Package 'scran'

October 16, 2018

Version 1.8.4

Date 2018-08-07

Title Methods for Single-Cell RNA-Seq Data Analysis

Maintainer Aaron Lun <alun@wehi.edu.au>

Depends R (>= 3.5), BiocParallel, SingleCellExperiment

Imports SummarizedExperiment, S4Vectors, BiocGenerics, Rcpp (>= 0.12.14), stats, methods, utils, graphics, grDevices, Matrix, scater, edgeR, limma, dynamicTreeCut, FNN, igraph, shiny, statmod, ggplot2, DT, viridis, DelayedArray, DelayedMatrixStats

Suggests testthat, BiocStyle, knitr, beachmat, HDF5Array, limSolve, org.Mm.eg.db, DESeq2, monocle, pracma, Biobase, irlba, aroma.light

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

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, correction of batch effects, identification of marker genes, and other common tasks in single-cell analysis workflows.

License GPL-3

NeedsCompilation yes

VignetteBuilder knitr

SystemRequirements C++11

LinkingTo beachmat, Rcpp, Rhdf5lib

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

RoxygenNote 6.0.1

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

git_branch RELEASE_3_7

git_last_commit bbf1d09

git_last_commit_date 2018-08-07

Date/Publication 2018-10-15

R topics documented:

build*NNGraph . . . 2
clusterModularity . . . 4
combineVar . . . 6
convertTo . . . 8
correlatePairs . . . 9
cyclone . . . 13
decomposeVar . . . 16
Deconvolution Methods . . . 19
Denoise with PCA . . . 22
Distance-to-median . . . 25
Explore Data . . . 26
findMarkers . . . 28
improvedCV2 . . . 32
makeTechTrend . . . 34
mnnCorrect . . . 36
multiBlockVar . . . 39
overlapExprs . . . 40
Parallel analysis . . . 43
Quick clustering . . . 45
sandbag . . . 47
Selector plot . . . 49
Spike-in normalization . . . 51
technicalCV2 . . . 52
testVar . . . 55
trendVar . . . 56

Index **61**

`build*NNGraph` *Build a nearest-neighbor graph*

Description

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

Usage

```
## S4 method for signature 'ANY'
buildSNNGraph(x, k=10, d=50, transposed=FALSE,
    pc.approx=FALSE, rand.seed=1000, irlba.args=list(),
    knn.args=list(), subset.row=NULL, BPPARAM=SerialParam())

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

## S4 method for signature 'ANY'
buildKNNGraph(x, k=10, d=50, directed=FALSE, transposed=FALSE,
    pc.approx=FALSE, rand.seed=1000, irlba.args=list(),
    knn.args=list(), subset.row=NULL, BPPARAM=SerialParam())
```

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

Arguments

x	A SingleCellExperiment object, or a matrix containing expression values for each gene (row) in each cell (column). If it is matrix, it can also be transposed.
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 k-NN search.
directed	A logical scalar indicating whether the output of buildKNNGraph should be a directed graph.
transposed	A logical scalar indicating whether x is transposed (i.e., rows are cells).
pc.approx	A logical scalar indicating whether approximate PCA should be performed.
subset.row	A logical, integer or character scalar indicating the rows of x to use.
irlba.args	A named list of additional arguments to pass to prcomp_irlba when pc.approx=TRUE.
knn.args	A named list of additional arguments to pass to get.knn, usually algorithm.
rand.seed	A numeric scalar specifying the seed for approximate PCA when pc.approx=TRUE. This can be set to NA to use the existing session seed.
BPPARAM	A BiocParallelParam object to use in bplapply for parallel processing.
...	Additional arguments to pass to buildSNNGraph,ANY-method.
assay.type	A string specifying which assay values to use.
get.spikes	A logical scalar specifying whether spike-in transcripts should be used.
use.dimred	A string specifying whether existing values in reducedDims(x) should be used.

Details

The buildSNNGraph method builds a shared nearest-neighbour graph using cells as nodes. For each cell, its k nearest neighbours are identified based on Euclidean distances in their expression profiles. An edge is drawn between all pairs of cells that share at least one neighbour. The weight of the edge between two cells is determined by the ranking of the shared nearest neighbors. More shared neighbors, or shared neighbors that are close to both cells, will yield larger weights.

The aim is to use the SNN graph to perform community-based clustering, using various methods in the **igraph** package. This is faster/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. The choice of k can be roughly interpreted as the minimum cluster size.

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 neighour relationships may not be reciprocal. However, by default, directed=FALSE such that an undirected graph is returned.

Value

An igraph-type 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 `x` to obtain the first `d` principal components. This is necessary in order to perform the k-NN search (done using the `get.knn` function) in reasonable time. By default, the first 50 components are chosen, which should retain most of the substructure in the data set. If `d` is `NA` or less than the number of cells, no dimensionality reduction is performed. If `pc.approx=TRUE`, `prcomp_irlba` will be used to quickly obtain the first `d` PCs.

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 `quickCluster` with `get.ranks=TRUE`. (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 is already transposed, `transposed=TRUE` avoids an unnecessary internal transposition.

By default, spike-in transcripts are removed from the expression matrix in `buildSNNGraph,SCESet-method`. However, any non-`NULL` setting of `subset.row` will override `get.spikes`. 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.

Author(s)

Aaron Lun

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

`get.knn`, `make_graph`

Examples

```
exprs <- matrix(rnorm(100000), ncol=100)
g <- buildSNNGraph(exprs)

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

---

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.values=FALSE)
```

Arguments

graph	A graph object from **igraph**, like that produced by buildSNNGraph.
clusters	A factor specifying the cluster identity for each node.
get.values	A logical scalar indicating whether the observed and expected edge 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 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. This allows users to determine which clusters are enriched for intra-cluster edges based on their high modularity scores. For comparison, clusterModularity also reports the modularity scores between pairs of 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).

Value

If get.values=FALSE, a symmetric 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 get.values=TRUE, a list is returned containing two symmetric 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, modularity

Examples

```
example(buildSNNGraph) # using the mocked-up graph in this example.

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

# Alternatively, compare the ratio of observed:expected.
out <- clusterModularity(g, clusters, get.values=TRUE)
log.ratio <- log2(out$observed/out$expected + 1)
image(log.ratio)
```

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=c("fisher", "z", "simes", "berger"), weighted=TRUE)
```

Arguments

...	Two or more DataFrames produced by decomposeVar.
method	String specifying how p-values are to be combined.
weighted	Logical scalar indicating whether weights should be used for combining statistics.

Details

This function is designed to merge results from multiple calls to decomposeVar, usually computed for different batches of cells. Separate variance decompositions are necessary in cases where different concentrations of spike-in have been added to the cells in each batch. This affects the technical mean-variance relationship and precludes the use of a common trend fit.

The output mean is computed as a weighted average of the means in each input DataFrame, where the weight is defined as the number of cells in that batch. This yields an equivalent value to the sample mean across all cells in all batches. Similarly, weighted averages are computed for all variance components, where the weight is defined as the residual d.f. used for variance estimation in each batch. This yields a variance equivalent to the residual variance obtained while blocking on the batch of origin.

Weighting can be turned off with weighted=FALSE. This may be useful to ensure that all batches contribute equally to the calculation of the combined statistics, avoiding cases where batches with many cells dominate the output. Of course, this comes at the cost of precision - large batches contain more information and *should* contribute more to the weighted average.

Value

A DataFrame with the same numeric fields as that produced by decomposeVar. Each field contains the average 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.

Combining p-values across batches

The default approach is to use method="fisher", which uses Fisher's method for combining p-values. This will test the global null hypothesis, i.e., that genes are not variable in any batch. As a result, it will identify genes that are highly variable in *any* batch.

Another option is to use method="z", where Stouffer's Z-score method is used to combine p-values across batches. Each batch is weighted according to the residual d.f. used to perform the test. This

is less sensitive that Fisher's method to low p-values in only a single batch, instead favouring genes that are significant in multiple batches.

Both Fisher's and Stouffer's methods assume independence between batches. If this is not the case, Simes' method should be used by setting `method="simes"`. This is more robust to correlations between tests (Sarkar and Chung, 1997; see also similar work on the related Benjamini-Hochberg method).

To identify genes that are detected as highly variable in *all* batches, Berger's IUT can be used by setting `method="berger"`. This defines the combined p-value as the maximum across batches for each gene. Needless to say, this is quite a conservative approach.

Only `method="z"` will perform any weighting of batches, and only if `weighted=TRUE`. In all other cases, all batches are assigned equal weight.

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.

See Also

`decomposeVar`

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.

y1 <- y[,1:100]
y1 <- normalize(y1) # normalize separately after subsetting.
fit1 <- trendVar(y1)
results1 <- decomposeVar(y1, fit1)

y2 <- y[,1:100 + 100]
y2 <- normalize(y2) # normalize separately after subsetting.
fit2 <- trendVar(y2)
results2 <- decomposeVar(y2, fit2)

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

convertTo *Convert to other classes*

Description

Convert a SingleCellExperiment object into other classes for entry into other analysis pipelines.

Usage

