL, 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 <code>DataFrame</code> should be sorted by a statistic relevant to <code>pval.type</code> .

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](#).

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 p-value across all comparisons if `log.p.out=FALSE`. This is a Simes' p-value if `pval.type="any"` or "some", otherwise it is an IUT p-value.

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`.

<OUTPUT>.Y: Numeric, where the value of <OUTPUT> is set to `output.field`. One of these fields is present for every other cluster Y in `clusters`. It contains the effect size of the comparison of X to Y , with typical values depending on the chosen method:

- `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.

Only reported when `full.stats=FALSE` and `effect.field` is not NULL.

`stats.y`: A nested DataFrame containing the same fields in the corresponding entry of `de.lists` for the X versus Y comparison. Only reported when `full.stats=TRUE`.

Ordering rules are:

- Within each DataFrame, if `sorted=TRUE`, genes are ranked by the `Top` column (if available) and then the `p.value` column. 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 .

Consolidating with DE against any other cluster

By default, each DataFrame is sorted by the `Top` value when `pval.type="any"`. (For genes with the same `Top`, ranking is performed based on the Simes combined p-value - see the comments for `pval.type="some"`.) 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 `Top` ≤ 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.

Note that this only relates to the default setting of `min.prop=0` with `pval.type="any"`. For user-specified values of `min.prop`, see the comments in the “some clusters” section below.

Consolidating with DE against all other clusters

If `pval.type="all"`, the null hypothesis is that the gene is not DE in all contrasts, and the IUT p-value is computed for each gene. 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.

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 is 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"`.

A slightly different flavor of this approach is achieved by setting `method="any"` with `min.prop=0.5`. Genes will then only be high-ranked if they are significant in at least some `min.prop` proportion of the 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 increases the stringency of the "any" setting but with similar loss of theoretical guarantees.

Correcting for multiple testing

The BH method is then applied on the Simes/IUT 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.

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(scater)
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

...	Two or more DataFrames of variance modelling results. For <code>combineVar</code> , these should be produced by <code>modelGeneVar</code> or <code>modelGeneVarWithSpikes</code> . For <code>combineCV2</code> , these should be produced by <code>modelGeneCV2</code> or <code>modelGeneCV2WithSpikes</code> .
method	String specifying how p-values are to be combined, see combinePValues for options.
pval.field	A string specifying the column name of each element of ... that contains the p-value.
other.fields	A character vector specifying the fields containing other statistics to combine.
equiweight	Logical scalar indicating whether each result is to be given equal weight in the combined statistics.
ncells	Numeric vector containing the number of cells used to generate each element of 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(scater)
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, type = NULL, assay.type = "counts",
  sf.out = FALSE, general.use = TRUE)

```

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.
type	Deprecated, a character vector specifying which spike-in sets to use.
assay.type	A string indicating which assay contains the counts.
sf.out	Deprecated, a logical scalar indicating whether only size factors should be returned.
general.use	A logical scalar indicating whether the size factors should be stored for general use by all genes.

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](#)).

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(scater)
sce <- mockSCE()
sce <- computeSpikeFactors(sce, "Spikes")
summary(sizeFactors(sce))

# Deprecated behavior:
y <- sce
altExps(y) <- NULL
y <- rbind(y, altExp(sce))
suppressWarnings(isSpike(y, "ERCC") <- nrow(sce) + seq_len(nrow(altExp(sce))))
suppressWarnings(y <- computeSpikeFactors(y))

y2 <- y
```

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, ..., subset.row = NULL, assay.type = "counts",
  get.spikes = FALSE, sf.out = FALSE)
```

Arguments

<code>x</code>	For <code>calculateSumFactors</code> , a numeric matrix-like object of counts, where rows are genes and columns are cells. Alternatively, a SummarizedExperiment object containing such a matrix. For <code>computeSumFactors</code> , a SingleCellExperiment object containing a count matrix.
<code>...</code>	For the <code>calculateSumFactors</code> generic, additional arguments to pass to each method. For the SummarizedExperiment method, additional methods to pass to the ANY method. For the <code>computeSumFactors</code> function, additional arguments to pass to <code>calculateSumFactors</code> .
<code>sizes</code>	A numeric vector of pool sizes, i.e., number of cells per pool.
<code>clusters</code>	An optional factor specifying which cells belong to which cluster, for deconvolution within clusters.
<code>ref.clust</code>	A level of <code>clusters</code> to be used as the reference cluster for inter-cluster normalization.
<code>max.cluster.size</code>	An integer scalar specifying the maximum number of cells in each cluster.
<code>positive</code>	A logical scalar indicating whether linear inverse models should be used to enforce positive estimates.
<code>scaling</code>	A numeric scalar containing scaling factors to adjust the counts prior to computing size factors.
<code>min.mean</code>	A numeric scalar specifying the minimum (library size-adjusted) average count of genes to be used for normalization.
<code>subset.row</code>	See ?" scran-gene-selection ".
<code>BPPARAM</code>	A <code>BiocParallelParam</code> object specifying whether and how clusters should be processed in parallel.
<code>assay.type</code>	A string specifying which assay values to use when <code>x</code> is a <code>SummarizedExperiment</code> or <code>SingleCellExperiment</code> .
<code>get.spikes</code>	See ?" scran-gene-selection ".
<code>sf.out</code>	Deprecated, a logical scalar indicating whether only size factors should be returned.

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 to improve accuracy

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. The prescaling factor should be close to the true size factor for each cell. (Obviously, the true size factor is unknown so we use the library size for each cell instead.) 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.

The default usage assumes that the library size is roughly proportional to the true size factor. This can be violated in pathological scenarios involving extreme differential expression. In such cases, we recommend running `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 use our initial estimates for prescaling by supplying them to the `scaling` argument. This weakens the assumption above by avoiding large inaccuracies in the prescaling factor.

Obviously, this involves twice as much computational work. As a result, performing multiple iterations is not the default recommendation, especially as the benefits are only apparent 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.

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(scater)
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,	get.spikes 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(scater)
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(scater)
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, cache.size = 100L,
  BPPARAM = SerialParam())

## S4 method for signature 'SingleCellExperiment'
correlatePairs(x, ..., use.names = TRUE,
  subset.row = NULL, assay.type = "logcounts", get.spikes = FALSE)
```

Arguments

x	A numeric matrix-like object of log-normalized expression values, where rows are genes and columns are cells. Alternatively, a SingleCellExperiment object containing such a matrix.
...	Additional arguments to pass to <code>correlatePairs,ANY-method</code> .
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 <code>correlateNull</code> 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 ?"scrn-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.
cache.size	Deprecated argument, ignored.

BPPARAM	A BiocParallelParam object that specifies the manner of parallel processing to use.
assay.type	A string specifying which assay values to use.
get.spikes	See ?" scran-gene-selection ".

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. Rows corresponding to spike-in transcripts are also removed by default with `get.spikes=FALSE`. This avoids picking up strong technical correlations between pairs of spike-in transcripts.

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 `correlateNull`. 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://lunlab.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(scater)
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)
```

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 'SingleCellExperiment'
cyclone(x, pairs, subset.row = NULL,
        ..., assay.type = "counts", get.spikes = FALSE)

```

Arguments

x	A numeric matrix-like object of gene expression values where rows are genes and columns are cells. Alternatively, a SingleCellExperiment object containing such a matrix.
...	For the generic, additional arguments to pass to specific methods. For the SingleCellExperiment method, additional arguments to pass to the ANY 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 in <code>bplapply</code> for parallel processing.
verbose	A logical scalar specifying whether diagnostics should be printed to screen.
subset.row	See ?" scran-gene-selection ".
assay.type	A string specifying which assay values to use, e.g., "counts" or "logcounts".
get.spikes	See ?" scran-gene-selection ".

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(scater)
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)

```

decomposeVar

*Decompose the gene-level variance***Description**

Decompose the gene-specific variance into biological and technical components for single-cell RNA-seq data.

Usage

```

## S4 method for signature 'ANY,list'
decomposeVar(x, fit, block=NA, design=NA, subset.row=NULL,
             BPPARAM=SerialParam(), ...)

