Package: batchelor (via r-universe)

November 1, 2026

Version 1.28.0

Date 2026-04-09

Title Single-Cell Batch Correction Methods

Depends SingleCellExperiment

Imports SummarizedExperiment, S4Vectors, BiocGenerics, Rcpp, stats, methods, utils, igraph, BiocNeighbors, BiocSingular, Matrix, SparseArray, DelayedArray (>= 0.31.5), DelayedMatrixStats, BiocParallel, scuttle, ResidualMatrix, ScaledMatrix, beachmat

Suggests testthat, BiocStyle, knitr, rmarkdown, scran, scater, bluster, scRNAseq

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

LinkingTo Rcpp

Description Implements a variety of methods for batch correction of single-cell (RNA sequencing) data. This includes methods based on detecting mutually nearest neighbors, as well as several efficient variants of linear regression of the log-expression values. Functions are also provided to perform global rescaling to remove differences in depth between batches, and to perform a principal components analysis that is robust to differences in the numbers of cells across batches.

License GPL-3

NeedsCompilation yes

VignetteBuilder knitr

SystemRequirements C++11

RoxygenNote 7.3.2

Config/pak/sysreqs libglpk-dev libxml2-dev zlib1g-dev

Repository Bioconductor 3.23

Date/Publication 2026-04-28 12:49:53 UTC

RemoteUrl https://github.com/bioc/batchelor

RemoteRef RELEASE_3_23

RemoteSha 5823437ffc4b954b83a9e0f26867da1d20fea268

Contents

applyMultiSCE . . . 2
batchCorrect . . . 5
batchelor-restrict . . . 8
BatchelorParam-class . . . 9
checkBatchConsistency . . . 11
clusterMNN . . . 12
convertPCsToSCE . . . 15
correctExperiments . . . 17
cosineNorm . . . 19
diagnostics-cluster . . . 21
divideIntoBatches . . . 22
fastMNN . . . 23
intersectRows . . . 30
mnnCorrect . . . 32
mnnDeltaVariance . . . 38
multiBatchNorm . . . 41
multiBatchPCA . . . 43
noCorrect . . . 47
quickCorrect . . . 48
reducedMNN . . . 51
regressBatches . . . 53
rescaleBatches . . . 56

Index **59**

`applyMultiSCE` *Apply function over multiple SingleCellExperiments*

Description

A generalization of `applySCE` to apply a function to corresponding parts of multiple SingleCellExperiments, each of which have one or more alternative Experiments.

Usage

```
applyMultiSCE(
  ...,
  FUN,
  WHICH = NULL,
  COMMON.ARGS = list(),
  MAIN.ARGS = list(),
  ALT.ARGS = list(),
  SIMPLIFY = TRUE
)
```

Arguments

...	One or more SingleCellExperiment objects containing counts and size factors. Each object should contain the same number of rows, corresponding to the same genes in the same order. If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. If a single object is supplied, `batch` should also be specified. Alternatively, one or more lists of SingleCellExperiments can be provided; this is flattened as if the objects inside were passed directly to ... .
FUN	Any function that accepts multiple SummarizedExperiment or SingleCellExperiment objects.
WHICH	A character or integer vector containing the names or positions of alternative Experiments to loop over. Defaults to all alternative Experiments that are present in each SingleCellExperiment in ... .
COMMON.ARGS	Further (named) arguments to pass to all calls to FUN.
MAIN.ARGS	A named list of arguments to pass to calls to FUN involving the main Experiment(s) only. Alternatively NULL, in which case the function is *not* applied to the main Experiment.
ALT.ARGS	A named list where each entry is named after an alternative Experiment and contains named arguments to use in FUN for that Experiment.
SIMPLIFY	Logical scalar indicating whether the output should be simplified to one or more SingleCellExperiments.

Details

This function is a generalization of applySCE whereby corresponding Experiments from ... are passed to FUN. To illustrate, if we passed objects x, y and z in ...:

1. We first call FUN on the set of all main Experiments from ..., obtaining a result equivalent to `FUN(x, y, z)` (more on the other arguments later).
2. Then we call FUN on the set of all first alternative Experiments. This is equivalent to `FUN(altExp(x), altExp(y), altExp(z))`.
3. Then we call FUN on the set of all second alternative Experiments. This is equivalent to `FUN(altExp(x, 2), altExp(y, 2), altExp(z, 2))`.
4. And so on.