```
## S4 method for signature 'SingleCellExperiment'
convertTo(x, type=c("edgeR", "DESeq2", "monocle"),
    row.fields=NULL, col.fields=NULL, ..., assay.type,
    use.all.sf=TRUE, subset.row=NULL, get.spikes=FALSE)
```

Arguments

x	A SingleCellExperiment object.
type	A string specifying the analysis for which the object should be prepared.
row.fields	Any set of indices specifying which columns of rowData(x) should be retained in the returned object.
col.fields	Any set of indices specifying which columns of colData(x) should be retained.
...	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.
use.all.sf	A logical scalar indicating whether multiple size factors should be used to generate the returned object.
subset.row	A logical, integer or character scalar indicating the rows of x to return.
get.spikes	A logical scalar specifying whether rows corresponding to spike-in transcripts should be returned.

Details

This function converts an SingleCellExperiment object into various other classes in preparation for entry into other analysis pipelines, as specified by type. Gene- and cell-specific data fields can be retained in the output object by setting row.fields and col.fields, respectively. Other arguments can be passed to the relevant constructors through the ellipsis.

By default, assay.type is set to "counts" such that count data is stored in the output object. This is consistent with the required inputs to analyses using count-based (e.g., negative binomial) models. Information about normalization is instead transmitted via size or normalization factors in the output object.

In all cases, rows corresponding to spike-in transcripts are removed from the output object by default. As such, rows in the returned object may not correspond directly to rows in x. Users should consider this when retrieving analysis results from these pipelines, e.g., match on row names in x before comparing to other results. This behaviour can be turned off by setting get.spikes=TRUE, such that all rows are retrieved in the output object. Users can also set subset.row to extract specific rows, in which case get.spikes is ignored.

For **edgeR** and **DESeq2**, different size factors for different rows (e.g., for spike-in sets) will be respected. For **edgeR**, an offset matrix will be constructed containing mean-centred log-size factors for each row. For **DESeq2**, a similar matrix will be constructed containing size factors scaled to

have a geometric mean of unity. This behaviour can be turned off with `use.all.sf=FALSE`, such that only `sizeFactors(x)` is used for normalization for all `type`. (These matrices are not generated if all rows correspond to `sizeFactors(x)`, as this information is already stored in the object.)

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 `rowData` and `colData` respectively.

Author(s)

Aaron Lun

See Also

DGEList, DESeqDataSetFromMatrix, newCellDataSet

Examples

```
example(computeSpikeFactors) # Using the mocked up data 'y' from this example.
sizeFactors(y) <- 2^rnorm(ncells) # Adding some additional embellishments.
rowData(y)$SYMBOL <- paste0("X", seq_len(nrow(y)))
y$other <- sample(LETTERS, ncells, replace=TRUE)

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

---

correlatePairs *Test for significant correlations*

Description

Identify pairs of genes that are significantly correlated based on a modified Spearman's rho.

Usage

```
correlateNull(ncells, iters=1e6, block=NULL, design=NULL, residuals=FALSE)

## S4 method for signature 'ANY'
correlatePairs(x, null.dist=NULL, tol=1e-8, iters=1e6,
    block=NULL, design=NULL, residuals=FALSE, lower.bound=NULL,
    use.names=TRUE, subset.row=NULL, pairings=NULL, per.gene=FALSE,
    cache.size=100L, BPPARAM=SerialParam())

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

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.
residuals	A logical scalar, deprecated.
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.
null.dist	A numeric vector of rho values under the null hypothesis.
BPPARAM	A BiocParallelParam object to use in bplapply for parallel processing.
tol	A numeric scalar indicating the maximum difference under which two expression values are tied.
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	A logical, integer or character vector indicating the rows of x to use to compute correlations.
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.
per.gene	A logical scalar specifying whether statistics should be summarized per gene.
lower.bound	A numeric scalar specifying the theoretical lower bound of values in x, only used when residuals=TRUE.
cache.size	An integer scalar specifying the number of cells for which ranked expression values are stored in memory. Smaller values can be used in machines with less memory, at the cost of processing speed.
...	Additional arguments to pass to correlatePairs,ANY-method.
assay.type	A string specifying which assay values to use.
get.spikes	A logical specifying whether spike-in transcripts should be used.

Details

The aim of the correlatePairs function is to identify 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.

An approximation of Spearman's rho is used to quantify correlations robustly based on ranks. 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 (ranked) expression within each gene.

The correlateNull function constructs an empirical null distribution for rho computed with ncells cells. When design=NULL, this is done by shuffling the ranks, calculating the rho and repeating until iters values are obtained. 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.

For `correlatePairs`, a pre-computed empirical distribution can be supplied as `null.dist` if available. Otherwise, it will be automatically constructed via `correlateNull` with `ncells` set to the number of columns in `x`. For `correlatePairs,SingleCellExperiment-method`, correlations should be computed for normalized expression values in the specified `assay.type`.

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%.

If `per.gene=TRUE`, results are summarized on a per-gene basis. 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. Setting `per.gene=TRUE` is useful for identifying correlated genes without regard to what they are correlated with (e.g., during feature selection).

Value

For `correlateNull`, a numeric vector of length `iters` is returned containing the sorted correlations under the null hypothesis of no correlations. Arguments to `design` and `residuals` are stored in the attributes.

For `correlatePairs` with `per.gene=FALSE`, 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`.

For `correlatePairs` with `per.gene=TRUE`, a dataframe is returned with one row per gene. For each row, the `rho` field contains the correlation with the largest magnitude across all gene pairs involving the corresponding gene. The `p.value` field contains the Simes p-value, and the `FDR` field contains the corresponding adjusted p-value.

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 (and it will take priority if both `block` and `design` are specified). While `design` can also be used for one-way layouts, this is not ideal as it involves more work/assumptions:

- It assumes normality, during both linear modelling and generation of the null distribution. This assumption is largely unavoidable for complex designs, where some quantitative constraints are required to remove nuisance effects. `x` should generally be log-transformed here, whereas this is not required for (but does not hurt) the first group-based approach.
- Residuals computed from expression values equal to `lower.bound` are set to a constant value below all other residuals. This preserves ties between zeroes and avoids spurious correlations between genes due to arbitrary tie-breaking. The value of `lower.bound` should be equal to log-prior count used during the log-transformation. It is automatically taken from `metadata(x)$log.exprs.offset` if `x` is a SingleCellExperiment object.

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. This will iteract properly with `pairings`, such that genes in `pairings` and not in `subset.row` will be ignored. With `correlatePairs,SingleCellExperiment-method`, 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` to contain only the subset of genes of interest. This reduces computational time by only testing correlations 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.

Lowly-expressed genes can also cause problems when `design` is non-`NULL` and `residuals=TRUE`. Tied counts, and zeroes in particular, may not result in tied residuals after fitting of the linear model. Model fitting may break ties in a consistent manner across genes, yielding large correlations between genes with many zero counts. Focusing on HVGs should mitigate the detection of these uninteresting correlations, as genes dominated by zeroes will usually have low variance.

Approximating Spearman's rho with tied values

As previously mentioned, an approximate version of Spearman's rho is used. Specifically, untied ranks are randomly assigned to any tied values. This means that a common empirical distribution can be used for all gene pairs, rather than having to do new permutations for every pair to account for the different pattern of ties. Generally, this modification has little effect on the results for expressed genes (and in any case, differences in library size break ties for normalized expression values). Some correlations may end up being spuriously large, but this should be handled by the error control machinery after multiplicity correction.

Author(s)

Aaron Lun

References

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.

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

See Also

bpparam, cor

Examples

```
set.seed(0)
ncells <- 100
null.dist <- correlateNull(ncells, iters=100000)
exprs <- matrix(rpois(ncells*100, lambda=10), ncol=ncells)
out <- correlatePairs(exprs, null.dist=null.dist)
hist(out$p.value)
```

---

cyclone *Cell cycle phase classification*

Description

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

Usage

```
## 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.
pairs	A list of data.frames produced by sandbag, containing pairs of marker genes.
gene.names	A character vector of gene names.
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 bplapply for parallel processing.
verbose	A logical scalar specifying whether diagnostics should be printed to screen.
subset.row	A logical, integer or character scalar indicating the rows of x to use.
...	Additional arguments to pass to cyclone,ANY-method.
assay.type	A string specifying which assay values to use, e.g., "counts" or "logcounts".
get.spikes	A logical specifying whether spike-in transcripts should be used.

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.

For cyclone,SingleCellExperiment-method, the matrix of counts is used but can be replaced with expression values by setting assay.type. By default, get.spikes=FALSE which means that any rows corresponding to spike-in transcripts will not be considered for score calculation. This is for the same reasons as described in ?sandbag.

Users can also manually set subset.row to specify which rows of x are to be used. This is better than subsetting x directly, as it reduces memory usage and also subsets gene.names at the same time. If this is specified, it will overwrite any setting of get.spikes.

While this method is described for cell cycle phase classification, any biological groupings can be used here – see ?sandbag for details. However, for non-cell cycle phase groupings, the output phases will be an empty character vector. Users should manually apply their own score thresholds for assigning cells into specific groups.

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

Examples

```
example(sandbag) # Using the mocked-up data in this example.

# Classifying (note: test.data!=training.data in real cases)
test <- training
assignments <- cyclone(test, out)
head(assignments$scores)
head(assignments$phases)

# Visualizing
col <- character(ncells)
col[is.G1] <- "red"
col[is.G2M] <- "blue"
```

```
col[is.S] <- "darkgreen"
plot(assignments$score$G1, assignments$score$G2M, col=col, pch=16)
```

`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, ...)

## 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 the output of `trendVar`, run 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.
`subset.row`	A logical, integer or character scalar indicating the rows of `x` to use.
`...`	For `decomposeVar,matrix,list-method`, additional arguments to pass to `testVar`. For `decomposeVar,SingleCellExperiment,list-method`, additional arguments to pass to the matrix method.
`assay.type`	A string specifying which assay values to use from `x`.
`get.spikes`	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.