## S4 method for signature 'SingleCellExperiment,list'
decomposeVar(x, fit, subset.row=NULL, ...,
             assay.type="logcounts", get.spikes=NA)

```

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 SingleCellExperiment object containing such a matrix.
fit	A list containing trend, a function that takes a numeric vector of abundances and returns the technical component of variation. This is usually produced by running trendVar on log-expression values for spike-in genes.
block	A factor containing the level of a blocking factor for each cell.
design	A numeric matrix describing the uninteresting factors contributing to expression in each cell. Alternatively, a single factor for one-way layouts.
subset.row	See ?" scran-gene-selection ".
BPPARAM	A BiocParallelParam object indicating whether and how parallelization should be performed across genes.

...	For <code>decomposeVar</code> , <code>matrix</code> , <code>list-method</code> , additional arguments to pass to <code>testVar</code> . For <code>decomposeVar</code> , <code>SingleCellExperiment</code> , <code>list-method</code> , additional arguments to pass to the <code>matrix</code> method.
<code>assay.type</code>	A string specifying which assay values to use from <code>x</code> .
<code>get.spikes</code>	A logical scalar specifying whether decomposition should be performed for spike-ins.

Details

This function computes the variance of the normalized log-counts for each endogenous gene. The technical component of the variance for each gene is determined by interpolating the fitted trend in `fit` at the mean log-count for that gene. This represents variance due to sequencing noise, variability in capture efficiency, etc. The biological component is determined by subtracting the technical component from the total variance.

Highly variable genes (HVGs) can be identified as those with large biological components. Unlike other methods for decomposition, this approach estimates the variance of the log-counts rather than of the counts themselves. The log-transformation blunts the impact of large positive outliers and ensures that HVGs are driven by strong log-fold changes between cells, not differences in counts. Interpretation is not compromised – HVGs will still be so, regardless of whether counts or log-counts are considered.

The `fit` list should contain at least `trend`, as this is necessary for the decomposition. If `x` is missing, `fit` should also contain `mean` and `var`, numeric vectors of the means and variances for all features. This will be used to perform the decomposition rather than (re)computing any statistics from `x`. The list may optionally contain `block` and `design`, but this will be overridden by any explicitly passed arguments.

If `assay.type="logcounts"` and the size factors are not centred at unity, a warning will be raised - see `?trendVar` for details.

Value

A `DataFrame` is returned where each row corresponds to and is named after a row of `x`. 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 `bio=0`.

If `get.spikes=NA`, the `p.value` and `FDR` fields will be set to `NA` for rows corresponding to spike-in transcripts. Otherwise, if `get.spikes=FALSE`, rows corresponding to spike-in transcripts are removed.

If `subset.row!=NULL`, each row of the output `DataFrame` corresponds to an element of `subset.row` instead.

The metadata field of the output `DataFrame` also contains `num.cells`, an integer scalar storing the number of cells in `x`; and `resid.df`, an integer scalar specifying the residual d.f. used for variance estimation.

Accounting for uninteresting factors

To account for uninteresting factors of variation, either `block` or `design` can be specified:

- Setting `block` will estimate the mean and variance of each gene for cells in each group (i.e., each level of `block`) separately. The technical component is also estimated for each group separately, based on the group-specific mean. Group-specific statistics are combined to obtain a single value per gene. For means and variances, this is done by taking a weighted average across groups, with weighting based on the residual d.f. (for variances) or number of cells (for means). For p-values, Stouffer's method is used on the group-specific p-values returned by `testVar`, with the residual d.f. used as weights.
- 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 variance is calculated from the residual variance of the fit. The technical component is estimated as the fitted value of the trend at the mean expression across all cells for that gene. This approach is useful for covariates or additive models that cannot be expressed as a one-way layout for use in `block`. Of course, one-way layouts can still be specified as a full design matrix or by passing a factor directly as `design`.

If either of these arguments are `NA`, they will be extracted from `fit`, assuming that the same cells were used to fit the trend. If `block` is specified, this will override any setting of `design`. Use of `block` is generally favoured as group-specific means result in a better estimate of the technical component than an average mean across all groups.

Note that the use of either `block` or `design` assumes that there are no systematic differences in the size factors across levels of an uninteresting factor. If such differences are present, we suggest using `multiBlockVar` instead, see the discussion in `?trendVar` for more details.

Feature selection

The behaviour of `get.spikes` and `subset.row` is the same as described in `?"scran-gene-selection"`. The only additional feature is that users can specify `get.spikes=NA`, which sets the p-value and FDR to `NA` for spike-in transcripts. This is the default as it returns the other variance statistics for diagnostic purposes, but ensures that the spike-ins are not treated as candidate HVGs. (Note that this setting is the same as `get.spikes=TRUE` when considering the interaction between `get.spikes` and `subset.row`.)

If `x` is not supplied, all genes used to fit the trend in `fit` will be used instead for the variance decomposition. This may be useful when a trend is fitted to all genes in `trendVar`, such that the statistics for all genes will already be available in `fit`. By not specifying `x`, users can avoid redundant calculations, which is particularly helpful for very large data sets.

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

Lun ATL (2018). Description of the HVG machinery in *scran*. <https://github.com/LTLA/HVGDetection2018>

See Also

`trendVar`, `testVar`

Examples

```

example(computeSpikeFactors) # Using the mocked-up data 'y' from this example.
y <- computeSumFactors(y) # Size factors for the the endogenous genes.
y <- computeSpikeFactors(y, general.use=FALSE) # Size factors for spike-ins.
y <- normalize(y) # Normalizing the counts by the size factors.

# Decomposing technical and biological noise.
fit <- trendVar(y)
results <- decomposeVar(y, fit)
head(results)

plot(results$mean, results$total)
o <- order(results$mean)
lines(results$mean[o], results$tech[o], col="red", lwd=2)

plot(results$mean, results$bio)

# A trend fitted to endogenous genes can also be used, pending assumptions.
fit.g <- trendVar(y, use.spikes=FALSE)
results.g <- decomposeVar(y, fit.g)
head(results.g)

```

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

denoisePCA(x, ..., subset.row = NULL, value = c("pca", "lowrank"),
  assay.type = "logcounts", get.spikes = FALSE, sce.out = TRUE)

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

```

Arguments

x A numeric matrix of log-expression values.
Alternatively, a [SingleCellExperiment](#) object containing such values.

...	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 <code>ANY</code> 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 <code>DataFrame</code> of variance decomposition results generated by <code>modelGeneVar</code> or related functions.
<code>subset.row</code>	See <code>?"scrn-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 <code>BiocSingularParam</code> object specifying the algorithm to use for PCA.
<code>BPPARAM</code>	A <code>BiocParallelParam</code> 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>get.spikes</code>	See <code>?"scrn-gene-selection"</code> .
<code>sce.out</code>	Deprecated, a logical scalar specifying whether a modified <code>SingleCellExperiment</code> object should be returned.
<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. 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 X PCs. For a given value of X , we compute the variance explained by all of the later PCs. We aim to find the largest value of X such that the sum of variances explained by the later PCs is still less than the variance attributable to technical noise. This X 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 X 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`.

(It naturally follows that, if our HVG selection strategy involves picking all genes with positive biological components, there is no need to set `subset.row` explicitly in this function, because that selection would be redundant with the filtering performed internally by `getDenoisedPCs`.)

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

In reality, the choice of X will only be 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. Thus, from a mathematical perspective, X will usually be underestimated if our aim was to retain all signal.

On the other hand, many aspects of biological variation are not that interesting in most applications (e.g., transcriptional bursting, metabolic fluctuations). It is often the case that we do not actually

need to retain all signal, in which case X is likely a gross overestimate in the context of the wider analysis. This can be mitigated by using `modelGeneVar` rather than `modelGeneVarWithSpikes`, as the former attempts to remove “uninteresting” biological variation; and by using a more stringent HVG subset in `subset.row`, to focus on the stronger aspects of biological variation.

Author(s)

Aaron Lun

References

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

See Also

`fitTrendVar` and `modelGeneVar`, for methods of computing technical components.
`runSVD`, for the underlying SVD algorithm(s).

Examples

```
library(scater)
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

Detect doublet cells

Description

Identify potential doublet cells based on simulations of putative doublet expression profiles.

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 'SingleCellExperiment'
doubletCells(x,
  size.factors.norm = sizeFactors(x), ..., subset.row = NULL,
  assay.type = "counts", get.spikes = FALSE)
```

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

BPPARAM	A BiocParallelParam object specifying whether the neighbour searches should be parallelized.
assay.type	A string specifying which assay values contain the count matrix.
get.spikes	See ?" scran-gene-selection ".

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 intermediate expression profiles.

Usage

```
doubletCluster(x, ...)

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

## S4 method for signature 'SingleCellExperiment'
doubletCluster(x, ...,
  subset.row = NULL, assay.type = "counts", get.spikes = FALSE)
```

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 [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 SingleCellExperiment method, additional arguments to pass to the ANY method.
clusters	A vector of cluster identities for all cells.
subset.row	See ?" scran-gene-selection ".
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".
get.spikes	See ?" scran-gene-selection ".

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 identify 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 *scrn*. https://l1tla.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(scater)
isOutlier(dbl$N, log=TRUE, type="lower")

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

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)

## S4 method for signature 'SingleCellExperiment'
findMarkers(x, ..., subset.row = NULL,
  assay.type = "logcounts", get.spikes = FALSE)
```

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 [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 `SingleCellExperiment` method, further arguments to pass to the ANY method.

groups	A vector of group assignments for all cells.
test.type	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".
pval.type	A string specifying how p-values are to be combined across pairwise comparisons for a given group/cluster.
min.prop	Numeric scalar specifying the minimum proportion of significant comparisons per gene, Defaults to 0.5 when pval.type="some", otherwise defaults to zero.
log.p	A logical scalar indicating if log-transformed p-values/FDRs should be returned.
full.stats	A logical scalar indicating whether all statistics in de.lists should be stored in the output for each pairwise comparison.
sorted	Logical scalar indicating whether each output DataFrame should be sorted by a statistic relevant to pval.type.
row.data	A DataFrame containing additional row metadata for each gene in x, to be included in each of the output DataFrames. If sorted=TRUE, this should have the same row names as the output of combineMarkers (usually rownames(x), see the gene.names argument in functions like pairwiseTTests).
subset.row	See ?"scran-gene-selection".
assay.type	A string specifying which assay values to use, usually "logcounts".
get.spikes	See ?"scran-gene-selection".

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

See ?[combineMarkers](#) for more details on the output format.

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.

Examples

```
library(scater)
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 <code>nls</code> .
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.iter`s 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(scater)
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,
            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>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`.

If `parametric=TRUE`, 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`. `weightedLowess` is then applied to the log-ratios of the variance to the fitted value for each gene. 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`, smoothing is performed directly on the log-variances using `weightedLowess`.

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.

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.

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 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. Note that these densities are computed after filtering on `min.mean`.