In effect, much like applySCE is analogous to lapply, applyMultiSCE is analogous to mapply. This allows users to easily apply the same function to all the Experiments (main and alternative) in a list of SingleCellExperiment objects.

Arguments in COMMON.ARGS (plus some extra arguments, see below) are passed to all calls to FUN. Arguments in MAIN.ARGS are only used in the call to FUN on the main Experiments. Arguments in ALT.ARGS are passed to the call to FUN on the alternative Experiments of the same name. For the last two, any arguments therein will override arguments of the same name in COMMON.ARGS.

Arguments in ... that are *not* SingleCellExperiments are actually treated as additional arguments for COMMON.ARGS. This is purely intended as a user convenience, to avoid the need to write `COMMON.ARGS=list()`

when specifying these arguments. However, explicitly using COMMON.ARGS is the safer approach and recommended for developers.

By default, looping is performed over alternative Experiments with names that are present across all entries of .... Values of WHICH should be unique if any simplification of the output is desired. If MAIN.ARGS=NULL, the main Experiment is ignored and the function is only applied to the alternative Experiments.

The default of SIMPLIFY=TRUE is aims to make the output easier to manipulate. If FUN returns a SingleCellExperiment, the outputs across main and alternative Experiments are simplified into a single SingleCellExperiment. If FUN returns a list of SingleCellExperiments of the same length, the outputs are simplified into one list of SingleCellExperiments. This assumes that WHICH contains no more than one reference to each alternative Experiment in x.

Value

In most cases or when SIMPLIFY=FALSE, a list is returned containing the output of FUN applied to each *corresponding* Experiment across all .... If MAIN.ARGS is not NULL, the first entry corresponds to the result generated from the main Experiments; all other results are generated according to the entries specified in WHICH and are named accordingly.

If SIMPLIFY=TRUE and certain conditions are fulfilled, we can either return:

- A single SingleCellExperiment, if all calls to FUN return a SingleCellExperiment. Here, the results of FUN on the main/alternative Experiments in ... are mapped to the main or alternative Experiments of the same name in the output.
- A list of SingleCellExperiments, if all calls to FUN return a list of SingleCellExperiments of the same length. The altExps of each output SingleCellExperiment contains the results from the corresponding call to FUN on the alternative Experiments of the same name in ....

In both cases, the aim is to mirror the organization of Experiments in each entry of ....

Author(s)

Aaron Lun

See Also

applySCE, for the simpler version that involves only one SingleCellExperiment object.

simplifyToSCE, for the conditions required for simplification.

Examples

```
# Setting up some objects with alternative Experiments.
d1 <- matrix(rnbinom(50000, mu=10, size=1), ncol=100)
sce1 <- SingleCellExperiment(list(counts=d1))
sizeFactors(sce1) <- runif(ncol(d1))
altExp(sce1, "Spike") <- sce1
altExp(sce1, "Protein") <- sce1

d2 <- matrix(rnbinom(20000, mu=50, size=1), ncol=40)
sce2 <- SingleCellExperiment(list(counts=d2))
```

```
sizeFactors(sce2) <- runif(ncol(d2))
altExp(sce2, "Spike") <- sce2
altExp(sce2, "Protein") <- sce2

# Applying a function over the main and alternative experiments.
normed <- applyMultiSCE(sce1, sce2, FUN=multiBatchNorm)
normed
altExp(normed[[1]]) # contains log-normalized values

regressed <- applyMultiSCE(normed, FUN=regressBatches)
regressed
altExp(regressed) # contains corrected expression values

rescaled <- applyMultiSCE(normed, FUN=rescaleBatches)
rescaled
altExp(rescaled) # contains corrected expression values

# We can also specify 'batch=' directly.
combined <- cbind(sce1, sce2)
batch <- rep(1:2, c(ncol(sce1), ncol(sce2)))

normed <- applyMultiSCE(combined, batch=batch, FUN=multiBatchNorm)
normed
altExp(normed) # contains log-normalized values

regressed <- applyMultiSCE(normed, batch=batch, FUN=regressBatches)
regressed
altExp(regressed) # contains corrected expression values

rescaled <- applyMultiSCE(normed, batch=batch, FUN=rescaleBatches)
rescaled
altExp(rescaled) # contains corrected expression values
```

---

batchCorrect *Batch correction methods*

Description

A common interface for single-cell batch correction methods.