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` (if `subset.row=NULL`; otherwise, each row corresponds to an element of `subset.row`). 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.

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`.

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 `combineVar` instead, see the discussion in `?trendVar` for more details.

Feature selection

If `get.spikes=NA` in `decomposeVar,SingleCellExperiment-method`, the p-value and FDR will not be returned 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. If `get.spikes=FALSE`, spike-in transcripts are filtered out from `x` and no statistics are returned. If `get.spikes=TRUE`, all statistics are computed for all rows, regardless of spike-in status.

Users can also directly specify which rows to use with `subset.row`. This is equivalent to running `decomposeVar` on `x[subset.row,]`, but is more efficient as it avoids the construction of large temporary matrices.

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

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)
```

`Deconvolution Methods` *Normalization by deconvolution*

Description

Methods to normalize single-cell RNA-seq data by deconvolving size factors from cell pools.

Usage

```
## S4 method for signature 'ANY'
computeSumFactors(x, sizes=seq(20, 100, 5), clusters=NULL, ref.clust=NULL,
    max.cluster.size=3000, positive=FALSE, errors=FALSE, min.mean=1,
    subset.row=NULL)

## S4 method for signature 'SingleCellExperiment'
computeSumFactors(x, ..., min.mean=1, subset.row=NULL,
    assay.type="counts", get.spikes=FALSE, sf.out=FALSE)
```

Arguments

`x`	A numeric matrix-like object of counts, where rows are genes and columns are cells. Alternatively, a SingleCellExperiment object containing such a matrix.
`sizes`	A numeric vector of pool sizes, i.e., number of cells per pool.
`clusters`	An optional factor specifying which cells belong to which cluster, for deconvolution within clusters.
`ref.clust`	A level of `clusters` to be used as the reference cluster for inter-cluster normalization.
`max.cluster.size`	An integer scalar specifying the maximum number of cells in each cluster.
`positive`	A logical scalar indicating whether linear inverse models should be used to enforce positive estimates.
`errors`	A logical scalar indicating whether the standard error should be returned. This option is deprecated, see below.
`min.mean`	A numeric scalar specifying the minimum (library size-adjusted) average count of genes to be used for normalization.
`subset.row`	A logical, integer or character scalar indicating the rows of `x` to use.
`...`	Additional arguments to pass to `computeSumFactors,ANY-method`.
`assay.type`	A string specifying which assay values to use, e.g., `"counts"` or `"logcounts"`.
`get.spikes`	A logical scalar specifying whether spike-in transcripts should be used.
`sf.out`	A logical scalar indicating whether only size factors should be returned.

Value

For `computeSumFactors,ANY-method`, a numeric vector of size factors for all cells in `x` is returned.

For `computeSumFactors,SingleCellExperiment-method`, an object of class `x` is returned containing the vector of size factors in `sizeFactors(x)`, if `sf.out=FALSE`. Otherwise, the vector of size factors is returned directly.

Overview of the deconvolution method

The `computeSumFactors` function provides an implementation of the deconvolution strategy for normalization. Briefly, a pool of cells is selected and the counts for those cells are summed together. The count sums for this pool 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.

Now, the bias for the pool is equal to the sum of the biases for the constituent cells. The same applies for the size factors (which 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 will feed back into the deconvolution process, thus improving the accuracy of the cell-based size factors.

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 across the data set. 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. Each cluster should contain a sufficient number of cells for pooling – an error is thrown if the number of cells is less than the maximum value of `sizes`. A convenince function `quickCluster` is provided for rapid clustering based on Spearman's rank correlation.

Size factors computed within each cluster must be rescaled for comparison between clusters. This is done by normalizing between clusters to identify the rescaling factor. One cluster is chosen as a "reference" (by default, that with the median of the mean per-cell library sizes is used) to which all others are normalized. Ideally, a cluster that is not extremely different from all other clusters should be used as the reference. This can be specified using `ref.clust` if there is prior knowledge about which cluster is most suitable, e.g., from PCA or t-SNE plots.

Additional details about pooling and deconvolution

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 20 to 100. Using a range of window sizes improves the precision of the estimates, at the cost of increased computational complexity. The defaults were chosen to provide a reasonable compromise between these two considerations. 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 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, 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.

Dealing with negative size factors

In theory, it is possible to obtain 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 counts are zero 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.

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, some protection can be provided by setting `positive=TRUE`, which will use linear inverse models to solve the system. This ensures that non-negative values for the size factors will always be obtained. Note that some cells may still have size factors of zero and should be removed prior to downstream analysis. Such occurrences are unavoidable – rather, the aim is to prevent negative values from affecting the estimates for all other cells.

Gene selection

By default, `get.spikes=FALSE` in `quickCluster,SingleCellExperiment-method` which means that spike-in transcripts are not included in the set of genes used for deconvolution. This is because they can behave differently from the endogenous genes. Users wanting to perform spike-in normalization should see `computeSpikeFactors` instead.

Note that pooling does not eliminate the need to filter out low-abundance genes. As mentioned above, 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. Filtering ensures that this is not the case, e.g., by removing genes with average counts below 1.

In general, genes with average counts below 1 (for read count data) or 0.1 (for UMI data) should not be used for normalization. Such genes will automatically be filtered out by applying a minimum threshold `min.mean` on the library size-adjusted average counts from `calcAverage`. If `clusters` is specified, filtering by `min.mean` is performed on the per-cluster average during within-cluster normalization, and then on the average of the per-cluster averages during between-cluster normalization.

Users can also set `subset.row` to specify which rows of `x` are to be used to calculate the normalization factors. This is equivalent to but more efficient than subsetting `x` directly, as it avoids constructing a (potentially large) temporary matrix. If `subset.row` is specified and `get.spikes=FALSE`, only the non-spike-in entries of `subset.row` will be used in deconvolution. Similarly, only the genes selected by `subset.row` and with average counts above `min.mean` will be used.

Obtaining standard errors

Previous versions of computeSumFactors would return the standard error for each size factor when errors=TRUE. This is no longer the case, as 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

Examples

```
# Mocking up some data.
set.seed(100)
popsize <- 200
ngenes <- 10000
all.facs <- 2^rnorm(popsize, sd=0.5)
counts <- matrix(rnbinom(ngenes*popsize, mu=all.facs*10, size=1), ncol=popsize, byrow=TRUE)

# Computing the size factors.
out.facs <- computeSumFactors(counts)
head(out.facs)
plot(colSums(counts), out.facs, log="xy")
```

Denoise with PCA *Denoise expression with PCA*

Description

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

Usage

```
## S4 method for signature 'ANY'
denoisePCA(x, technical, design=NULL, subset.row=NULL,
    value=c("pca", "n", "lowrank"), min.rank=5, max.rank=100,
    approximate=FALSE, rand.seed=1000, irlba.args=list())

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

Arguments

x	A numeric matrix of log-expression values for denoisePCA,ANY-method, or a SingleCellExperiment object containing such values for denoisePCA,SingleCellExperiment-meth
technical	A function that computes the technical component of the variance for a gene with a given mean (log-)expression, see ?trendVar. This can also be a numeric vector containing the technical component for each gene in x; or the entire DataFrame generated by decomposeVar or combineVar.
design	A numeric matrix containing the experimental design. This is a deprecated option, see the details below.
subset.row	A logical, integer or character vector indicating the rows of x to use for PCA. All genes are used by default in denoisePCA,ANY-method. Only non-spike-in transcripts are used by default when x is a SingleCellExperiment.
value	A string specifying the type of value to return; the PCs, the number of retained components, or a low-rank approximation.
min.rank, max.rank	Integer scalars specifying the minimum and maximum number of PCs to retain.
approximate	A logical scalar indicating whether approximate SVD should be performed via irlba.
rand.seed	A numeric scalar specifying the seed for approximate PCA when approximate=TRUE. This can be set to NA to use the existing session seed.
irlba.args	A named list of additional arguments to pass to irlba when approximate=TRUE.
...	Further arguments to pass to denoisePCA,ANY-method.
assay.type	A string specifying which assay values to use.
get.spikes	A logical scalar specifying whether spike-in transcripts should be used. This will be intersected with subset.row if the latter is specified.
sce.out	A logical scalar specifying whether a modified SingleCellExperiment object should be returned.

Details

This function performs a principal components analysis to reduce 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 substructure should be correlated and captured by earlier PCs, as this explains more variance for sets of genes. The idea is to discard later PCs to remove technical noise and improve the resolution of substructure.

The choice of the number of PCs to discard is based on the estimates of technical variance in technical. This can be a vector of technical components for each gene, as produced by decomposeVar. Alternatively, the trend function obtained from trendVar is used to compute the technical component for each gene, based on its mean abundance. An estimate of the overall technical variance is estimated by summing the values across genes. Genes with negative biological components are ignored during downstream analyses to ensure that the total variance is greater than the overall technical estimate.

The function works by assuming that the first X PCs contain all of the biological signal, while the remainder contains technical noise. For a given value of X, an estimate of the total technical variance is calculated from the sum of variance explained by all of the later PCs. A value of X is found such that the predicted technical variance equals the estimated technical variance. Note that X will be coerced to lie between min.rank and max.rank.

Value

For denoisePCA,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 denoisePCA,SingleCellExperiment-method, the return value is the same as denoisePCA,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. This is directly compatible with functions such as plotPCA.