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

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

Examples

```
library(scater)
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 matrix `x`, we can choose to perform calculations only on a subset of rows with `subset.row`. 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 subsetted matrix. The same reasoning applies for any `x` that is a [SingleCellExperiment](#) object.

Handling spike-in transcripts

Many functions only make sense when performed on the endogenous genes. For such functions, spike-in transcripts are automatically removed when the input is a `SingleCellExperiment` and `get.spikes=FALSE`. This is achieved based on the spike-in information returned by `isSpike`.

If `get.spikes=TRUE`, no filtering on the spike-in transcripts will be performed. Filtering will not be performed (and in fact, the argument should be unavailable) if the input is not a `SingleCellExperiment` object.

If `get.spikes=FALSE` and `subset.row` is not `NULL`, the two selections are intersected. That is, only the non-spike-in entries of `subset.row` will be used in the function.

A number of functions may require special treatment of spike-ins, e.g., `trendVar`. Refer to the corresponding documentation for more details.

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 `calcAverage`, 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 `get.spikes=FALSE` and/or a specified `subset.row`. For example, only non-spike-in genes that pass the `min.mean` filter are retained if `get.spikes=TRUE`. Similarly, only selected genes that pass the filter are retained if `subset.row` is specified.

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

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

Details

We 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 driving 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

```
sce <- scater::mockSCE()
sce <- scater::logNormCounts(sce)
sce <- scater::runPCA(sce)

output <- getClusteredPCs(reducedDim(sce))
output

metadata(output)$chosen
```

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(scater)
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

<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>n</code>	Integer scalar specifying the number of markers to obtain from each pairwise comparison, if <code>pairwise=FALSE</code> . Otherwise, the number of top genes to take from each cluster's combined marker set, see Details .
<code>pval.field</code>	String specifying the column of each <code>DataFrame</code> in <code>de.lists</code> 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(scater)
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]]
```

improvedCV2

Stably model the technical coefficient of variation

Description

Model the technical coefficient of variation as a function of the mean, and determine the significance of highly variable genes.

This has been deprecated in favor of the [modelGeneCV2](#) function.

Usage

```
## S4 method for signature 'ANY'
improvedCV2(x, is.spike, sf.cell=NULL, sf.spike=NULL,
            log.prior=NULL, df=4, robust=FALSE, use.spikes=FALSE)

## S4 method for signature 'SingleCellExperiment'
improvedCV2(x, spike.type=NULL, ..., assay.type="logcounts",
            logged=NULL, normalized=NULL)
```

Arguments

x	A numeric matrix of counts, normalized counts or normalized log-expression values, where each column corresponds to a cell and each row corresponds to a spike-in transcript. Alternatively, a <code>SingleCellExperiment</code> object that contains such values.
is.spike	A vector indicating which rows of x correspond to spike-in transcripts.
sf.cell	A numeric vector containing size factors for endogenous genes. If this is not specified, counts for endogenous genes are assumed to already be normalized. This is ignored if <code>log.prior!=NULL</code> .
sf.spike	A numeric vector containing size factors for spike-in transcripts. If this is not specified, counts for the spike-in transcripts are assumed to already be normalized. This is ignored if <code>log.prior!=NULL</code> .
log.prior	A numeric scalar specifying the pseudo-count added prior to log-transformation. If this is set, x is assumed to contain normalized log-expression values, otherwise it is assumed to contain counts.
df	An integer scalar indicating the number of degrees of freedom for the spline fit with <code>smooth.spline</code> .
robust	Logical scalar specifying whether robust fitting should be performed with <code>robustSmoothSpline</code> from the aroma.light package.
use.spikes	A logical scalar indicating whether p-values should be returned for spike-in transcripts.
spike.type	A character vector containing the names of the spike-in sets to use.
...	Additional arguments to pass to <code>improvedCV2, ANY-method</code> .
assay.type	A string specifying which assay values to use.

logged	A logical scalar indicating if <code>assay.type</code> contains log-expression values. This is automatically determined if <code>assay.type="counts"</code> , <code>"logcounts"</code> or <code>"normcounts"</code> .
normalized	A logical scalar indicating if <code>assay.type</code> is normalized, also automatically determined where possible.

Details

This function will estimate the squared coefficient of variation (CV2) and mean for each spike-in transcript. Both values are log-transformed and a mean-dependent trend is fitted to the log-CV2 values, using a linear model with a natural spline of degree `df`. The trend is used to obtain the technical contribution to the CV2 for each gene. The biological contribution is computed by subtracting the technical contribution from the total CV2. We use a more robust approach to trend fitting that makes this function an improvement over the `technicalCV2` function on which it is based.

Deviations from the trend are identified by modelling the CV2 estimates for the spike-in transcripts as log-normally distributed around the fitted trend. This accounts for sampling variance as well as any variability in the true dispersions (e.g., due to transcript-specific amplification biases). The p-value for each gene is calculated from a one-sided Z-test on the log-CV2, using the fitted value as the mean and the robust scale estimate as the standard deviation. A Benjamini-Hochberg adjustment is applied to correct for multiple testing.

If `log.prior` is specified, `x` is assumed to contain log-expression values. These are converted back to the count scale prior to calculation of the CV2. Otherwise, `x` is assumed to contain raw counts, which need to be normalized with `sf.cell` and `sf.spike` prior to calculating the CV2. Note that both sets of size factors are set to 1 by default if their values are not supplied to the function.

For any given data set, the maximum CV2 that can be achieved is equal to the number of cells. (This occurs when only one cell has a non-zero expression value - proof via Holder's inequality.) Genes with CV2 values equal to the maximum are ignored during trend fitting. This ensures that the trend is not distorted by the presence of an upper bound on CV2 values, especially at low means.

For `improvedCV2,ANY-method`, the rows corresponding to spike-in transcripts are specified with `is.spike`. These rows will be used for trend fitting, while all other rows are treated as endogenous genes. By default, p-values are set to NA for the spike-in transcripts, such that they do not contribute to the multiple testing correction. This behaviour can be modified with `use.spikes=TRUE`, which will return p-values for all features.

For `improvedCV2,SingleCellExperiment-method`, transcripts from spike-in sets named in `spike.type` will be used for trend fitting. If `spike.type=NULL`, all spike-in sets listed in `x` will be used. Size factors for endogenous genes are automatically extracted via `sizeFactors`. Spike-in-specific size factors for `spike.type` are extracted from `x`, if available; otherwise they are set to the size factors for the endogenous genes. Note that the spike-in-specific factors must be the same for each set in `spike.type`.

Users can also set `is.spike` to NA in `improvedCV2,ANY-method`; or `spike.type` to NA in `improvedCV2,SingleCellExperiment-method`. In such cases, all rows will be used for trend fitting, and (adjusted) p-values will be reported for all rows. This should be used in cases where there are no spike-ins. Here, the assumption is that most endogenous genes do not exhibit high biological variability and thus can be used to model decompose variation.

Value

A `DataFrame` is returned containing one row per row of `x` (including both endogenous genes and spike-in transcripts). Each row contains the following information:

mean: A numeric field, containing mean (scaled) counts for all genes and transcripts.

- cv2:** A numeric field, containing CV2 values for all genes and transcripts.
- trend:** A numeric field, containing the fitted value of the trend in the CV2 values. Note that the fitted value is reported for all genes and transcripts, but the trend is only fitted using the transcripts.
- ratio:** A numeric field, containing the ratio of cv2 to trend.
- p.value:** A numeric field, containing p-values for all endogenous genes (NA for rows corresponding to spike-in transcripts).
- FDR:** A numeric field, containing adjusted p-values for all genes.

Author(s)

Aaron Lun

See Also

[technicalCV2](#), for the original method that this function improves on.

Examples

```
# Mocking up some data.
ngenes <- 10000
nsamples <- 50
means <- 2^runif(ngenes, 6, 10)
dispersions <- 10/means + 0.2
counts <- matrix(rnbinom(ngenes*nsamples, mu=means, size=1/dispersions), ncol=nsamples)
is.spike <- logical(ngenes)
is.spike[seq_len(500)] <- TRUE

# Running it directly on the counts.
out <- improvedCV2(counts, is.spike)
head(out)

plot(out$mean, out$cv2, log="xy")
points(out$mean, out$trend, col="red", pch=16, cex=0.5)
```

makeTechTrend

Make a technical trend

Description

Manufacture a mean-variance trend for log-transformed expression values, assuming Poisson or NB-distributed technical noise for count data.

Usage

```
makeTechTrend(means, size.factors=1, tol=1e-6, dispersion=0,
  pseudo.count=1, approx.npts=Inf, x=NULL, BPPARAM=SerialParam())
```

Arguments

means	A numeric vector of average counts. Note that there are means of the counts, <i>not</i> means of the log-expression values.
size.factors	A numeric vector of size factors.
tol	A numeric scalar specifying the tolerance for approximating the mean/variance. Lower values result in greater accuracy.
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.
approx.npts	An integer scalar specifying the number of interpolation points to use.
x	A SingleCellExperiment object from which size.factors and pseudo.count are extracted, and means can be automatically inferred.
BPPARAM	A BiocParallelParam object indicating whether and how parallelization should be performed across means.

Details

At each value of means, this function will examine the distribution of Poisson/NB-distributed counts with the corresponding mean. All counts are log₂-transformed after addition of pseudo.count, and the mean and variance is computed for the log-transformed values. Setting dispersion to a non-zero value will use a NB distribution instead of the default Poisson.

If size.factors is a vector, one count distribution is generated for each of its elements, where the mean is scaled by the corresponding size factor. Counts are then divided by the size factor prior to log-transformation, mimicking the effect of normalization in [normalize](#). A composite distribution of log-values is constructed by pooling all of these individual distributions. The mean and variance is then computed for a composite distribution.

Finally, a function is fitted to all of the computed variances, using the means of the log-values as the covariate. Note that the returned function accepts mean log-values as input, *not* the mean counts that were supplied in means. This means that the function is directly usable as a replacement for the trend returned by [trendVar](#).

If x is set, pseudo.count is overridden by metadata(sce)\$log.exprs.offset; size.factors is overridden by sizeFactors(sce) (or the column sums of counts(sce), if no size factors are present in x); and means is automatically determined from the range of row averages of logcounts(sce) (after undoing the log-transformation).

If approx.npts is finite and less than length(size.factors), an approximate approach is used to construct the trend. We define approx.npts evenly spaced points (on the log-scale) between the smallest and largest size factors. Each point is treated as a proxy size factor and used to construct a count distribution as previously described. The expected log-count (and expected sum of squares) at each point is computed, and interpolation is used to obtain the corresponding values at the actual size factors. This avoids constructing separate distributions for each element of size.factors.