Usage

```
batchCorrect(
  ...,
  batch = NULL,
  restrict = NULL,
  subset.row = NULL,
  correct.all = FALSE,
  assay.type = NULL,
```

```
  PARAM
)

## S4 method for signature 'ClassicMnnParam'
batchCorrect(
  ...,
  batch = NULL,
  restrict = NULL,
  subset.row = NULL,
  correct.all = FALSE,
  assay.type = "logcounts",
  PARAM
)

## S4 method for signature 'FastMnnParam'
batchCorrect(
  ...,
  batch = NULL,
  restrict = NULL,
  subset.row = NULL,
  correct.all = FALSE,
  assay.type = "logcounts",
  PARAM
)

## S4 method for signature 'RescaleParam'
batchCorrect(
  ...,
  batch = NULL,
  restrict = NULL,
  subset.row = NULL,
  correct.all = FALSE,
  assay.type = "logcounts",
  PARAM
)

## S4 method for signature 'RegressParam'
batchCorrect(
  ...,
  batch = NULL,
  restrict = NULL,
  subset.row = NULL,
  correct.all = FALSE,
  assay.type = "logcounts",
  PARAM
)

## S4 method for signature 'NoCorrectParam'
```

```
batchCorrect(
  ...,
  batch = NULL,
  restrict = NULL,
  subset.row = NULL,
  correct.all = FALSE,
  assay.type = "logcounts",
  PARAM
)
```

Arguments

`...`	One or more matrix-like objects containing single-cell gene expression matrices. Alternatively, one or more SingleCellExperiment objects can be supplied. If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. Objects of different types can be mixed together. If a single object is supplied, `batch` should also be specified.
`batch`	A factor specifying the batch of origin for each cell if only one batch is supplied in `...`. This will be ignored if two or more batches are supplied.
`restrict`	A list of length equal to the number of objects in `...`. Each entry of the list corresponds to one batch and specifies the cells to use when computing the correction.
`subset.row`	A vector specifying the subset of genes to use for correction. Defaults to `NULL`, in which case all genes are used.
`correct.all`	A logical scalar indicating whether to return corrected expression values for all genes, even if `subset.row` is set. Used to ensure that the output is of the same dimensionality as the input.
`assay.type`	A string or integer scalar specifying the assay to use for correction. Only used for SingleCellExperiment inputs.
`PARAM`	A BatchelorParam object specifying the batch correction method to dispatch to, and the parameters with which it should be run. ClassicMnnParam will dispatch to `mnnCorrect`; FastMnnParam will dispatch to `fastMNN`; RescaleParam will dispatch to `rescaleBatches`; and RegressParam will dispatch to `regressBatches`.

Details

Users can pass parameters to each method directly via `...` or via the constructors for `PARAM`. While there is no restriction on which parameters go where, we recommend only passing data-agnostic and method-specific parameters to `PARAM`. Data-dependent parameters - and indeed, the data themselves - should be passed in via `...`. This means that different data sets can be used without modifying `PARAM`, allowing users to switch to a different algorithm by only changing `PARAM`.

Value

A SingleCellExperiment where the first assay contains corrected gene expression values for all genes. Corrected values should be returned for all genes if `subset.row=NULL` or if `correct.all=TRUE`; otherwise they should only be returned for the genes in the subset.

Cells should be reported in the same order that they are supplied. In cases with multiple batches, the cell identities are simply concatenated from successive objects in their specified order, i.e., all cells from the first object (in their provided order), then all cells from the second object, and so on. For a single input object, cells should be reported in the same order as the input.

The `colData` slot should contain `batch`, a vector specifying the batch of origin for each cell.

Author(s)

Aaron Lun

See Also

BatchelorParam classes to determine dispatch.

`correctExperiments`, to obtain corrected values while retaining the original expression data.