Generating low-rank approximations

When value="lowrank", a low-rank approximation of the original matrix is computed using only the first X components. This is useful for denoising prior to downstream applications that expect gene-wise expression profiles.

Note that approximation values are returned for *all* genes. This includes "unselected" genes, i.e., with negative biological components or that were not selected with subset.row. The low-rank approximation is obtained for these genes by projecting their expression profiles into the low-dimensional space defined by the SVD on the selected genes. The exception is when get.spikes=FALSE, whereby zeroes are returned for all spike-in rows.

Handling uninteresting factors of variation

Previous versions of this function allowed users to specify a design matrix to regress out uninteresting factors of variation. This behaviour is now deprecated in favour of users running appropriate batch correction functions beforehand. If a batch-corrected expression matrix is supplied in x, users should also supply a DataFrame as technical, as calculation of the residual variance or mean from a corrected matrix is not accurate without knowledge of the blocking factors.

Any correction should preserve the residual variance of each gene for the variances to be correctly interpreted. Specifically, calculation of the variance within each batch should yield the same result in both the corrected and original matrices. This limits the possible methods for batch correction:

- removeBatchEffect performs a linear regression for each gene, removing unwanted factors while retaining relevant biological effects (if known). This simply involves fitting a linear model and removing undesired coefficients, so the residual variance is unaffected.
- mnnCorrect will identify cells of the same biological identity across batches, and use them to compute correction vectors. This usually requires setting cos.norm.out=FALSE and sigma to some large value to preserve the variance.

Author(s)

Aaron Lun

See Also

trendVar, decomposeVar

Examples

```
# Mocking up some data.
ngenes <- 1000
is.spike <- 1:100
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)
rownames(counts) <- paste0("Gene", seq_len(ngenes))

# Fitting a trend.
lcounts <- log2(counts + 1)
fit <- trendVar(lcounts, subset.row=is.spike)

# Denoising (not including the spike-ins in the PCA;
# spike-ins are automatically removed with the SingleCellExperiment method).
pcs <- denoisePCA(lcounts, technical=fit$trend, subset.row=-is.spike)
dim(pcs)
```

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)
```

Explore Data *Shiny app for explorative data analysis*

Description

Generate an interactive Shiny app to explore gene expression patterns in single-cell data

Usage

```
exploreData(x, cell.data, gene.data, red.dim, run=TRUE)
```

Arguments

`x`	A numeric matrix of expression values to be visualized.
`cell.data`	A data frame of cell information, where each row corresponds to a column of `x`.
`gene.data`	A data frame of gene information, where each row corresponds to a row of `x`.
`red.dim`	A numeric matrix with two colums, specifying the reduced-dimension coordinates for each cell.
`run`	A logical scalar specifying whether the app should be run immediately.

Details

Note that this function is deprecated; we suggest using the **iSEE** package for data exploration instead.

This function will return a Shiny app object that can be run with `runApp`. The app allows the user to interactively explore gene expression patterns in single-cell RNA-seq data. Explorative analysis is focused on comparing gene exression between different groups of cells, as defined by the covariates of `cell.data`.

Three plots are shown in the app:

- a scatterplot of cell locations based on the `red.dim` coordinates, colored by a selected covariate

- a scatterplot of cell locations based on the `red.dim` coordinates, colored by expression of a selected gene
- boxplot(s) of expression values for a selected gene, grouped by a selected covariate.

Several options are available within the app:

"Color by": Covariate to be used for coloring the first scatter plot.

"Group by": Covariate with which expression values are grouped in the boxplots.

In addition, the `gene.data` data frame is rendered as an interactive table using the JavaScript library `DataTable`. This allows the user to subset/search the feature data and select a gene by clicking on the corresponding row.

Value

If `run=FALSE`, a Shiny app object is returned, which can be run with `runApp`. If `run=TRUE`, a Shiny app object is created and run.

Author(s)

Karsten Bach

See Also

`runApp`,

Examples

```
# Set up example data
example(SingleCellExperiment)
cell.data <- DataFrame(stuff=sample(LETTERS, ncol(sce), replace=TRUE))
gene.data <- DataFrame(lengths=runif(nrow(sce)))

# Mocking up  some reduced dimensions.
pca <- prcomp(t(exprs(sce)))
red.dim <- pca$x[,1:2]

# Creating the app object.
app <- exploreData(exprs(sce), cell.data, gene.data, red.dim, run=FALSE)
if (interactive()) { shiny::runApp(app) }

## Not run: # Running directly from the function.
saved <- exploreData(x, cell.data, gene.data, red.dim)

## End(Not run)
```

`findMarkers` *Find marker genes*

Description

Find candidate marker genes for clusters of cells, by testing for differential expression between clusters.

Usage

```
## S4 method for signature 'ANY'
findMarkers(x, clusters, block=NULL, design=NULL,
    pval.type=c("any", "all"), direction=c("any", "up", "down"),
    lfc=0, log.p=FALSE, full.stats=FALSE, subset.row=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 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.
`clusters`	A vector of cluster identities for all cells.
`block`	A factor specifying the blocking level for each cell.
`design`	A numeric matrix containing blocking terms, i.e., uninteresting factors driving expression across cells.
`pval.type`	A string specifying the type of combined p-value to be computed, i.e., Simes' or IUT.
`direction`	A string specifying the direction of log-fold changes to be considered for each cluster.
`lfc`	A positive numeric scalar specifying the log-fold change threshold to be tested against.
`log.p`	A logical scalar indicating if log-transformed p-values/FDRs should be returned.
`full.stats`	A logical scalar indicating whether all statistics (i.e., raw and BH-adjusted p-values) should be returned for each pairwise comparison.
`subset.row`	A logical, integer or character scalar indicating the rows of `x` to use.
`...`	Additional arguments to pass to the matrix method.
`assay.type`	A string specifying which assay values to use, e.g., `"counts"` or `"logcounts"`.
`get.spikes`	A logical scalar specifying whether decomposition should be performed for spike-ins.

Details

This function performs t-tests to identify differentially expressed genes (DEGs) between pairs of clusters. For each cluster, the log-fold changes and other statistics from all relevant pairwise comparisons are combined into a single table. A list of such tables is returned for all clusters to define a set of potential marker genes.

Users can specify the genes to check for differential expression (DE) by setting the `subset.row` argument. In addition, spike-in transcripts are ignored in the SingleCellExperiment method when `get.spikes=FALSE`. If this is set, it will intersect with any non-`NULL` value of `subset.row`, i.e., only non-spike-in transcripts in the specified set will be used.

Value

A named list of DataFrames. Each DataFrame corresponds to a cluster in `clusters`, where rows correspond to genes and are ranked by importance. Within the DataFrame for each cluster (e.g., cluster X), there are the following fields:

`Top`: Integer, the minimum rank across all pairwise comparisons if `rank.type="any"`.

`IUT.p`: Numeric, the IUT p-value across all comparisons if `rank.type="all"`. This is log-transformed and reported as `log.IUT.p` if `log.p=TRUE`.

`FDR`: Numeric, the BH-adjusted p-value for each gene. This is log-transformed and reported as `log.FDR` if `log.p=TRUE`.

`logFC.Y`: Numeric for every other cluster Y in `clusters`, containing the log-fold change of X over Y when `full.stats=FALSE`.

`stats.Y`: DataFrame for every other cluster Y in `clusters`, returned when `full.stats=TRUE`. This contains the numeric fields `logFC`, the log-fold change of X over Y; `p.value` or `log.p.value`, the (log-transformed) p-value for the pairwise comparison between X and Y; and `FDR` or `log.FDR`, the (log-transformed) BH-adjusted p-value. Setting `log.p=TRUE` will yield the log-transformed output.

Genes are ranked by the `Top` or `IUT.p` column, depending on `rank.type`.

Explanation of the hypothesis tests

By default, this function will perform a Welch t-test to identify DEGs between each pair of clusters. This is simple, fast and performs quite well for single-cell count data (Soneson and Robinson, 2018). However, if one of the clusters contains fewer than two cells, no p-value will be reported for this comparison.

If `block` is specified, the same t-tests are performed between clusters within each level of `block`. For each pair of clusters, the p-values for each gene across all levels of `block` are combined using Stouffer's Z-score method. The p-value for each level is assigned a weight inversely proportional to the expected variance of the log-fold change estimate for that level. Blocking levels are ignored if no p-value was reported, e.g., if there were insufficient cells for a cluster in a particular level.

If `design` is specified, a linear model is instead fitted to the expression profile for each gene. This linear model will include the `clusters` as well as any blocking factors in `design`. A t-test is then performed to identify DEGs between pairs of clusters, using the values of the relevant coefficients and the gene-wise residual variance.

Note that `block` will override any `design` if both are specified. This reflects our preference for the former, which accommodates differences in the variance of expression in each cluster via Welch's t-test. As a result, it is more robust to misspecification of the clusters, as misspecified clusters (and inflated variances) do not affect the inferences for other clusters. Use of `block` also avoids assuming additivity of effects between the blocking factors and the cluster identities.

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 clusters (assuming that design is not confounded with the cluster identities). This may not be the case with block if a pair of clusters never co-occur in a single blocking level.

Direction and magnitude of the log-fold change

If direction="any", two-sided tests will be performed for each pairwise comparisons between clusters. Otherwise, one-sided tests in the specified direction will be used to compute p-values for each gene. This can be used to focus on genes that are upregulated in each cluster of interest, which is often easier to interpret.