Value

A function accepting a mean log-expression as input and returning the variance of the log-expression as the output.

Author(s)

Aaron Lun

See Also

[trendVar](#), [normalize](#)

Examples

```
means <- 1:100/10
out <- makeTechTrend(means)
curve(out(x), xlim=c(0, 5))
```

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 'SingleCellExperiment'
modelGeneCV2(x, size.factors = NULL,
  ..., assay.type = "counts")
```

Arguments

x	A numeric matrix of counts, or a 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 SingleCellExperiment method, further arguments to pass to the ANY 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 x. If specified, variance modelling is performed separately within each block and statistics are combined across blocks.
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.
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 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 `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 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(scater)
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 'SingleCellExperiment'
modelGeneCV2WithSpikes(x, spikes,
  size.factors = NULL, spike.size.factors = NULL, ...,
  assay.type = "counts")

```

Arguments

x	A numeric matrix of counts for endogenous genes, or a 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 SingleCellExperiment method, further arguments to pass to the ANY method.
spikes	A numeric matrix of counts for spike-in transcripts. 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 x, to be used for scaling gene expression.
spike.size.factors	A numeric vector of size factors for each cell in spikes, to be used for scaling spike-ins.
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.
subset.row	See ?"scrans-gene-selection" , specifying the rows for which to model the variance. Defaults to all genes in x.
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 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(scater)
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 'SingleCellExperiment'
modelGeneVar(x, ...,
  assay.type = "logcounts")

```

Arguments

<code>x</code>	A numeric matrix of log-counts, or a SingleCellExperiment containing such a matrix.
<code>block</code>	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.
<code>design</code>	A numeric matrix containing blocking terms for uninteresting factors of variation.
<code>subset.row</code>	See <code>?"scran-gene-selection"</code> , specifying the rows for which to model the variance. Defaults to all genes in <code>x</code> .
<code>subset.fit</code>	An argument similar to <code>subset.row</code> , specifying the rows to be used for trend fitting. Defaults to <code>subset.row</code> .
<code>...</code>	For the generic, further arguments to pass to each method. For the ANY method, further arguments to pass to fitTrendVar . For the SingleCellExperiment 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 combinePValues .
<code>BPPARAM</code>	A BiocParallelParam 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(scater)
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 'SingleCellExperiment'
modelGeneVarByPoisson(x,
  size.factors = sizeFactors(x), ..., assay.type = "counts")

```

Arguments

x	A numeric matrix of counts for endogenous genes, or a 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 SingleCellExperiment method, further arguments to pass to the ANY 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 <code>?"scran-gene-selection"</code> , 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(scater)
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")
}
```

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 'SingleCellExperiment'
modelGeneVarWithSpikes(x, spikes,
  size.factors = NULL, spike.size.factors = NULL, ...,
  assay.type = "counts")
```

Arguments

x	A numeric matrix of counts for endogenous genes, or a 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 SingleCellExperiment method, further arguments to pass to the ANY method.
spikes	A numeric matrix of counts for spike-in transcripts. 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 x, to be used for scaling gene expression.
spike.size.factors	A numeric vector of size factors for each cell in spikes, to be used for scaling spike-ins.
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.
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 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(scater)
sce <- mockSCE()
sce <- logNormCounts(sce)

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

plot(spk$mean, spk$total, log="xy")
points(metadata(spk)$mean, metadata(spk)$cv2, 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 <- 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")
}
```

Description

Perform scaling normalization within each block in a manner that preserves the relative scale of counts between spike-in transcripts and endogenous genes.

This has now been deprecated as rescaling is performed automatically in [modelGeneVarWithSpikes](#).

Usage

```
multiBlockNorm(x, block, ...)
```

Arguments

x	A SingleCellExperiment object containing counts and size factors.
block	A factor specifying the blocking level for each cell in x.
...	Further arguments to pass to normalize .

Details

When comparing spike-in and endogenous variances, the assumption is that a spike-in transcript is comparable to an endogenous gene with similar magnitudes of the counts. This is motivated by the mean-variance relationship in Poisson and other count-based models. Thus, we want to ensure that the (relative) average abundances after normalization reflect the similarity in the average count between spike-in transcripts and endogenous genes. This is usually achieved by centering all sets of size factors so that the normalization does not systematically alter the mean between spike-ins and endogenous genes. Indeed, this is the default mode of operation in [normalize](#).

However, centering across all cells is not appropriate when `block` contains multiple levels and we want to fit trends within each level (see [multiBlockVar](#)). In such cases, we want size factors to be centered *within* each level, which is not guaranteed by global centering. To overcome this, we adjust the spike-in size factors so that the mean within each level of the blocking factor is the same as that of the endogenous size factors for that level. This avoids cases where spike-in abundances are systematically shifted up or down relative to the abundances of the endogenous genes (e.g., due to addition of different spike-in quantities across blocks).

In all cases, the outcome of normalization for endogenous genes is guaranteed to be the same as that from [normalize](#). Only the size factors and normalized values for spike-in transcripts will be different when using this function. This ensures that comparisons of gene-level expression profiles between cells (e.g., during clustering or dimensionality reduction) are not altered.

Value

A SingleCellExperiment with normalized log-expression values in the "logcounts" slot (depending on the arguments to [normalize](#)).

Author(s)

Aaron Lun

See Also

[normalize](#), [multiBlockVar](#)

Examples

```

example(computeSpikeFactors) # Using the mocked-up data 'y' from this example.

# Normalizing (gene-based factors for genes, spike-in factors for spike-ins)
y <- computeSumFactors(y)
y <- computeSpikeFactors(y, general.use=FALSE)

# Setting up the blocking levels.
block <- sample(3, ncol(y), replace=TRUE)
y <- multiBlockNorm(y, block)
assayNames(y)

```

multiBlockVar	<i>Per-block variance statistics</i>
---------------	--------------------------------------

Description

Fit a mean-dependent trend to the per-gene variances for each blocking level, and decompose them to biological and technical components.

This has now been deprecated in favor of the `block` argument in [modelGeneVar](#) and related functions.

Usage

```

multiBlockVar(x, block, make.tech.trend=FALSE, trend.args=list(),
             dec.args=list(), assay.type="logcounts", ...)

```

Arguments

<code>x</code>	A SingleCellExperiment object containing log-normalized expression values, computed with multiBlockNorm .
<code>block</code>	A factor specifying the blocking level for each cell in <code>x</code> .
<code>make.tech.trend</code>	A logical scalar indicating whether to use makeTechTrend to create the mean-variance trend.
<code>trend.args</code>	A list of named arguments to pass to trendVar if <code>make.tech.trend=FALSE</code> or makeTechTrend otherwise.
<code>dec.args</code>	A list of named arguments to pass to decomposeVar .
<code>assay.type</code>	A string or integer scalar specifying the assay in <code>x</code> to use for all calculations (except for makeTechTrend).
<code>...</code>	Additional arguments to pass to combineVar .

Details

This function models the variance of expression in each level of `block` separately. Each subset of cells is passed to [trendVar](#) (or [makeTechTrend](#)) to fit a block-specific trend, and then passed to [decomposeVar](#) to obtain block-specific biological and technical components. Results are consolidated across blocks using the [combineVar](#) function. The aim is to enable users to handle differences in the mean-variance relationship across, e.g., different experimental batches.

We assume that the size factors for the endogenous genes have the same mean as the size factors for the spike-ins *within* each block. This ensures that the spike-in normalized values are comparable to those of the endogenous genes. Centering should be performed by running `multiBlockNorm` before calling this function. Otherwise, a warning will be raised about non-centered size factors.

Value

A `DataFrame` is returned containing all components returned by `combineVar`, in addition to a `per.block` column. This additional column is a `DataFrame` containing nested `DataFrames`, each containing a result of `decomposeVar` for the corresponding level of block. The trend function from `trendVar` or `makeTechTrend` is also stored as `trend` in the metadata of the per-block nested `DataFrames`.

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

`trendVar`, `decomposeVar`, `combineVar`, `multiBlockNorm`

Examples

```
example(computeSpikeFactors) # Using the mocked-up data 'y' from this example.

# Normalizing (gene-based factors for genes, spike-in factors for spike-ins)
y <- computeSumFactors(y)
y <- computeSpikeFactors(y, general.use=FALSE)

# Setting up the blocking levels.
block <- sample(3, ncol(y), replace=TRUE)
y <- multiBlockNorm(y, block)
out <- multiBlockVar(y, block=block)

# Creating block-level plots.
par(mfrow=c(1,3))
is.spike <- isSpike(y)
for (x in as.character(1:3)) {
  current <- out$per.block[[x]]
  plot(current$mean, current$total, col="black", pch=16)
  points(current$mean[is.spike], current$total[is.spike], col="red", pch=16)
  curve(metadata(current)$trend(x), col="dodgerblue", lwd=2, add=TRUE)
}
```

overlapExprs	<i>Overlap expression profiles</i>
--------------	------------------------------------

Description

Compute the gene-specific overlap in expression profiles between two groups of cells. This is deprecated in favor of calling [findMarkers](#) with `type="wilcox"`.

Usage

```
## S4 method for signature 'ANY'
overlapExprs(x, groups, gene.names=rownames(x), block=NULL,
             pval.type=c("any", "some", "all"), direction=c("any", "up", "down"), tol=1e-8,
             log.p=FALSE, full.stats=FALSE, subset.row=NULL, BPPARAM=SerialParam())