Examples

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

# Switching easily between batch correction methods.
m.out <- batchCorrect(B1, B2, PARAM=ClassicMnnParam())
f.out <- batchCorrect(B1, B2, PARAM=FastMnnParam(d=20))
r.out <- batchCorrect(B1, B2, PARAM=RescaleParam(pseudo.count=0))
n.out <- batchCorrect(B1, B2, PARAM=NoCorrectParam())
```

---

batchelor-restrict *Using restriction*

Description

Using restriction

Motivation

It is possible to compute the correction using only a subset of cells in each batch, and then extrapolate that correction to all other cells. This may be desirable in experimental designs where a control set of cells from the same source population were run on different batches. Any difference in the controls must be artificial in origin and can be directly removed without making further biological assumptions. Similarly, if certain cells are known to be of a batch-specific subpopulation, it may be desirable to exclude them to ensure that they are not inadvertently used during the batch correction.

Setting the `restrict` argument

To perform restriction, users should set `restrict` to specify the subset of cells in each batch to be used for correction. This should be set to a list of length equal to the number of objects passed to the `...` argument of the batch correction function. Each element of this list should be a subsetting vector to be applied to the columns of the corresponding batch. A `NULL` element indicates that all the cells from a batch should be used. In situations where one input object contains multiple batches, `restrict` should simply a list containing a single subsetting vector for that object.

Correction functions that support `restrict` will only use the restricted subset of cells in each batch to perform the correction. For example, `fastMNN` will only use the restricted cells to identify MNN pairs and the center of the orthogonalization. However, it will apply the correction to all cells in each batch - hence the extrapolation. This means that the output is always of the same dimensionality, regardless of whether `restrict` is specified.

As a general rule, users can expect the corrected values in the restricted cells to be the same as if the inputs were directly subsetted to only contain those cells (see Examples). This is appealing as it demonstrates that correction only uses information from the restricted subset of cells. If batch correction functions do not follow this rule, they will explicitly state so, e.g., in `?fastMNN`.

Author(s)

Aaron Lun

See Also

`rescaleBatches`, `regressBatches`, `fastMNN` and `mnnCorrect`, as examples of batch correction methods that support restriction.

Examples

```
means <- 2^rgamma(1000, 2, 1)
A1 <- matrix(rpois(10000, lambda=means), ncol=50) # Batch 1
A2 <- matrix(rpois(10000, lambda=means*runif(1000, 0, 2)), ncol=50) # Batch 2

B1 <- log2(A1 + 1)
B2 <- log2(A2 + 1)
out <- regressBatches(B1, B2, restrict=list(1:10, 1:10))
assay(out)[,c(1:10, 50+1:10)]

# Compare to actual subsetting:
out.sub <- regressBatches(B1[,1:10], B2[,1:10])
assay(out.sub)
```

`BatchelorParam-class` *BatchelorParam methods*

Description

Constructors and methods for the batchelor parameter classes.

Usage

```
ClassicMnnParam(...)

FastMnnParam(...)

RescaleParam(...)

RegressParam(...)

NoCorrectParam(...)
```

Arguments

...	Named arguments to pass to individual methods upon dispatch. These should not include arguments named in the batchCorrect generic.

Details

BatchelorParam objects are intended to store method-specific parameter settings to pass to the batchCorrect generic. These values should refer to data-agnostic parameters; parameters that depend on data (or the data itself) should be specified directly in the batchCorrect call.

The BatchelorParam classes are all derived from SimpleList objects and have the same available methods, e.g., [[, $. These can be used to access or modify the object after construction.

Note that the BatchelorParam class itself is not useful and should not be constructed directly. Instead, users should use the constructors shown above to create instances of the desired subclass.

Value

The constructors will return a BatchelorParam object of the specified subclass, containing parameter settings for the corresponding batch correction method.

Author(s)

Aaron Lun

See Also

batchCorrect, where the BatchelorParam objects are used for dispatch to individual methods.

Examples

```
# Specifying the number of neighbors, dimensionality.
fp <- FastMnnParam(k=20, d=10)
fp

# List-like behaviour:
fp$k
fp$k <- 10
fp$k
```

checkBatchConsistency *Check batch inputs*

Description

Utilities to check inputs into batch correction functions.

Usage