To interpret the setting of direction, consider the DataFrame for cluster X, in which we are comparing to another cluster Y. If direction="up", genes will only be significant in this DataFrame if they are upregulated in cluster X compared to Y. If direction="down", genes will only be significant if they are downregulated in cluster 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.

Consolidating p-values into a ranking

By default, each table is sorted by the Top value when pval.type="any". This is the minimum rank across all pairwise comparisons for each gene, and specifies the size of the candidate marker set. Taking all rows with Top values no greater than some integer X will yield a set containing the top X genes (ranked by significance) from each pairwise comparison. For example, if X is 5, the set will consist of the *union* of the top 5 genes from each pairwise comparison. The marker set for each cluster allows it to be distinguished from each other cluster based on the expression of at least one gene.

This approach does not explicitly favour genes that are uniquely expressed in a cluster. Such a strategy is often too stringent, especially in cases involving overclustering or cell types defined by combinatorial gene expression. However, 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. This yields a IUT.p field instead of a Top field in the output table. Ranking based on the IUT p-value will focus on genes that are uniquely DE in that cluster.

Correcting for multiple testing

When pval.type="any", a combined p-value is calculated by consolidating p-values across contrasts for each gene using Simes' method. This represents the evidence against the null hypothesis is that the gene is not DE in any of the contrasts. The BH method is then applied on the combined p-values across all genes to obtain the FDR field. The same procedure is done with pval.type="all", but using the IUT p-values across genes instead.

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

Note that 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.

Weighting across blocking levels

When `block` is specified, the weight for the p-value in a particular level is defined as $(1/Nx + 1/Ny)^{-1}$, where Nx and Ny are the number of cells in clusters X and Y, respectively, for that level. This is inversely proportional to the expected variance of the log-fold change, provided that all clusters and blocking levels have the same variance.

In theory, a better weighting scheme would be to use the estimated standard error of the log-fold change to compute the weight. This would be more responsive to differences in variance between blocking levels, focusing on levels with low variance and high power. However, this is not safe in practice as genes with many zeroes can have very low standard errors, dominating the results inappropriately.

Like the p-values, the reported log-fold change for each gene is a weighted average of log-fold changes from all levels of the blocking factor. The weight for each log-fold change is inversely proportional to the expected variance of the log-fold change in that level. Unlike p-values, though, this calculation will use blocking levels where both clusters contain only one cell.

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.

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

`normalize`

Examples

```
# Using the mocked-up data 'y2' from this example.
example(computeSpikeFactors)
y2 <- normalize(y2)
kout <- kmeans(t(logcounts(y2)), centers=2) # Any clustering method is okay.
out <- findMarkers(y2, clusters=kout$cluster)
```

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 is intended to be a more stable version of technicalCV2.

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 SingleCellExperiment 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 log.prior!=NULL.
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 log.prior!=NULL.
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 smooth.spline.
robust	A logical scalar indicating whether robust fitting should be performed with robustSmoothSpline.
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 improvedCV2,ANY-method.
assay.type	A string specifying which assay values to use.
logged	A logical scalar indicating if assay.type contains log-expression values. This is automatically determined if assay.type="counts", "logcounts" or "normcounts".
normalized	A logical scalar indicating if assay.type 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.

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,SingleCellExp`
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 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

See Also

ns, technicalCV2

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)

# 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)
X <- normalize(X)

# Running it.
out <- improvedCV2(X, spike.type="Spikes")
head(out)
```

---

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, x=NULL)
```

Arguments

means	A numeric vector of average counts. Note that there are means of the counts, *not* means of the log-expression values.
size.factors	A numeric vector of size factors. If supplied, these should be centred at unity.
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, this is equivalent to a Poisson distribution.
pseudo.count	A numeric scalar specifying the pseudo-count to be added to the scaled counts before log-transformation.
x	A SingleCellExperiment object from which size.factors and pseudo.count are extracted, and means can be automatically inferred.

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 specified); and means is automatically determined from the range of row averages of logcounts(sce) (after undoing the log-transformation).

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))
```

mnnCorrect *Mutual nearest neighbors correction*

Description

Correct for batch effects in single-cell expression data using the mutual nearest neighbors method.

Usage

```
mnnCorrect(..., k=20, sigma=0.1, cos.norm.in=TRUE, cos.norm.out=TRUE,
    svd.dim=0L, var.adj=TRUE, compute.angle=FALSE, subset.row=NULL,
    order=NULL, pc.approx=FALSE, irlba.args=list(),
    BPPARAM=SerialParam())
```

Arguments

...	Two or more expression matrices where genes correspond to rows and cells correspond to columns. Each matrix should contain cells from the same batch; multiple matrices represent separate batches of cells. Each matrix should contain the same number of rows, corresponding to the same genes (in the same order).
k	An integer scalar specifying the number of nearest neighbors to consider when identifying mutual nearest neighbors.
sigma	A numeric scalar specifying the bandwidth of the Gaussian smoothing kernel used to compute the correction vector for each cell.
cos.norm.in	A logical scalar indicating whether cosine normalization should be performed on the input data prior to calculating distances between cells.
cos.norm.out	A logical scalar indicating whether cosine normalization should be performed prior to computing corrected expression values.
svd.dim	An integer scalar specifying the number of dimensions to use for summarizing biological substructure within each batch.
var.adj	A logical scalar indicating whether variance adjustment should be performed on the correction vectors.
compute.angle	A logical scalar specifying whether to calculate the angle between each cell's correction vector and the biological subspace of the reference batch.
subset.row	A vector specifying the genes with which distances between cells are calculated, e.g., for identifying mutual nearest neighbours. All genes are used by default.
order	An integer vector specifying the order in which batches are to be corrected.
pc.approx	A logical scalar indicating whether irlba should be used to identify the biological subspace.
irlba.args	A list of arguments to pass to irlba when pc.approx=TRUE.
BPPARAM	A BiocParallelParam object specifying whether the nearest-neighbor searches should be parallelized.

Details

This function is designed for batch correction of single-cell RNA-seq data where the batches are partially confounded with biological conditions of interest. It does so by identifying pairs of mutual nearest neighbors (MNN) in the high-dimensional expression space. Each MNN pair represents cells in different batches that are of the same cell type/state, assuming that batch effects are mostly orthogonal to the biological manifold. Correction vectors are calculated from the pairs of MNNs and corrected expression values are returned for use in clustering and dimensionality reduction.

The concept of a MNN pair can be explained by considering cells in each of two batches. For each cell in one batch, the set of `k` nearest cells in the other batch is identified, based on the Euclidean distance in expression space. Two cells in different batches are considered to be MNNs if each cell is in the other's set. The size of `k` can be interpreted as the minimum size of a subpopulation in each batch. The algorithm is generally robust to the choice of `k`, though values that are too small will not yield enough MNN pairs, while values that are too large will ignore substructure within each batch.

For each MNN pair, a pairwise correction vector is computed based on the difference in the expression profiles. The correction vector for each cell is computed by applying a Gaussian smoothing kernel with bandwidth `sigma` is the pairwise vectors. This stabilizes the vectors across many MNN pairs and extends the correction to those cells that do not have MNNs. The choice of `sigma` determines the extent of smoothing - a value of 0.1 is used by default, corresponding to 10% of the radius of the space after cosine normalization.

Value

A named list containing two components:

`corrected`**:** A list of length equal to the number of batches, containing matrices of corrected expression values for each cell in each batch. The order of batches is the same as supplied in `...`, and the order of cells in each matrix is also unchanged.

`pairs`**:** A named list of length equal to the number of batches, containing DataFrames specifying the MNN pairs used for correction. Each row of the DataFrame defines a pair based on the cell in the current batch and another cell in an earlier batch. The identity of the other cell and batch are stored as run-length encodings to save space.

`angles`**:** A named list of length equal to the number of batches, containing numeric vectors of angles. Each angle is computed between each cell's correction vector with the first two basis vectors of the first batch of cells (plus any previously corrected batches). This is only returned if `compute.angle=TRUE`.

Choosing the gene set

Distances between cells are calculated with all genes if `subset.row=NULL`. However, users can set `subset.row` to perform the distance calculation on a subset of genes, e.g., highly variable genes or marker genes. This may provide more meaningful identification of MNN pairs by reducing the noise from irrelevant genes.

Regardless of whether `subset.row` is specified, corrected values are returned for *all* genes. This is possible as `subset.row` is only used to identify the MNN pairs and other cell-based distance calculations. Correction vectors between MNN pairs can then be computed in the original space involving all genes in the supplied matrices.

Expected type of input data

The input expression values should generally be log-transformed, e.g., log-counts, see `normalize` for details. They should also be normalized within each data set to remove cell-specific biases in

capture efficiency and sequencing depth. By default, a further cosine normalization step is performed on the supplied expression data to eliminate gross scaling differences between data sets.

- When `cos.norm.in=TRUE`, cosine normalization is performed on the matrix of expression values used to compute distances between cells. This can be turned off when there are no scaling differences between data sets.
- When `cos.norm.out=TRUE`, cosine normalization is performed on the matrix of values used to calculate correction vectors (and on which those vectors are applied). This can be turned off to obtain corrected values on the log-scale, similar to the input data.

Users should note that the order in which batches are corrected will affect the final results. The first batch in `order` is used as the reference batch against which the second batch is corrected. Corrected values of the second batch are added to the reference batch, against which the third batch is corrected, and so on. This strategy maximizes the chance of detecting sufficient MNN pairs for stable calculation of correction vectors in subsequent batches.

Further options

The function depends on a shared biological manifold, i.e., one or more cell types/states being present in multiple batches. If this is not true, MNNs may be incorrectly identified, resulting in over-correction and removal of interesting biology. Some protection can be provided by removing components of the correction vectors that are parallel to the biological subspaces in each batch. The biological subspace in each batch is identified with a SVD on the expression matrix, using either `svd` or `irlba`. The number of dimensions of this subspace can be controlled with `svd.dim`. (By default, this option is turned off by setting `svd.dim=0`.)

If `var.adj=TRUE`, the function will adjust the correction vector to equalize the variances of the two data sets along the batch effect vector. In particular, it avoids "kissing" effects whereby MNN pairs are identified between the surfaces of point clouds from different batches. Naive correction would then bring only the surfaces into contact, rather than fully merging the clouds together. The adjustment ensures that the cells from the two batches are properly intermingled after correction. This is done by identifying each cell's position on the correction vector, identifying corresponding quantiles between batches, and scaling the correction vector to ensure that the quantiles are matched after correction.

Author(s)

Laleh Haghverdi, with modifications by Aaron Lun

See Also

`get.knnx` `irlba`

Examples