## S4 method for signature 'SingleCellExperiment'
overlapExprs(x, ..., subset.row=NULL, assay.type="logcounts",
             get.spikes=FALSE)
```

Arguments

<code>x</code>	A numeric matrix of expression values, where each column corresponds to a cell and each row corresponds to an endogenous gene. Alternatively, a <code>SingleCellExperiment</code> object containing such a matrix.
<code>groups</code>	A vector of group assignments for all cells.
<code>gene.names</code>	A character vector of gene names with one value for each row of <code>x</code> .
<code>block</code>	A factor specifying the blocking level for each cell.
<code>pval.type</code>	A string specifying the type of combined p-value to be computed, i.e., Simes' or IUT.
<code>direction</code>	A string specifying which direction of change in expression should be used to rank genes in the output.
<code>tol</code>	A numeric scalar specifying the tolerance with which ties are considered.
<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 (i.e., raw and BH-adjusted p-values) should be returned for each pairwise comparison.
<code>subset.row</code>	See ?" scran-gene-selection ".
<code>BPPARAM</code>	A <code>BiocParallelParam</code> object to use in <code>bplapply</code> for parallel processing.
<code>...</code>	Additional arguments to pass to the matrix method.
<code>assay.type</code>	A string specifying which assay values to use, e.g., "counts" or "logcounts".
<code>get.spikes</code>	See ?" scran-gene-selection ".

Details

This function provides a convenience wrapper for marker gene identification, based on running [pairwiseWilcox](#) and passing the result to [combineMarkers](#). All of the arguments above are supplied directly to one of these two functions.

Note that `log.p` only affects the combined p-values and FDRs. If `full.stats=TRUE`, the p-values for each pairwise comparison will be log-transformed regardless of the value of `log.p`.

Value

A named list of [DataFrames](#), each of which contains a sorted marker gene list for the corresponding cluster. See [?combineMarkers](#) for more details on the output format.

See Also

See [pairwiseWilcox](#) and [combineMarkers](#) for the component functions.

See [findMarkers](#) for the equivalent function using t-tests.

Examples

```
# Using the mocked-up data 'y2' from this example.
example(computeSpikeFactors)
y2 <- normalize(y2)
groups <- sample(3, ncol(y2), replace=TRUE)
out <- overlapExprs(y2, groups, subset.row=1:10)
```

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, groups, block = NULL, restrict = NULL,
  exclude = NULL, direction = c("any", "up", "down"),
  threshold = 1e-08, lfc = 0, log.p = FALSE,
  gene.names = rownames(x), clusters = NULL, subset.row = NULL,
  BPPARAM = SerialParam())
```

Arguments

x	A numeric matrix-like object of counts, where each column corresponds to a cell and each row corresponds to a gene.
groups	A vector specifying the group assignment for each cell.
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	A character vector of gene names with one value for each row of x.
clusters	Deprecated, same as groups.
subset.row	See ?" scrn-gene-selection ".
BPPARAM	A BiocParallelParam object indicating how parallelization should be performed across genes.

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`, on which this function is based.

Examples

```

library(scater)
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, groups, block = NULL, design = NULL,
  restrict = NULL, exclude = NULL, direction = c("any", "up",
  "down"), lfc = 0, std.lfc = FALSE, log.p = FALSE,
  gene.names = rownames(x), clusters = NULL, subset.row = NULL,
  BPPARAM = SerialParam())

```

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.
groups	A vector specifying the group assignment for each cell.
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.

<code>std.lfc</code>	A logical scalar indicating whether log-fold changes should be standardized.
<code>log.p</code>	A logical scalar indicating if log-transformed p-values/FDRs should be returned.
<code>gene.names</code>	A character vector of gene names with one value for each row of <code>x</code> .
<code>clusters</code>	Deprecated, same as <code>groups</code> .
<code>subset.row</code>	See ?"scrans-gene-selection" .
<code>BPPARAM</code>	A BiocParallelParam object indicating whether and how parallelization should be performed across genes.

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.

Examples

```
library(scater)
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, groups, block = NULL, restrict = NULL,
  exclude = NULL, direction = c("any", "up", "down"), lfc = 0,
  log.p = FALSE, gene.names = rownames(x), clusters = NULL,
  subset.row = NULL, BPPARAM = SerialParam())
```

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.
groups	A vector specifying the group assignment for each cell.
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 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	A character vector of gene names with one value for each row of x.
clusters	Deprecated, same as groups.
subset.row	See ?“ scrans-gene-selection ”.
BPPARAM	A BiocParallelParam object indicating whether and how parallelization should be performed across genes.

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.

Examples

```
library(scater)
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
```

Parallel analysis

Parallel analysis for PCA

Description

Perform a parallel analysis to choose the number of principal components.

This is now deprecated in favor of `parallelPCA` from the **PCAtools** package.

Usage

```
## S4 method for signature 'ANY'
parallelPCA(x, subset.row=NULL, value=c("pca", "n", "lowrank"),
            min.rank=5, max.rank=100, niters=50, threshold=0.1, BSPARAM=ExactParam(),
            BPPARAM=SerialParam())

## S4 method for signature 'SingleCellExperiment'
parallelPCA(x, ..., subset.row=NULL,
            value=c("pca", "n", "lowrank"), assay.type="logcounts",
            get.spikes=FALSE, sce.out=TRUE)
```

Arguments

<code>x</code>	A numeric matrix of log-expression values for <code>parallelPCA</code> , <code>ANY</code> -method, or a <code>SingleCellExperiment</code> object containing such values for <code>parallelPCA</code> , <code>SingleCellExperiment</code> -method.
<code>subset.row</code>	See ?" scrn-gene-selection ".
<code>value</code>	A string specifying the type of value to return; the PCs, the number of retained components, or a low-rank approximation.
<code>min.rank</code> , <code>max.rank</code>	Integer scalars specifying the minimum and maximum number of PCs to retain.
<code>niters</code>	Integer scalar specifying the number of iterations to use for the parallel analysis.
<code>threshold</code>	Numeric scalar representing the “p-value” threshold above which PCs are to be ignored.
<code>BSPARAM</code>	A BiocSingularParam object specifying the algorithm to use for PCA.
<code>BPPARAM</code>	A BiocParallelParam object specifying how the iterations should be parallelized.
<code>...</code>	Further arguments to pass to <code>denoisePCA</code> , <code>ANY</code> -method.
<code>assay.type</code>	A string specifying which assay values to use.
<code>get.spikes</code>	See ?" scrn-gene-selection ".
<code>sce.out</code>	A logical scalar specifying whether a modified <code>SingleCellExperiment</code> object should be returned.

Details

This function performs Horn’s parallel analysis to decide how many PCs to retain in a principal components analysis. Parallel analysis involves permuting the expression vector for each gene and repeating the PCA to obtain the fractions of variance explained under a random null model. The number of PCs to retain is determined by the intersection of the “fraction explained” lines on a scree plot. This is justified as discarding PCs that explain less variance than would be expected under a random model.

In practice, we discard all PCs from the first PC that has a fraction explained *similar* to that under the null. A PC is considered similar if the permuted fractions exceed the observed fraction in more than `threshold` of iterations. (For want of a better word, we have described this as a “p-value” threshold, though it is not interpretable as a measure of significance.) This is a more conservative criterion than discarding PCs with fractions below the average null fraction, which tends to overstate the rank in noisy datasets. Note that the number of PCs will be coerced to lie between `min.rank` and `max.rank`.

This function can be sped up by specifying `BSPARAM=Ir1baParam()` or similar, to use approximate strategies for performing the PCA. Another option is to set `BPPARAM` to perform the iterations in parallel.

Value

For `parallelPCA, ANY-method`, a numeric matrix is returned containing the selected PCs (columns) for all cells (rows) if `value="pca"`. If `value="n"`, it will return an integer scalar specifying the number of retained components. If `value="lowrank"`, it will return a low-rank approximation of `x` with the *same* dimensions.

For `parallelPCA, SingleCellExperiment-method`, the return value is the same as `parallelPCA, ANY-method` if `sce.out=FALSE` or `value="n"`. Otherwise, a `SingleCellExperiment` object is returned that is a modified version of `x`. If `value="pca"`, the modified object will contain the PCs as the "PCA" entry in the `reducedDims` slot. If `value="lowrank"`, it will return a low-rank approximation in assays slot, named "lowrank".

In all cases, the fractions of variance explained by the first `max.rank` PCs will be stored as the "percentVar" attribute in the return value. Fractions of variance explained by these PCs after each permutation iteration are also recorded as a matrix in "permuted.percentVar".

Author(s)

Aaron Lun

References

Buja A and Eyuboglu N (1992). Remarks on Parallel Analysis. *Multivariate Behav. Res.*, 27:509-40.

See Also

[denoisePCA](#)

Examples

```
# Mocking up some data.
ngenes <- 1000
means <- 2^runif(ngenes, 6, 10)
dispersions <- 10/means + 0.2
nsamples <- 50
counts <- matrix(rnbinom(ngenes*nsamples, mu=means,
                        size=1/dispersions), ncol=nsamples)

# Choosing the number of PCs
lcounts <- log2(counts + 1)
parallelPCA(lcounts, min.rank=0, value="n")
```

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 = cluster_walktrap, BSPARAM = bsparam(),
      BPPARAM = SerialParam(), block = NULL,
      block.BPPARAM = SerialParam(), ...)