```
checkBatchConsistency(batches, cells.in.columns = TRUE)

checkIfSCE(batches)

checkRestrictions(batches, restrictions, cells.in.columns = TRUE)
```

Arguments

batches	A list of batches, usually containing gene expression matrices or SingleCellExperiment objects.
cells.in.columns	A logical scalar specifying whether batches contain cells in the columns.
restrictions	A list of length equal to batches, specifying the cells in each batch that should be used for correction.

Details

These functions are intended for internal use and other package developers.

checkBatchConsistency will check whether the input batches are consistent with respect to the size of the dimension containing features (i.e., not cells). It will also verify that the dimension names are consistent, to avoid problems from variable ordering of rows/columns in the inputs.

checkRestrictions will check whether restrictions are consistent with the supplied batches, in terms of the length and names of the two lists. It will also check that each batch contains at least one usable cell after restriction.

Value

checkBatchConsistency return an invisible NULL if there are no errors.

checkIfSCE will return a logical vector specifying whether each element of batches is a SingleCellExperiment objects.

checkRestrictions will return NULL if restrictions=NULL. Otherwise, it will return a list by taking restrictions and converting each non-NULL element into an integer subsetting vector.

Author(s)

Aaron Lun

See Also

divideIntoBatches

Examples

```
checkBatchConsistency(list(cbind(1:5), cbind(1:5, 2:6)))
try( # fails
    checkBatchConsistency(list(cbind(1:5), cbind(1:4, 2:5)))
)
```

---

clusterMNN *Cluster-based MNN*

Description

Perform MNN correction based on cluster centroids, using the corrected centroid coordinates to correct the per-cell expression values with a variable bandwidth Gaussian kernel.

Usage

```
clusterMNN(
  ...,
  batch = NULL,
  restrict = NULL,
  clusters,
  cluster.d = 50,
  cos.norm = TRUE,
  merge.order = NULL,
  auto.merge = FALSE,
  min.batch.skip = 0,
  subset.row = NULL,
  correct.all = FALSE,
  assay.type = "logcounts",
  BSPARAM = IrlbaParam(),
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam()
)
```

Arguments

...	One or more log-expression matrices where genes correspond to rows and cells correspond to columns. Alternatively, one or more SingleCellExperiment objects can be supplied containing a log-expression matrix in the assay.type assay. Each object should contain the same number of rows, corresponding to the same genes in the same order. Objects of different types can be mixed together.

If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. If a single object is supplied, it is assumed to contain cells from all batches, so `batch` should also be specified.

Alternatively, one or more lists of matrices or SingleCellExperiments can be provided; this is flattened as if the objects inside each list were passed directly to ....

`batch` A vector or factor specifying the batch of origin for all cells when only a single object is supplied in .... This is ignored if multiple objects are present.

`restrict` A list of length equal to the number of objects in .... Each entry of the list corresponds to one batch and specifies the cells to use when computing the correction.

`clusters` A list of length equal to ... containing the assigned cluster for each cell in each batch. Alternatively, a BlusterParam object from the **bluster** package, specifying the clustering to be applied to each batch.

`cluster.d` Integer scalar indicating how many PCs should be used in clustering. Only used if `clusters` is not a list. If `NA`, no PCA is performed and clustering is applied directly to the (cosine-normalized) gene expression values.

`cos.norm` A logical scalar indicating whether cosine normalization should be performed on the input data prior to PCA.

`merge.order` An integer vector containing the linear merge order of batches in .... Alternatively, a list of lists representing a tree structure specifying a hierarchical merge order.

`auto.merge` Logical scalar indicating whether to automatically identify the "best" merge order.

`min.batch.skip` Numeric scalar specifying the minimum relative magnitude of the batch effect, below which no correction will be performed at a given merge step.

`subset.row` A vector specifying which features to use for correction.

`correct.all` Logical scalar indicating whether corrected expression values should be computed for genes not in `subset.row`. Only relevant if `subset.row` is not `NULL`.

`assay.type` A string or integer scalar specifying the assay containing the log-expression values. Only used for SingleCellExperiment inputs.

`BSPARAM` A BiocSingularParam object specifying the algorithm to use for PCA. Only used if `clusters` is not a list.

`BNPARAM` A BiocNeighborParam object specifying the nearest neighbor algorithm.

`BPPARAM` A BiocParallelParam object specifying whether the PCA and nearest-neighbor searches should be parallelized.

Details

These functions are motivated by the scenario where each batch has been clustered separately and each cluster has already been annotated with some meaningful biological state. We want to identify which biological states match to each other across batches; this is achieved by identifying mutual nearest neighbors based on the cluster centroids with `reducedMNN`.

MNN pairs are identified with k=1 to ensure that each cluster has no more than one match in another batch. This reduces the risk of inadvertently merging together different clusters from the same batch. By comparison, higher values of k may result in many-to-one mappings between batches such that the correction will implicitly force different clusters together.

Using this guarantee of no-more-than-one mappings across batches, we can form meta-clusters by identifying all components of the resulting MNN graph. Each meta-cluster can be considered to represent some biological state (e.g., cell type), and all of its constituents are the matching clusters within each batch.

As an extra courtesy, clusterMNN will also compute corrected values for each cell. This is done by applying a Gaussian kernel to the correction vectors for the centroids, where the bandwidth is proportional to the distance between that cell and the closest cluster centroid. This yields a smooth correction function that avoids edge effects at cluster boundaries.

If clusters is set to a BlusterParam object (see the **bluster** package), a PCA is performed in each batch with the specified BSPARAM. The PCs are then used in clustering with clusterRows to obtain a list of clusters. This can be used to mimic per-cell batch correction in the absence of *a priori* clusters.

Value

A SingleCellExperiment containing per-cell expression values where each row is a gene and each column is a cell. This has the same format as the output of fastMNN but with an additional cluster field in the colData containing the cluster identity of each cell. The metadata contains:

- merge.info, a DataFrame with the same format as the output of fastMNN. However, the pairs and lost.var refer to the cluster centroids, not the cells.
- clusters, a DataFrame with one row for each cluster across all batches in .... This can be row-indexed by the values in pairs to determine the identity of the clusters in each MNN pair. An additional meta column is provided that describes the meta-cluster to which each cluster belongs.

Author(s)

Aaron Lun

References

Lun ATL (2019). Cluster-based mutual nearest neighbors correction https://marionilab.github.io/FurtherMNN2018/theory/clusters.html

Lun ATL (2019). A discussion of the known failure points of the fastMNN algorithm. https://marionilab.github.io/FurtherMNN2018/theory/failure.html

See Also

reducedMNN, which is used internally to perform the correction.

Examples