```
B1 <- matrix(rnorm(10000), ncol=50) # Batch 1
B2 <- matrix(rnorm(10000), ncol=50) # Batch 2
out <- mnnCorrect(B1, B2) # corrected values
```

multiBlockVar *Per-block variance statistics*

Description

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

Usage

```
multiBlockVar(x, block, trend.args=list(), dec.args=list(), ...)
```

Arguments

x	A SingleCellExperiment object containing log-normalized expression values, computed from size factors centred in each block - see normalize for details.
block	A factor specifying the blocking level for each cell.
trend.args	A list of named arguments to pass to trendVar.
dec.args	A list of named arguments to pass to decomposeVar.
...	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 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 are centred *within* each block when computing log-normalized expression values. This preserves the scale of the counts within each block, and ensures that the spike-in normalized values are comparable to those of the endogenous genes. Centring can be performed by setting the size_factor_grouping argument in normalize. Otherwise, a warning will be raised about non-centred 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 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, 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)

# Setting up the blocking levels.
block <- sample(3, ncol(y), replace=TRUE)
y <- normalize(y, size_factor_grouping=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 *Overlap expression profiles*

Description

Compute the gene-specific overlap in expression profiles between two groups of cells.

Usage

```
## S4 method for signature 'ANY'
overlapExprs(x, groups, block=NULL, design=NULL,
    rank.type=c("any", "all"), direction=c("any", "up", "down"),
    tol=1e-8, BPPARAM=SerialParam(), subset.row=NULL,
    lower.bound=NULL, residuals=FALSE)

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

Arguments

x	A numeric matrix of 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.
groups	A vector of group assignments for all cells.
block	A factor specifying the blocking level for each cell.

design	A numeric matrix containing blocking terms, i.e., uninteresting factors driving expression across cells.
rank.type	A string specifying which comparisons should be used to rank genes in the output.
direction	A string specifying which direction of change in expression should be used to rank genes in the output.
tol	A numeric scalar specifying the tolerance with which ties are considered.
BPPARAM	A BiocParallelParam object to use in bplapply for parallel processing.
subset.row	A logical, integer or character scalar indicating the rows of x to use.
lower.bound	A numeric scalar specifying the theoretical lower bound of values in x, only used when residuals=TRUE.
residuals	A logical scalar indicating whether overlaps should be computed between residuals of a linear model.
...	Additional arguments to pass to the matrix method.
assay.type	A string specifying which assay values to use, e.g., "counts" or "logcounts".
get.spikes	A logical scalar specifying whether decomposition should be performed for spike-ins.

Details

For two groups of cells A and B, consider the distribution of expression values for gene X across those cells. The overlap proportion is defined as the probability that a randomly selected cell in A has a greater expression value of X than a randomly selected cell in B. Overlap proportions near 0 or 1 indicate that the expression distributions are well-separated. In particular, large proportions indicate that most cells of the first group (A) express the gene more highly than most cells of the second group (B).

This function computes, for each gene, the overlap proportions between all pairs of groups in groups. It will then rank the genes based on how well they differentiate between groups. overlapExprs is designed to complement findMarkers, which reports the log-fold changes between groups. This is useful for prioritizing candidate markers without needing to plot their expression values.

Expression values that are tied between groups are considered to be 50% likely to be greater in either group. Thus, if all values were tied, the overlap proportion would be equal to 0.5. The tolerance with which ties are considered can be set by changing tol.

Users can specify which subset of genes to perform these calculations on, by supplying a non-NULL value of subset.row. By default, spike-in transcripts are ignored in overlapExprs,SingleCellExperiment-method with get.spikes=FALSE. If get.spikes=FALSE and subset.row!=NULL, the function will only use the non-spike-in transcripts in subset.row.

Value

A named list of DataFrames. Each DataFrame corresponds to a group in groups and contains one row per gene in x (or the subset specified by subset.row). Within the DataFrame for each group (e.g., group A), there are the following fields:

Top: Integer, the minimum rank across all pairwise comparisons if rank.type="any".

Worst: Numeric, the value of the overlap proportion corresponding to the smallest separation statistic across all comparisons if rank.type="all".

overlap.B: Numeric for every other group B in groups, containing overlap proportions between groups A and B for that gene.

Genes are ranked by the Top or Best column, depending on rank.type.

Ranking genes in the output

Each overlap proportion is first converted into a separation statistic. The definition of the seperation statistic depends on the specified `direction`:

- If `direction="any"` (the default), the separation statistic is defined as the absolute difference of the overlap proportion from zero or 1 (whichever is closer). Thus, if the overlap between the expression distributions for A and B is poor, the separation statistic will be large.
- If `direction="up"`, the separation statistic is defined as the difference of the overlap proportion from zero. Thus, the separation statistic will only be large when the distribution of A is shifted upwards compared to B.
- If `direction="down"`, the separation statistic is defined as the difference of the overlap proportion from 1. Thus, the separation statistic will only be large when the distribution of A is shifted downwards compared to B.

If `rank.type="any"`, the genes in each group-specific DataFrame are ranked using a similar logic to that in `findMarkers`. This involves calculation of a `Top` value for each gene, representing the minimum ranking of the separation statistics across pairwise comparisons. To illustrate, consider the DataFrame for group A, and take the set of all genes with `Top` values less than or equal to some integer X. This set is the union of the top X genes with the largest separation statistics from each pairwise comparison between group A and every other group. Ranking genes based on the `Top` value prioritizes genes that exhibit low overlaps between group A and *any* other group.

If `rank.type="all"`, the genes in each group-specific DataFrame are ranked by the `Worst` value instead. This is the overlap proportion corresponding to the smallest separation statistic across all pairwise comparisons between group A and the other groups. (In other words, this is the proportion for the pairwise comparison that exhibits the worst discrimination between distributions.) By using this metric, genes can only achieve a high ranking if the separation statistics between group A and *all* other groups are large. This tends to be quite conservative but can be helpful for quickly identifying uniquely differentially expressed markers.

Accounting for uninteresting variation

If the experiment has known (and uninteresting) factors of variation, these can be included in `design` or `block`. The approach used to remove these factors depends on which argument is used. If there is only one factor, using `block` is recommended whereby the levels of the factor are defined as separate groups. Overlaps between groups are computed within each block, and a weighted mean (based on the number of cells in each block) of the overlaps is taken across all blocks.

This approach avoids the need for linear modelling and the associated assumptions regarding normality and correct model specification. In particular, it avoids problems with breaking of ties when counts or expression values are converted to residuals. However, it also makes less use of information, e.g., we ignore any blocks containing cells from only one group. `NA` proportions may also be observed for a pair of groups if there is no block that contains cells from that pair.

For experiments containing multiple factors or covariates, a linear model is fitted to the expression values with an appropriate matrix in `design`. Overlap proportions are then computed using the residuals of the fitted model. This approach is not ideal, requiring log-transformed `x` and setting of `lower.bound` - see `?correlatePairs` for a related discussion. Where possible for one-way layouts, we suggest using `block` instead.

Author(s)

Aaron Lun

See Also

findMarkers

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)
```

Parallel analysis *Parallel analysis for PCA*

Description

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

Usage

```
## S4 method for signature 'ANY'
parallelPCA(x, subset.row=NULL, scale=NULL, value=c("pca", "n", "lowrank"),
    min.rank=5, max.rank=100, niters=50, threshold=0.1, approximate=FALSE,
    irlba.args=list(), 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

x	A numeric matrix of log-expression values for parallelPCA,ANY-method, or a SingleCellExperiment object containing such values for parallelPCA,SingleCellExperiment-met
subset.row	A logical, integer or character vector indicating the rows of x to use for PCA. All genes are used by default.
scale	A numeric vector specifying the scaling to apply to each row of x, if any.
value	A string specifying the type of value to return; the PCs, the number of retained components, or a low-rank approximation.
min.rank, max.rank	Integer scalars specifying the minimum and maximum number of PCs to retain.
niters	Integer scalar specifying the number of iterations to use for the parallel analysis.
threshold	Numeric scalar representing the "p-value" threshold above which PCs are to be ignored.
approximate	A logical scalar indicating whether approximate SVD should be performed via irlba.
irlba.args	A named list of additional arguments to pass to irlba when approximate=TRUE.
BPPARAM	A BiocParallelParam object.