## S4 method for signature 'SingleCellExperiment'
quickCluster(x, subset.row = NULL, ...,
      assay.type = "counts", get.spikes = FALSE)
```

Arguments

x	A numeric count matrix where rows are genes and columns are cells. Alternatively, a SingleCellExperiment 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 buildSNNGraph for method="igraph". or to cutreeDynamic for method="hclust". For the SingleCellExperiment 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.

<code>assay.type</code>	A string specifying which assay values to use.
<code>get.spikes</code>	See ?" scran-gene-selection ".

Details

This function provides a convenient wrapper to quickly define clusters of a minimum size `min.size`. 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.

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 "0" 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.

[scaledColRanks](#), to get the rank matrix directly.

Examples

```
library(scater)
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)
```

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, groups, normalize = TRUE,
  prepFUN = NULL, min.ncells = 50, clusterFUN = NULL,
  format = "%s.%s")

## S4 method for signature 'SingleCellExperiment'
quickSubCluster(x, groups,
  normalize = TRUE, prepFUN = NULL, min.ncells = 50,
  clusterFUN = NULL, format = "%s.%s", assay.type = "counts")
```

Arguments

x	A matrix of counts or log-normalized expression values (if <code>normalize=FALSE</code>). Alternatively, a SingleCellExperiment object containing such a matrix.
...	Further arguments to pass to specific methods.
groups	A vector of group assignments for all cells, usually corresponding to cluster identities.
normalize	Logical scalar indicating whether each subset of x should be log-transformed prior to further analysis.
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.
format	A string to be passed to <code>sprintf</code> , specifying how the subclusters should be named with respect to the parent level in groups and the level returned by <code>clusterFUN</code> .
assay.type	String or integer scalar specifying the relevant assay.

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 graph-based clustering with `buildSNNGraph` and `cluster_walktrap`. 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

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.

Author(s)

Aaron Lun

See Also

`quickCluster`, for a related function to quickly obtain clusters.

Examples

```
library(scater)
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)
    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 'SingleCellExperiment'
sandbag(x, phases, subset.row = NULL,
        ..., assay.type = "counts", get.spikes = FALSE)
```

Arguments

x	A numeric matrix of gene expression values where rows are genes and columns are cells. Alternatively, a SingleCellExperiment object containing such a matrix.
...	For the generic, additional arguments to pass to specific methods. For the SingleCellExperiment 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.
fraction	A numeric scalar specifying the minimum fraction to define a marker gene pair.
subset.row	See ?" scran-gene-selection ".
assay.type	A string specifying which assay values to use, e.g., "counts" or "logcounts".
get.spikes	See ?" scran-gene-selection ".

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.

By default, `get.spikes=FALSE` which means that any rows corresponding to spike-in transcripts will not be considered when picking markers. This is because the amount of spike-in RNA added will vary between experiments and will not be a robust predictor. Nonetheless, if all rows are required, users can set `get.spikes=TRUE`.

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(scater)
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)
```

Arguments

<code>x</code>	A numeric matrix-like object containing cells in columns and features in the rows.
<code>subset.row</code>	A logical, integer or character scalar indicating the rows of <code>x</code> to use, see ?"scrn-gene-selection" .
<code>min.mean</code>	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 <code>NULL</code> .
<code>transposed</code>	A logical scalar specifying whether the output should be transposed.
<code>as.sparse</code>	A logical scalar indicating whether the output should be sparse.
<code>withDimnames</code>	A logical scalar specifying whether the output should contain the dimnames of <code>x</code> .

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(scater)
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)
```

 technicalCV2

Model the technical coefficient of variation

Description

Model the technical coefficient of variation as a function of the mean, and determine the significance of highly variable genes.

This has been deprecated in favor of the [modelGeneCV2](#) function.

Usage

```
## S4 method for signature 'ANY'
technicalCV2(x, is.spike, sf.cell=NULL, sf.spike=NULL,
             cv2.limit=0.3, cv2.tol=0.8, min.bio.disp=0.25)

## S4 method for signature 'SingleCellExperiment'
technicalCV2(x, spike.type=NULL, ..., assay.type="counts")
```

Arguments

x	A numeric matrix of counts, where each column corresponds to a cell and each row corresponds to a spike-in transcript. Alternatively, a <code>SingleCellExperiment</code> object that contains such values.
is.spike	A vector indicating which rows of x correspond to spike-in transcripts.
sf.cell	A numeric vector containing size factors for endogenous genes.
sf.spike	A numeric vector containing size factors for spike-in transcripts.
cv2.limit, cv2.tol	Numeric scalars that determine the minimum mean abundance for the spike-in transcripts to be used for trend fitting.
min.bio.disp	A numeric scalar specifying the minimum biological dispersion.
spike.type	A character vector containing the names of the spike-in sets to use.
...	Additional arguments to pass to <code>technicalCV2, ANY-method</code> .
assay.type	A string specifying which assay values to use.

Details

This function will estimate the squared coefficient of variation (CV2) and mean for each spike-in transcript. A mean-dependent trend is fitted to the CV2 values for the transcripts using a Gamma GLM with [glmGamFit](#). Only high-abundance transcripts are used for stable trend fitting. (Specifically, a mean threshold is selected by taking all transcripts with CV2 above `cv2.limit`, and taking the quantile of this subset at `cv2.tol`. A warning will be thrown and all spike-ins will be used if the subset is empty.)

The trend is used to determine the technical CV2 for each endogenous gene based on its mean. To identify highly variable genes, the null hypothesis is that the total CV2 for each gene is less than or equal to the technical CV2 plus `min.bio.disp`. Deviations from the null are identified using a chi-squared test. The additional `min.bio.disp` is necessary for a ratio-based test, as otherwise genes with large relative (but small absolute) CV2 would be favoured.

For `technicalCV2, ANY-method`, the rows corresponding to spike-in transcripts are specified with `is.spike`. These rows will be used for trend fitting, while all other rows are treated as endogenous genes. If either `sf.cell` or `sf.spike` are not specified, the `estimateSizeFactorsForMatrix` function is applied to compute size factors.

For `technicalCV2, SingleCellExperiment-method`, transcripts from spike-in sets named in `spike.type` will be used for trend fitting. If `spike.type=NULL`, all spike-in sets listed in `x` will be used. Size factors for the endogenous genes are automatically extracted via `sizeFactors`. Spike-in-specific size factors for `spike.type` are extracted from `x`, if available; otherwise they are set to the size factors for the endogenous genes. Note that the spike-in-specific factors must be the same for each set in `spike.type`.

Users can also set `is.spike` to `NA` in `technicalCV2, ANY-method`; or `spike.type` to `NA` in `technicalCV2, SingleCellExperiment-method`. In such cases, all rows will be used for trend fitting, and (adjusted) p-values will be reported for all rows. This should be used in cases where there are no spike-ins. Here, the assumption is that most endogenous genes do not exhibit high biological variability and thus can be used to model technical variation.

Value

A data frame is returned containing one row per row of `x` (including both endogenous genes and spike-in transcripts). Each row contains the following information:

`mean`: A numeric field, containing mean (scaled) counts for all genes and transcripts.

`var`: A numeric field, containing the variances for all genes and transcripts.

`cv2`: A numeric field, containing CV2 values for all genes and transcripts.

`trend`: A numeric field, containing the fitted value of the trend in the CV2 values. Note that the fitted value is reported for all genes and transcripts, but the trend is only fitted using the transcripts.

`p.value`: A numeric field, containing p-values for all endogenous genes (`NA` for rows corresponding to spike-in transcripts).

`FDR`: A numeric field, containing adjusted p-values for all genes.

Author(s)

Aaron Lun, based on code from Brennecke et al. (2013)

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

[glmGamFit](#), [estimateSizeFactorsForMatrix](#)

Examples

```
# Mocking up some data.
ngenes <- 10000
means <- 2*runif(ngenes, 6, 10)
dispersions <- 10/means + 0.2
nsamples <- 50
counts <- matrix(rnbinom(ngenes*nsamples, mu=means, size=1/dispersions), ncol=nsamples)
```

```

is.spike <- logical(ngenes)
is.spike[seq_len(500)] <- TRUE

# Running it directly on the counts.
out <- technicalCV2(counts, is.spike)
head(out)
plot(out$mean, out$cv2, log="xy")
points(out$mean, out$trend, col="red", pch=16, cex=0.5)

# Same again with an SingleCellExperiment.
rownames(counts) <- paste0("X", seq_len(ngenes))
colnames(counts) <- paste0("Y", seq_len(nsamples))
X <- SingleCellExperiment(list(counts=counts))
isSpike(X, "Spikes") <- is.spike

# Dummying up some size factors (for convenience only, use computeSumFactors() instead).
sizeFactors(X) <- 1
X <- computeSpikeFactors(X, general.use=FALSE)

# Running it.
out <- technicalCV2(X, spike.type="Spikes")
head(out)

```

testVar

Test for significantly large variances

Description

Test for whether the total variance exceeds that expected under some null hypothesis, for sample variances estimated from normally distributed observations.

This function is deprecated.

Usage

```
testVar(total, null, df, design=NULL, test=c("chisq", "f"), second.df=NULL, log.p=FALSE)
```

Arguments

total	A numeric vector of total variances for all genes.
null	A numeric scalar or vector of expected variances under the null hypothesis for all genes.
df	An integer scalar specifying the degrees of freedom on which the variances were estimated.
design	A design matrix, used to determine the degrees of freedom if df is missing.
test	A string specifying the type of test to perform.
second.df	A numeric scalar specifying the second degrees of freedom for the F-distribution when test="f".
log.p	A logical scalar indicating whether log-transformed p-values should be returned.

Details

The null hypothesis is that the true variance for each gene is equal to null. (Technically, it is that the variance is equal to or less than this value, but the most conservative test is obtained at equality.) If `test="chisq"`, variance estimates are assumed to follow a chi-squared distribution on `df` degrees of freedom and scaled by `null/df`. This is used to compute a p-value for `total` being greater than null. The underlying assumption is that the observations are normally distributed under the null, which is reasonable for log-counts with low-to-moderate dispersions.

The aim is to use this function to identify significantly highly variable genes (HVGs). For example, the null vector can be set to the values of the trend fitted to the spike-in variances. This will identify genes with variances significantly greater than technical noise. Alternatively, it can be set to the trend fitted to the cellular variances, which will identify those that are significantly more variable than the bulk of genes. Selecting HVGs on p-values is better than using `total -null`, as the latter is less precise when null is large.