```
# Mocking up some data for multiple batches:
means <- matrix(rnorm(3000), ncol=3)
colnames(means) <- LETTERS[1:3]

B1 <- means[,sample(LETTERS[1:3], 500, replace=TRUE)]
B1 <- B1 + rnorm(length(B1))

B2 <- means[,sample(LETTERS[1:3], 500, replace=TRUE)]
B2 <- B2 + rnorm(length(B2)) + rnorm(nrow(B2)) # batch effect.

# Applying the correction with some made-up clusters:
cluster1 <- kmeans(t(B1), centers=10)$cluster
cluster2 <- kmeans(t(B2), centers=10)$cluster
out <- clusterMNN(B1, B2, clusters=list(cluster1, cluster2))

rd <- reducedDim(out, "corrected")
plot(rd[,1], rd[,2], col=out$batch)

# Obtaining the clusters internally.
out2 <- clusterMNN(B1, B2, clusters=bluster::NNGraphParam())
rd2 <- reducedDim(out2, "corrected")
plot(rd2[,1], rd2[,2], col=out$batch)
```

`convertPCsToSCE` *Convert corrected PCs to a SingleCellExperiment*

Description

Convert low-dimensional corrected PCs to a SingleCellExperiment containing corrected expression values. This is a low-level function and most users should not need to call it.

Usage

```
convertPCsToSCE(
  corrected.df,
  pc.info,
  assay.name = "reconstructed",
  dimred.name = "corrected"
)
```

Arguments

`corrected.df`	A DataFrame containing a nested matrix of low-dimensional `corrected` values and a vector of `batch` identities. Typically produced from `reducedMNN`.
`pc.info`	A list containing PCA statistics, in particular a `rotation` matrix. Typically obtained from the `metadata` of the output from `multiBatchPCA`.

`assay.name` String specifying the name of the assay to use to store the corrected expression values.

`dimred.name` String containing the name fo the `reducedDims` to store the low-dimensional corrected values. Set to `NULL` to avoid storing these.

Details

The corrected expression values are obtained by simply taking the crossproduct of the corrected PCs with the rotation matrix. This reverses the original projection to PC space while retaining the effect of the correction. These values are best used for visualization; the low-dimensional corrected coordinates are more efficient for per-cell operations like clustering, while the original uncorrected expression values are safer to interpret for per-gene analyses.

Value

A SingleCellExperiment containing a LowRankMatrix with the corrected per-gene expression values. The `colData` contains the batch identities, the `rowData` contains the rotation matrix, and the `reducedDims` contains the low-dimensional corrected values (if `dimred.name` is not `NULL`). All additional metadata from `corrected.df` and `pc.info` is stored in `metadata`.

Author(s)

Aaron Lun

See Also

`reducedMNN`, to compute `corrected.df`; and `multiBatchPCA`, to compute `pc.info`.

`fastMNN`, which uses this function to obtain low-rank corrected values.

Examples