...	Further arguments to pass to denoisePCA,ANY-method.
assay.type	A string specifying which assay values to use.
get.spikes	A logical scalar specifying whether spike-in transcripts should be used. This will be intersected with subset.row if the latter is specified.
sce.out	A logical scalar specifying whether a modified SingleCellExperiment 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 approximate=TRUE, which will 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")
```

Quick clustering *Quick clustering of cells*

Description

Cluster similar cells based on rank correlations in their gene expression profiles.

Usage

```
## S4 method for signature 'ANY'
quickCluster(x, min.size=200, max.size=NULL, method=c("hclust", "igraph"),
    pc.approx=TRUE, get.ranks=FALSE, subset.row=NULL, min.mean=1, ...)

## 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.
`min.size`	An integer scalar specifying the minimum size of each cluster.
`max.size`	An integer scalar specifying the maximum size of each cluster.
`get.ranks`	A logical scalar specifying whether a matrix of adjusted ranks should be returned.
`method`	A string specifying the clustering method to use.
`pc.approx`	Argument passed to `buildSNNGraph` when `method="igraph"`, otherwise ignored.
`subset.row`	A logical, integer or character scalar indicating the rows of `x` to use.
`min.mean`	A numeric scalar specifying the filter to be applied on the average count for each filter prior to computing ranks. Disabled by setting to `NULL`.
`...`	For `quickCluster,ANY-method`, additional arguments to be passed to `cutreeDynamic` for `method="hclust"`, or `buildSNNGraph` for `method="igraph"`. For `quickCluster,SingleCellEx` additional arguments to pass to `quickCluster,ANY-method`.
`assay.type`	A string specifying which assay values to use, e.g., `"counts"` or `"logcounts"`.
`get.spikes`	A logical specifying whether spike-in transcripts should be used.

Details

This function provides a correlation-based approach to quickly define clusters of a minimum size `min.size`. Two clustering strategies are available:

- If `method="hclust"`, a distance matrix is constructed using Spearman's rho on the counts between cells. (Some manipulation is performed to convert Spearman's rho into a proper distance metric.) Hierarchical clustering is performed and a dynamic tree cut 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 `cluster_fast_greedy` function. Again, neighbors are identified using distances based on Spearman's rho. This should be used in situations where there are too many cells to build a distance matrix.

A correlation-based approach is preferred here as it is invariant to scaling normalization. This avoids circularity between normalization and clustering, e.g., in `computeSumFactors`.

Value

If `get.ranks=FALSE`, a character vector of cluster identities for each cell in `counts` is returned.

If `get.ranks=TRUE`, a numeric matrix is returned where each column contains ranks for the expression values in each cell.

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 `cluster_fast_greedy`. 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`.

Gene selection

In `quickCluster,SingleCellExperiment-method`, spike-in transcripts are not used by default as they provide little information on the biological similarities between cells. This may not be the case if subpopulations differ by total RNA content, in which case setting `get.spikes=TRUE` may provide more discriminative power.

Users can also set `subset.row` to specify which rows of `x` are to be used to calculate correlations. This is equivalent to but more efficient than subsetting `x` directly, as it avoids constructing a (potentially large) temporary matrix. Note that if `subset.row` is specified, it will intersect with any setting of `get.spikes`.

By default, the function will also filter out genes with average counts (as defined by `calcAverage`) below `min.mean`. 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. This can be disabled completely by setting it to `NULL`.

Obtaining the scaled and centred ranks

Users can also set `get.ranks=TRUE`, in which case a matrix of ranks will be returned. Each column contains the ranks for the expression values within a single cell after standardization and mean-centring. Computing Euclidean distances between the rank vectors for pairs of cells will yield the same correlation-based distance as that used above. This allows users to apply their own clustering algorithms on the ranks, which protects against outliers and is invariant to scaling (at the cost of sensitivity).

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

`cutreeDynamic`, `computeSumFactors`, `buildSNNGraph`

Examples

```
set.seed(100)
popsize <- 200
ngenes <- 1000
all.facs <- 2^rnorm(popsize, sd=0.5)
counts <- matrix(rnbinom(ngenes*popsize, mu=all.facs, size=1), ncol=popsize, byrow=TRUE)

clusters <- quickCluster(counts, min.size=20)
clusters <- quickCluster(counts, method="igraph")
```

---

sandbag *Cell cycle phase training*

Description

Use gene expression data to train a classifier for cell cycle phase.

Usage

```
## 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.
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	A logical, integer or character scalar indicating the rows of x to use.
...	Additional arguments to pass to sandbag,ANY-method.
assay.type	A string specifying which assay values to use, e.g., "counts" or "logcounts".
get.spikes	A logical specifying whether spike-in transcripts should be used.

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.

For sandbag,SingleCellExperiment-method, the matrix of counts is used but can be replaced with expression values by setting assays. 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. Users can also manually select which rows to use via subset.row, which will override any setting of get.spikes.

While sandbag and its partner function cyclone were originally designed for cell cyclone phase classification, the same computational strategy can be used to classify cells into any mutually exclusive groupings. Any number and nature of groups can be specified in phases, e.g., differentiation lineages, activation states. Only the names of phases need to be modified to reflect the biology being studied.

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

Examples

```
ncells <- 50
ngenes <- 20
training <- matrix(rnorm(ncells*ngenes), ncol=ncells)
rownames(training) <- paste0("X", seq_len(ngenes))

is.G1 <- 1:20
is.S <- 21:30
is.G2M <- 31:50
out <- sandbag(training, 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"))
```

Selector plot *Construct a selector plot via Shiny*

Description

Generate an interactive Shiny plot in which cells can be selected for further analysis.

Usage

```
selectorPlot(x, y, persist=FALSE, plot.width=5, plot.height=500, run=TRUE, pch=16, ...)
```

Arguments

x, y	Numeric vectors of x-y coordinates, of length equal to the number of cells.
persist	A logical scalar indicating whether selections should persist after stopping the app.
plot.width	A numeric scalar specifying the plot width, see width in ?column.
plot.height	A numeric scalar specifying the plot height in pixels.
run	A logical scalar specifying whether the Shiny app should be run.
pch, ...	Other arguments to pass to plot.

Details

Note that this function is deprecated; we suggest using the **iSEE** package for data exploration and point selection instead.

This function will return a Shiny app object that can be run with runApp. The aim is to perform dimensionality reduction to obtain coordinates for each cell, e.g., from PCA or t-SNE. These coordinates can be plotted with selectorPlot, and subpopulations of interest can be interactively selected. The selections can then be saved for further manipulation in R.

The app allows users to select groups of cells; mark them as cells of interest; and then save the marked cells into a list. Currently marked cells will be shown in red, previously saved cells are shown in orange, and all other cells are shown in grey. The distribution of saved cells is also shown in a separate plot indicating the list element to which they were saved. This can be repeated multiple times to obtain several groups of interest.

Several buttons are available within the app:

"Select": Marks the current selection of cells.

"Deselect": Unmarks the current selection of cells.

"Clear selection": Unmarks all currently marked cells.

"Add to list": Saves currently marked cells into a list.

"Reset all": Removes all marking, removes all saved cells from the list.

"Save list to R": Stops the app and returns the list of saved cells to R.

Value

If run=FALSE, a Shiny app object is returned, which can be run with runApp. This transfers control to a browser window where cells can be selected. Upon stopping the app with the "Save list to R" button, control is transferred back to R and the list of saved cells is returned. Each element of the list is a logical vector indicating which cells were saved in that group of interest.

If run=TRUE, a Shiny app object is created and run. This returns a list of saved cells upon stopping the app as previously described.

Author(s)

Aaron Lun

See Also

runApp

Examples

```
# Setting up PCs.
example(SingleCellExperiment)
pcs <- reducedDim(sce, "PCA")
x <- pcs[,1]
y <- pcs[,2]

# Creating the app object.
app <- selectorPlot(x, y, run=FALSE)
if (interactive()) { saved <- shiny::runApp(app) }

## Not run: # Running the app directly from the function.
```

```
saved <- selectorPlot(x, y)

## End(Not run)
```

`Spike-in normalization` *Normalization with spike-in counts*

Description

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

Usage

```
## S4 method for signature 'SingleCellExperiment'
computeSpikeFactors(x, type=NULL, assay.type="counts", sf.out=FALSE, general.use=TRUE)
```

Arguments

`x`	A SingleCellExperiment object with rows corresponding spike-in transcripts.
`type`	A character vector specifying which spike-in sets to use.
`assay.type`	A string indicating which assay contains the counts.
`sf.out`	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 size factor for each cell is defined as the sum of all spike-in counts in each cell. This is equivalent to normalizing to equalize spike-in coverage between cells. Size factors are scaled so that the mean of all size factors is unity, for standardization purposes if one were to compare different sets of size factors.

Spike-in counts are assumed to be stored in the rows specified by `isSpike(x)`. This specification should have been performed by supplying the names of the spike-in sets – see `?isSpike` for more details. By default, if multiple spike-in sets are available, all of them will be used to compute the size factors. The function can be restricted to a subset of the spike-ins by specifying the names of the desired spike-in sets in `type`. An error will be raised if no spike-in rows are detected.

By default, the function will store several copies of the same size factors in the output object. One copy will also be stored in `sizeFactors(x, type=s)`, where `s` is the name of each spike-in set in `type`. (If `type=NULL`, a copy is stored for every spike-in set, as all of them would be used to compute the size factors.) Separate storage allows spike-in-specific normalization in `normalize`. If `general.use=TRUE`, a copy will also be stored in `sizeFactors(x)` for normalization of all genes.

Value

If `sf.out=TRUE`, a numeric vector of size factors is returned directly.

Otherwise, an object of class `x` is returned, containing size factors for all cells. A copy of the vector is stored for each spike-in set that was used to compute the size factors. If `general.use=TRUE`, a copy is also stored for use by non-spike-in genes.

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

isSpike

Examples