If `test="f"`, the true variance of each spike-in transcript is assumed to be sampled from a scaled inverse chi-squared distribution. This accounts for any inflated scatter around the trend due to differences in amplification efficiency between transcripts. As a result, the gene-wise variance estimates should be F-distributed around the trend under the null. The second degrees of freedom is estimated from the scatter around the trend in `trendVar` using `fitFDistRobustly`, and needs to be supplied to `second.df` to calculate an appropriate p-value.

Value

A numeric vector of p-values for all genes.

Author(s)

Aaron Lun

References

Law CW, Chen Y, Shi W and Smyth GK (2014). voom: precision weights unlock linear model analysis tools for RNA-seq read counts *Genome Biol.* 15(2), R29.

See Also

[trendVar](#), [decomposeVar](#), [fitFDistRobustly](#)

Examples

```
set.seed(100)
null <- 100/runif(1000, 50, 2000)
df <- 30
total <- null * rchisq(length(null), df=df)/df

# Direct test:
out <- testVar(total, null, df=df)
hist(out)

# Rejecting the null:
alt <- null * 5 * rchisq(length(null), df=df)/df
out <- testVar(alt, null, df=df)
plot(alt[order(out)]-null)

# Focusing on genes that have high absolute increases in variability:
```

```
out <- testVar(alt, null+0.5, df=df)
plot(alt[order(out)]-null)
```

trendVar	<i>Fit a variance trend</i>
----------	-----------------------------

Description

Fit a mean-dependent trend to the gene-specific variances in single-cell RNA-seq data.

Usage

```
## S4 method for signature 'ANY'
trendVar(x, method=c("loess", "spline"), parametric=FALSE,
         loess.args=list(), spline.args=list(), nls.args=list(), block=NULL,
         design=NULL, weighted=TRUE, min.mean=0.1, subset.row=NULL,
         BPPARAM=SerialParam())

## S4 method for signature 'SingleCellExperiment'
trendVar(x, subset.row=NULL, ..., assay.type="logcounts", use.spikes=TRUE)
```

Arguments

x	A numeric matrix-like object of normalized log-expression values, where each column corresponds to a cell and each row corresponds to a spike-in transcript. Alternatively, a <code>SingleCellExperiment</code> object that contains such values.
method	A string specifying the algorithm to use for smooth trend fitting.
parametric	A logical scalar indicating whether a parametric curve should be fitted prior to smoothing.
loess.args	A named list of arguments to pass to <code>loess</code> when <code>method="loess"</code> .
spline.args	A named list of arguments to pass to <code>robustSmoothSpline</code> when <code>method="spline"</code> .
nls.args	A named list of arguments to pass to <code>nls</code> when <code>parametric=TRUE</code> .
block	A factor specifying the blocking level for each cell.
design	A numeric matrix describing the uninteresting factors contributing to expression in each cell. Alternatively, a single factor can be supplied for one-way layouts.
weighted	A logical scalar indicated whether weighted trend fitting should be performed when <code>block!=NULL</code> .
min.mean	A numeric scalar specifying the minimum mean log-expression in order for a gene to be used for trend fitting.
subset.row	See ?" scran-gene-selection ".
BPPARAM	A <code>BiocParallelParam</code> object indicating whether and how parallelization should be performed across genes.
...	Additional arguments to pass to <code>trendVar</code> , ANY-method.
assay.type	A string specifying which assay values in <code>x</code> to use.
use.spikes	A logical scalar specifying whether the trend should be fitted to variances for spike-in transcripts or endogenous genes.

Details

This function fits an abundance-dependent trend to the variance of the log-normalized expression for the spike-in transcripts. For SingleCellExperiment objects, these expression values are computed by `normalize` after setting the size factors, e.g., with `computeSpikeFactors`. Log-transformed values are used as these are more robust to genes/transcripts with strong expression in only one or two outlier cells. It also allows the fitted trend to be applied in downstream procedures that use log-transformed counts.

The mean and variance of the normalized log-counts is calculated for each spike-in transcript, and a trend is fitted to the variance against the mean for all transcripts. The fitted value of this trend represents technical variability due to sequencing, drop-outs during capture, etc. at a given mean. This assumes that a constant amount of spike-in RNA was added to each cell, such that any differences in observed expression are purely due to measurement error. Variance decomposition to biological and technical components for endogenous genes can then be performed later with `decomposeVar`.

Value

A named list is returned, containing:

mean: A numeric vector of mean log-expression values for all spike-in transcripts, if `block=NULL`. Otherwise, a numeric matrix of means where each row corresponds to a spike-in and each column corresponds to a level of `block`.

var: A numeric vector of the variances of log-expression values for all spike-in transcripts, if `block=NULL`. Otherwise, a numeric matrix of variances where each row corresponds to a spike-in and each column corresponds to a level of `block`.

resid.df: An integer scalar specifying the residual d.f. used for variance estimation of each spike-in transcript, if `block=NULL`. Otherwise, a integer vector where each entry specifies the residual d.f. used in each level of `block`.

block: A factor identical to the input `block`, only returned if it was not `NULL`.

design: A numeric matrix (or factor) identical to the input `design`, only returned if it was not `NULL` and `block=NULL`.

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

df2: A numeric scalar, specifying the second degrees of freedom for a scaled F-distribution describing the variability of variance estimates around the trend.

Trend fitting options

If `parametric=FALSE`, smoothing is performed directly on the log-variances. This is the default as it provides the most stable performance on arbitrary mean-variance relationships.

If `parametric=TRUE`, 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`. A smoothing algorithm is then applied to the log-ratios of the variance to the fitted value for each gene. 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.

The `method` argument specifies the smoothing algorithm to be applied on the log-ratios/variances. By default, a robust loess curve is used for trend fitting via `loess`. This provides a fairly flexible

fit while protecting against genes with very large or very small variances. Arguments to `loess` are specified with `loess.args`, with defaults of `span=0.3`, `family="symmetric"` and `degree=1` unless otherwise specified. Some experimentation with these parameters may be required to obtain satisfactory results.

If `method="spline"`, smoothing will instead be performed using `robustSmoothSpline` (a robustified version of the `smooth.spline` function in the `aroma.light` package). Splines can be more effective than loess at capturing smooth curves with strong non-linear gradients. Spline fitting is also faster for very large numbers of points, which may occur in large data sets with many genes and/or many levels of block. Arguments are specified with `spline.args` with a default `df=4` and "symmetric" robust weighting. Users can force the use of cross-validation by supplying an invalid `df=0`, but this tends to result in a rather bumpy trend.

The `trendVar` function will produce an output trend function with which fitted values can be computed. When extrapolating to values below the smallest observed mean, the output function will approach zero. 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.

Handling uninteresting factors of variation

There are three approaches to handling unwanted factors of variation. The simplest approach is to use a design matrix containing the uninteresting factors can be specified in `design`. This will fit a linear model to the log-expression values for each gene, yielding an estimate for the residual variance. The trend is then fitted to the residual variance against the mean for each spike-in transcripts.

Another approach is to use `block`, where all cells in each level of the blocking factor are treated as a separate group. Means and variances are estimated within each group and the resulting sets of means/variances are pooled across all groups. The trend is then fitted to the pooled observations, where observations from different levels are weighted according to the residual d.f. used for variance estimation. This effectively multiplies the number of points by the number of levels in `block`. If both `block` and `design` are specified, `block` will take priority and `design` will be ignored.

The final approach is to subset the data set for each level of the blocking factor and model the variances within each block separately. This is done using the `multiBlockVar` function, see its documentation for more details. Separate modelling and trend fitting is the most correct approach if there are systematic differences in the size factors (spike-in or endogenous) between levels. With the other two methods, such differences would be normalized out in the full log-expression matrix, preventing proper estimation of the level-specific abundance.

Assuming there are no differences in the size factors between levels, we suggest using `block` wherever possible instead of `design`. This is because the use of `block` preserves differences in the means/variances between levels of the factor. In contrast, using `design` will effectively compute an average mean/variance. This may yield an inaccurate representation of the trend, as the fitted value at an average mean may not be equal to the average variance for non-linear trends. Nonetheless, we still support `design` as it can accommodate additive models, whereas `block` only handles one-way layouts.

Additional notes on row selection

The selection of spike-in transcripts can be adjusted in `trendVar, SingleCellExperiment-method` using the `use.spikes` method.

- By default, `use.spikes=TRUE` which means that only rows labelled as spike-ins with `isSpike(x)` will be used. An error will be raised if no rows are labelled as spike-in transcripts.
- If `use.spikes=FALSE`, only the rows *not* labelled as spike-in transcripts will be used.

- If use.spikes=NA, every row will be used for trend fitting, regardless of whether it corresponds to a spike-in transcript or not.

If use.spikes=FALSE, this implies that trendVar will be applied to the endogenous genes in the SingleCellExperiment object. For trendVar, ANY-method, it is equivalent to manually supplying a matrix of normalized expression for endogenous genes. This assumes that most genes exhibit technical variation and little biological variation, e.g., in a homogeneous population.

Low-abundance genes with mean log-expression below min.mean are not used in trend fitting, to 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. 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.

Users can directly specify which rows to use with subset.row. See ?"scran-gene-selection" for how the different options interact with each other.

Note that features are not used under any circumstance if they have a variance of zero. These are not compatible with trend fitting on the log-scale. In practice, it should be effectively impossible to obtain variances of zero for any expressed gene, though some care may be required with simulated data sets. Any NA variances are similarly ignored.

Note that, depending on the output of the various row selection parameters, the number of used rows may be much less than the number of rows of x. In pathological cases, this may result in an error stating that insufficient points are available for the trend points.

Warning on size factor centring

If assay.type="logcounts", trendVar, SingleCellExperiment-method will attempt to determine if the expression values were computed from counts via `normalize`. If so, a warning will be issued if the size factors are not centred at unity. This is because different size factors are typically used for endogenous genes and spike-in transcripts. If these size factor sets are not centred at the same value, there will be systematic differences in abundance between these features. This precludes the use of a spike-in fitted trend with abundances for endogenous genes in `decomposeVar`.