```
B1 <- matrix(rnorm(10000), nrow=50) # Batch 1
B2 <- matrix(rnorm(10000), nrow=50) # Batch 2

# Equivalent to fastMNN().
cB1 <- cosineNorm(B1)
cB2 <- cosineNorm(B2)
pcs <- multiBatchPCA(cB1, cB2)
mnn.out <- reducedMNN(pcs[[1]], pcs[[2]])

sce <- convertPCsToSCE(mnn.out, metadata(pcs))
sce
```

`correctExperiments` *Correct SingleCellExperiment objects*

Description

Apply a correction to multiple SingleCellExperiment objects, while also combining the assay data and column metadata for easy downstream use. This augments the simpler `batchCorrect` function, which returns only the corrected values.

Usage

```
correctExperiments(
  ...,
  batch = NULL,
  restrict = NULL,
  subset.row = NULL,
  correct.all = FALSE,
  assay.type = "logcounts",
  PARAM = FastMnnParam(),
  combine.assays = NULL,
  combine.coldata = NULL,
  include.rowdata = TRUE,
  add.single = TRUE
)
```

Arguments

`...`	One or more SingleCellExperiment objects. If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. If a single object is supplied, `batch` should also be specified. Alternatively, one or more lists of SingleCellExperiments can be provided; this is flattened so that it is as if the objects inside were passed directly to `...`.
`batch`	A factor specifying the batch of origin for each cell if only one batch is supplied in `...`. This will be ignored if two or more batches are supplied.
`restrict`	A list of length equal to the number of objects in `...`. Each entry of the list corresponds to one batch and specifies the cells to use when computing the correction.
`subset.row`	A vector specifying the subset of genes to use for correction. Defaults to `NULL`, in which case all genes are used.
`correct.all`	A logical scalar indicating whether to return corrected expression values for all genes, even if `subset.row` is set. Used to ensure that the output is of the same dimensionality as the input.
`assay.type`	A string or integer scalar specifying the assay to use for correction.

PARAM — A BatchelorParam object specifying the batch correction method to dispatch to, and the parameters with which it should be run. ClassicMnnParam will dispatch to mnnCorrect; FastMnnParam will dispatch to fastMNN; RescaleParam will dispatch to rescaleBatches; and RegressParam will dispatch to regressBatches.

combine.assays — Character vector specifying the assays from each entry of ... to combine together without correction. By default, any named assay that is present in all entries of ... is combined. This can be set to character(0) to avoid combining any assays.

combine.coldata — Character vector specifying the column metadata fields from each entry of ... to combine together. By default, any column metadata field that is present in all entries of ... is combined. This can be set to character(0) to avoid combining any metadata.

include.rowdata — Logical scalar indicating whether the function should attempt to include rowRanges.

add.single — Logical scalar indicating whether merged fields should be added to the original SingleCellExperiment. Only relevant when a single object is provided in .... If TRUE, combine.assays, combine.coldata and include.rowdata are ignored.

Details

This function makes it easy to retain information from the original SingleCellExperiment objects in the post-merge object. Operations like differential expression analyses can be easily performed on the uncorrected expression values, while common annotation can be leveraged in cell-based analyses like clustering.

- All assays shared across the original objects are cbinded and added to the merged object. This can be controlled with combine.assays. Any original assay with the same name as an assay in the output of batchCorrect will be ignored with a warning.
- Any column metadata fields that are shared will also be included in the merged object. This can be controlled with combine.coldata. If any existing field has the same name as any colData field produced by batchCorrect, it will be ignored in favor of the latter.
- Row metadata from ... is included in the merged object if include.rowdata=TRUE. In such cases, only non-conflicting row data fields are preserved, i.e., fields with different names or identically named fields with the same values between objects in .... Any conflicting fields are ignored with a warning. rowRanges are only preserved if they are identical (ignoring the mcols) for all objects in ....

If a single SingleCellExperiment object was supplied in ..., the default behavior is to prepend all assays, reducedDims, colData, rowData and metadata fields from the merged object into the original (removing any original entries with names that overlap those of the merged object). This is useful as it preserves all (non-overlapping) aspects of the original object, especially the reduced dimensions that cannot, in general, be sensibly combined across multiple objects. Setting add.single=FALSE will force the creation of a new SingleCellExperiment rather than prepending.

Value

A SingleCellExperiment containing the merged expression values in the first assay and a batch column metadata field specifying the batch of origin for each cell, as described in batchCorrect.

Author(s)

Aaron Lun

See Also

batchCorrect, which does the correction inside this function.

noCorrect, for another method to combine uncorrected assay values.

Examples