```
################
# Mocking up some data.
set.seed(100)
ncells <- 200

nspikes <- 100
spike.means <- 2^runif(nspikes, 3, 8)
spike.disp <- 100/spike.means + 0.5
spike.data <- matrix(rnbinom(nspikes*ncells, mu=spike.means, size=1/spike.disp), ncol=ncells)

ngenes <- 2000
cell.means <- 2^runif(ngenes, 2, 10)
cell.disp <- 100/cell.means + 0.5
cell.data <- matrix(rnbinom(ngenes*ncells, mu=cell.means, size=1/cell.disp), ncol=ncells)

combined <- rbind(cell.data, spike.data)
colnames(combined) <- seq_len(ncells)
rownames(combined) <- seq_len(nrow(combined))
y <- SingleCellExperiment(list(counts=combined))
isSpike(y, "Spike") <- ngenes + seq_len(nspikes)

################
# Computing and storing spike-in size factors.
y2 <- computeSpikeFactors(y)
head(sizeFactors(y2))
head(sizeFactors(y2, type="Spike"))

# general.use=FALSE does not modify general size factors
sizeFactors(y2) <- 1
sizeFactors(y2, type="Spike") <- 1
y2 <- computeSpikeFactors(y2, general.use=FALSE)
head(sizeFactors(y2))
head(sizeFactors(y2, type="Spike"))
```

---

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.

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 SingleCellExperiment 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 technicalCV2,ANY-method.
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 glmgam.fit. 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,SingleCellE
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

`glmgam.fit`, `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.

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 are 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 *Fit a variance trend*

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(),
    span=NULL, family=NULL, degree=NULL, df=NULL, start=NULL,
    block=NULL, design=NULL, weighted=TRUE, min.mean=0.1, subset.row=NULL)

## 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 SingleCellExperiment 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 loess when method="loess".
spline.args	A named list of arguments to pass to robustSmoothSpline when method="spline".
nls.args	A named list of arguments to pass to nls when parametric=TRUE.
span, family, degree	Deprecated arguments to pass to loess.
df	Deprecated argument to pass to ns.
start	Deprecated argument to pass to nls.
block	A factor specifying the blocking level for each cell.
design	A numeric matrix describing the uninteresting factors contributing to expression in each cell.
weighted	A logical scalar indicated whether weighted trend fitting should be performed when block!=NULL.
min.mean	A numeric scalar specifying the minimum mean log-expression in order for a gene to be used for trend fitting.
subset.row	A logical, integer or character scalar indicating the rows of x to use.
...	Additional arguments to pass to trendVar,ANY-method.
assay.type	A string specifying which assay values in x 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 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 algorihtm 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 the `smooth.spline` function Arguments are specified with `spline.args`, with a default degrees of freedom of `df=4` unless otherwise specified. Splines can be more effective than loess at capturing smooth curves with strong non-linear gradients.

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, re-run normalize for each subset to centre the size factors (see below), and run trendVar and decomposeVar for each subset separately. Results from all levels are then consolidated using the combineVar function. This 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 should be used.

Users can also directly specify which rows to use with `subset.row`. All of these parameters - including `min.mean` and `use.spikes` - interact with each other sensibly, by taking the intersection of rows that fulfill all criteria. For example, if `subset.row` is specified and `use.spikes=TRUE`, rows are only used if they are both spike-in transcripts *and* in `subset.row`. Otherwise, if `use.spikes=FALSE`, only rows in `subset.row` that are *not* spike-in transcripts are used.

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

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 **clustering**
cyclone, 13
sandbag, 47
∗Topic **correlation**
correlatePairs, 9
∗Topic **normalization**
Deconvolution Methods, 19
Quick clustering, 45
Spike-in normalization, 51
∗Topic **variance**
decomposeVar, 16
Distance-to-median, 25
improvedCV2, 32
multiBlockVar, 39
technicalCV2, 52
testVar, 55
trendVar, 56
∗Topic
correlatePairs, 9

bpparam, *13*
build∗NNGraph, 2
buildKNNGraph *(*build∗NNGraph*)*, 2
buildKNNGraph,ANY-method *(*build∗NNGraph*)*, 2
buildKNNGraph,SingleCellExperiment-method *(*build∗NNGraph*)*, 2
buildSNNGraph, *5*, *45–47*
buildSNNGraph *(*build∗NNGraph*)*, 2
buildSNNGraph,ANY-method *(*build∗NNGraph*)*, 2
buildSNNGraph,SingleCellExperiment-method *(*build∗NNGraph*)*, 2

calcAverage, *21*, *46*
cluster_fast_greedy, *46*
clusterModularity, 4
column, *49*
combineVar, 6, *17*, *23*, *39*, *40*, *59*
computeSpikeFactors, *21*, *57*, *60*
computeSpikeFactors *(*Spike-in normalization*)*, 51
computeSpikeFactors,SingleCellExperiment-method *(*Spike-in normalization*)*, 51
computeSumFactors, *46*, *47*, *60*
computeSumFactors *(*Deconvolution Methods*)*, 19
computeSumFactors,ANY-method *(*Deconvolution Methods*)*, 19
computeSumFactors,SingleCellExperiment-method *(*Deconvolution Methods*)*, 19
convertTo, 8
convertTo,SingleCellExperiment-method *(*convertTo*)*, 8
cor, *13*
correlateNull *(*correlatePairs*)*, 9
correlatePairs, 9, *42*
correlatePairs,ANY-method *(*correlatePairs*)*, 9
correlatePairs,SingleCellExperiment-method *(*correlatePairs*)*, 9
cutreeDynamic, *45–47*
cyclone, 13, *48*, *49*
cyclone,ANY-method *(*cyclone*)*, 13
cyclone,SingleCellExperiment-method *(*cyclone*)*, 13

decomposeVar, *6*, *7*, 16, *23*, *24*, *39*, *40*, *56*, *58–60*
decomposeVar,ANY,list-method *(*decomposeVar*)*, 16
decomposeVar,SingleCellExperiment,list-method *(*decomposeVar*)*, 16
Deconvolution Methods, 19
Denoise with PCA, 22
denoisePCA, *44*
denoisePCA *(*Denoise with PCA*)*, 22
denoisePCA,ANY-method *(*Denoise with PCA*)*, 22
denoisePCA,SingleCellExperiment-method *(*Denoise with PCA*)*, 22
DESeqDataSetFromMatrix, *9*
DGEList, *9*
Distance-to-median, 25
DM *(*Distance-to-median*)*, 25

estimateSizeFactorsForMatrix, *53*, *54*
Explore Data, 26

exploreData *(*Explore Data*)*, 26

findMarkers, 28, *41–43*
findMarkers,ANY-method *(*findMarkers*)*, 28
findMarkers,SingleCellExperiment-method *(*findMarkers*)*, 28
fitFDistRobustly, *56*

get.knn, *3, 4*
get.knnx, *38*
glmgam.fit, *53, 54*

improvedCV2, 32
improvedCV2,ANY-method *(*improvedCV2*)*, 32
improvedCV2,SingleCellExperiment-method *(*improvedCV2*)*, 32
irlba, *23*, *36*, *38*, *43*
isSpike, *51, 52*

loess, *57, 58*, *60*

make_graph, *4*
makeTechTrend, 34
mnnCorrect, *24*, 36
modularity, *5*
multiBlockVar, 39

newCellDataSet, *9*
nls, *57, 58*, *60*
normalize, *31*, *35*, *37*, *39, 40*, *51*, *57*, *59, 60*
ns, *34*, *57*

overlapExprs, 40
overlapExprs,ANY-method *(*overlapExprs*)*, 40
overlapExprs,SingleCellExperiment-method *(*overlapExprs*)*, 40

Parallel analysis, 43
parallelPCA *(*Parallel analysis*)*, 43
parallelPCA,ANY-method *(*Parallel analysis*)*, 43
parallelPCA,SingleCellExperiment-method *(*Parallel analysis*)*, 43
plotPCA, *24*
prcomp_irlba, *3, 4*

Quick clustering, 45
quickCluster, *4*, *20*, *22*
quickCluster *(*Quick clustering*)*, 45
quickCluster,ANY-method *(*Quick clustering*)*, 45
quickCluster,SingleCellExperiment-method *(*Quick clustering*)*, 45

removeBatchEffect, *24*
robustSmoothSpline, *32*, *57*
runApp, *26, 27*, *50*
runmed, *25*

sandbag, *14, 15*, 47
sandbag,ANY-method *(*sandbag*)*, 47
sandbag,SingleCellExperiment-method *(*sandbag*)*, 47
Selector plot, 49
selectorPlot *(*Selector plot*)*, 49
sizeFactors, *33*, *53*
smooth.spline, *32*, *58*
Spike-in normalization, 51
svd, *38*

technicalCV2, *32*, *34*, 52
technicalCV2,ANY-method *(*technicalCV2*)*, 52
technicalCV2,SingleCellExperiment-method *(*technicalCV2*)*, 52
testVar, *16–18*, 55
treat, *30*
trendVar, *16–18*, *23, 24*, *35*, *39, 40*, *56*, 56, *59*
trendVar,ANY-method *(*trendVar*)*, 56
trendVar,SingleCellExperiment-method *(*trendVar*)*, 56