For other expression values and in trendVar, ANY-method, the onus is on the user to ensure that normalization (i) does not introduce differences in abundance between spike-in and endogenous features, while (ii) preserving differences in abundance within the set of endogenous or spike-in features. In short, the scaling factors used to normalize each feature should have the same mean across all cells. This ensures that spurious differences in abundance are not introduced by the normalization process.

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

Lun ATL (2018). Description of the HVG machinery in *scran*. <https://github.com/LTLA/HVGDetection2018>

See Also

`nls`, `loess`, `decomposeVar`, `computeSpikeFactors`, `computeSumFactors`, `normalize`

Examples

```
example(computeSpikeFactors) # Using the mocked-up data 'y' from this example.

# Normalizing (gene-based factors for genes, spike-in factors for spike-ins)
y <- computeSumFactors(y)
y <- computeSpikeFactors(y, general.use=FALSE)
y <- normalize(y)

# Fitting a trend to the spike-ins.
fit <- trendVar(y)
plot(fit$mean, fit$var)
curve(fit$trend(x), col="red", lwd=2, add=TRUE)

# Fitting a trend to the endogenous genes.
fit.g <- trendVar(y, use.spikes=FALSE)
plot(fit.g$mean, fit.g$var)
curve(fit.g$trend(x), col="red", lwd=2, add=TRUE)
```

Index

*Topic **normalization**

multiBlockNorm, 81

*Topic **variance**

decomposeVar, 35

Distance-to-median, 41

improvedCV2, 61

multiBlockVar, 83

technicalCV2, 106

testVar, 108

trendVar, 110

altExp, 68, 70, 78, 80

altExps, 19, 20

as.undirected, 10

binom.test, 88

BiocNeighborParam, 5, 43

BiocParallelParam, 5, 27, 30, 39, 44, 52, 66,
69, 72, 75, 79, 87, 90, 93, 96, 98

BiocSingularParam, 5, 39, 43, 96, 98

bootstrapCluster, 3

buildKNNGraph (buildSNNGraph), 4

buildKNNGraph, ANY-method
(buildSNNGraph), 4

buildKNNGraph, SingleCellExperiment-method
(buildSNNGraph), 4

buildSNNGraph, 4, 11, 57, 98–100, 102

buildSNNGraph, ANY-method
(buildSNNGraph), 4

buildSNNGraph, SingleCellExperiment-method
(buildSNNGraph), 4

calcAverage, 56

calculateAverage, 24, 99

calculateSumFactors
(computeSumFactors), 20

calculateSumFactors, ANY-method
(computeSumFactors), 20

calculateSumFactors, SummarizedExperiment-method
(computeSumFactors), 20

cleanSizeFactors, 8, 23

cluster_louvain, 7

cluster_walktrap, 7, 57, 102

clusterModularity, 10

combineCV2 (combineVar), 17

combineMarkers, 11, 49, 50, 60, 85–87, 90, 93

combinePValues, 14, 15, 17, 18, 66, 67, 69,
70, 72, 73, 75, 76, 78, 80

combineVar, 17, 83, 84

computeSpikeFactors, 19, 111, 113

computeSumFactors, 9, 20, 99, 100, 113

convertTo, 25

cor, 31, 32

correlateGenes, 26, 32

correlateNull, 27, 29, 30

correlatePairs, 26–28, 29

correlatePairs, ANY-method
(correlatePairs), 29

correlatePairs, SingleCellExperiment-method
(correlatePairs), 29

cutree, 57

cutreeDynamic, 98–100

cyclone, 32, 103, 104

cyclone, ANY-method (cyclone), 32

cyclone, SingleCellExperiment-method
(cyclone), 32

DataFrame, 12, 17, 26, 30, 39, 47, 49, 57–59,
62, 66, 69, 72, 76, 79, 86, 87, 90, 94

decomposeVar, 35, 83, 84, 109, 111, 113

decomposeVar, ANY, list-method
(decomposeVar), 35

decomposeVar, SingleCellExperiment, list-method
(decomposeVar), 35

denoisePCA, 38, 97–99

denoisePCANumber (denoisePCA), 38

DESeqDataSetFromMatrix, 25

dgCMatrix, 99

DGEList, 25

Distance-to-median, 41

DM (Distance-to-median), 41

doubletCells, 42, 47

doubletCells, ANY-method (doubletCells),
42

doubletCells, SingleCellExperiment-method
(doubletCells), 42

doubletCluster, 45, 45

- doubletCluster, ANY-method (doubletCluster), 45
- doubletCluster, SingleCellExperiment-method (doubletCluster), 45
- estimateSizeFactorsForMatrix, 107
- findKNN, 6, 7
- findMarkers, 14, 46, 47, 48, 85, 86
- findMarkers, ANY-method (findMarkers), 48
- findMarkers, SingleCellExperiment-method (findMarkers), 48
- findNeighbors, 43
- fitFDistRobustly, 109
- fitTrendCV2, 50, 65–69, 71
- fitTrendPoisson, 52, 75
- fitTrendVar, 39, 41, 52, 53, 53, 72–81
- Gene selection, 55
- getClusteredPCs, 56
- getDenoisedPCs (denoisePCA), 38
- getDenoisedPCs, ANY-method (denoisePCA), 38
- getDenoisedPCs, SingleCellExperiment-method (denoisePCA), 38
- getTopHVGs, 58
- getTopMarkers, 59
- glmGamFit, 106, 107
- graph, 7, 10
- improvedCV2, 61
- improvedCV2, ANY-method (improvedCV2), 61
- improvedCV2, SingleCellExperiment-method (improvedCV2), 61
- IrlbaParam, 7
- isSpike, 56
- librarySizeFactors, 19, 20, 22, 44, 46, 66, 70, 76, 79
- List, 12, 57, 60, 102
- loess, 110–113
- logNormCounts, 24, 46, 90
- make_graph, 7
- makeTechTrend, 63, 83, 84
- modelGeneCV2, 17, 18, 52, 59, 61, 65, 69, 71, 106
- modelGeneCV2, ANY-method (modelGeneCV2), 65
- modelGeneCV2, SingleCellExperiment-method (modelGeneCV2), 65
- modelGeneCV2WithSpikes, 17, 52, 66, 68
- modelGeneCV2WithSpikes, ANY-method (modelGeneCV2WithSpikes), 68
- modelGeneCV2WithSpikes, SingleCellExperiment-method (modelGeneCV2WithSpikes), 68
- modelGeneVar, 17, 18, 39, 41, 54, 55, 58, 59, 71, 77, 79, 81, 83, 99, 102
- modelGeneVar, ANY-method (modelGeneVar), 71
- modelGeneVar, SingleCellExperiment-method (modelGeneVar), 71
- modelGeneVarByPoisson, 74
- modelGeneVarByPoisson, ANY-method (modelGeneVarByPoisson), 74
- modelGeneVarByPoisson, SingleCellExperiment-method (modelGeneVarByPoisson), 74
- modelGeneVarWithSpikes, 17, 19, 41, 54, 55, 72, 74, 75, 77, 82
- modelGeneVarWithSpikes, ANY-method (modelGeneVarWithSpikes), 77
- modelGeneVarWithSpikes, SingleCellExperiment-method (modelGeneVarWithSpikes), 77
- modularity, 10, 11
- multiBlockNorm, 81, 83, 84
- multiBlockVar, 37, 82, 83, 112
- neighborsToKNNGraph (buildSNNGraph), 4
- neighborsToSNNGraph (buildSNNGraph), 4
- newCellDataSet, 25
- nls, 8, 9, 51, 54, 110, 111, 113
- nls.control, 8
- normalize, 64, 65, 82, 111, 113
- overlapExprs, 85
- overlapExprs, ANY-method (overlapExprs), 85
- overlapExprs, SingleCellExperiment-method (overlapExprs), 85
- pairwiseBinom, 13, 48, 50, 86
- pairwiseTTests, 12–14, 48–50, 60, 89, 93
- pairwiseWilcox, 13, 14, 48, 50, 85, 86, 92
- Parallel analysis, 95
- parallelPCA (Parallel analysis), 95
- parallelPCA, ANY-method (Parallel analysis), 95
- parallelPCA, SingleCellExperiment-method (Parallel analysis), 95
- quickCluster, 4, 23, 24, 97, 102, 105
- quickCluster, ANY-method (quickCluster), 97
- quickCluster, SingleCellExperiment-method (quickCluster), 97
- quickSubCluster, 101
- quickSubCluster, ANY-method (quickSubCluster), 101

quickSubCluster, SingleCellExperiment-method
(quickSubCluster), 101

reducedDims, 5, 40, 102

Rle, 55

robustSmoothSpline, 110

runmed, 42

runPCA, 57

runSVD, 7, 41

sandbag, 33, 34, 103

sandbag, ANY-method (sandbag), 103

sandbag, SingleCellExperiment-method
(sandbag), 103

scaledColRanks, 7, 98–100, 104

scran-gene-selection (Gene selection),
55

set.seed, 7

SimpleList, 47

SingleCellExperiment, 3, 5, 7, 19, 21, 25,
29, 33, 38, 43, 45, 48, 55, 65, 66, 68,
70, 72, 75, 76, 78–80, 98, 101–103

sizeFactors, 19, 21, 43, 62, 66, 70, 76, 79,
80, 107

smooth.spline, 61, 112

sprintf, 101

SummarizedExperiment, 21

t.test, 92

technicalCV2, 62, 63, 106

technicalCV2, ANY-method (technicalCV2),
106

technicalCV2, SingleCellExperiment-method
(technicalCV2), 106

testVar, 36, 37, 108

treat, 91

trendVar, 35–37, 56, 64, 65, 83, 84, 109, 110

trendVar, ANY-method (trendVar), 110

trendVar, SingleCellExperiment-method
(trendVar), 110

weightedLowess, 54

wilcox.test, 94, 95