```
sce1 <- scuttle::mockSCE()
sce1 <- scuttle::logNormCounts(sce1)
sce2 <- scuttle::mockSCE()
sce2 <- scuttle::logNormCounts(sce2)

f.out <- correctExperiments(sce1, sce2)
colData(f.out)
assayNames(f.out)
```

---

cosineNorm *Cosine normalization*

Description

Perform cosine normalization on the column vectors of an expression matrix.

Usage

```
cosineNorm(
  x,
  mode = c("matrix", "all", "l2norm"),
  subset.row = NULL,
  BPPARAM = SerialParam()
)
```

Arguments

x	A gene expression matrix with cells as columns and genes as rows.
mode	A string specifying the output to be returned.
subset.row	A vector specifying which features to use to compute the L2 norm.
BPPARAM	A BiocParallelParam object specifying how parallelization is to be performed. Only used when x is a DelayedArray object.

Details

Cosine normalization removes scaling differences between expression vectors. In the context of batch correction, this is usually applied to remove differences between batches that are normalized separately. For example, `fastMNN` uses this function on the log-expression vectors by default.

Technically, separate normalization introduces scaling differences in the normalized expression, which should manifest as a shift in the log-transformed expression. However, in practice, single-cell data will contain many small counts (where the log function is near-linear) or many zeroes (which remain zero when the pseudo-count is 1). In these applications, scaling differences due to separate normalization are better represented as scaling differences in the log-transformed values.

If applied to the raw count vectors, cosine normalization is similar to library size-related (i.e., L1) normalization. However, we recommend using dedicated methods for computing size factors to normalize raw count data.

While the default is to directly return the cosine-normalized matrix, it may occasionally be desirable to obtain the L2 norm, e.g., to apply an equivalent normalization to other matrices. This can be achieved by setting `mode` accordingly.

The function will return a DelayedMatrix if `x` is a DelayedMatrix. This aims to delay the calculation of cosine-normalized values for very large matrices.

Value

If `mode="matrix"`, a double-precision matrix of the same dimensions as `X` is returned, containing cosine-normalized values.

If `mode="l2norm"`, a double-precision vector is returned containing the L2 norm for each cell.

If `mode="all"`, a named list is returned containing the fields `"matrix"` and `"l2norm"`, which are as described above.

Author(s)

Aaron Lun

See Also

`mnnCorrect` and `fastMNN`, where this function gets used.

Examples

```
A <- matrix(rnorm(1000), nrow=10)
str(cosineNorm(A))
str(cosineNorm(A, mode="l2norm"))
```

diagnostics-cluster *Cluster-based correction diagnostics*

Description

A variety of correction diagnostics that make use of clustering information, usually obtained by clustering on cells from all batches in the corrected data.

Usage

```
clusterAbundanceTest(x, batch)

clusterAbundanceVar(x, batch, pseudo.count = 10)
```

Arguments

x	A factor or vector specifying the assigned cluster for each cell in each batch in the corrected data. Alternatively, a matrix or table containing the number of cells in each cluster (row) and batch (column).
batch	A factor or vector specifying the batch of origin for each cell. Ignored if x is a matrix or table.
pseudo.count	A numeric scalar containing the pseudo-count to use for the log-transformation.

Details

For clusterAbundanceTest, the null hypothesis for each cluster is that the distribution of cells across batches is proportional to the total number of cells in each batch. We then use chisq.test to test for deviations from the expected proportions, possibly indicative of imperfect mixing across batches. This works best for technical replicates where the population composition should be identical across batches. However, the interpretation of the p-value loses its meaning for experiments where there is more biological variability between batches.

For clusterAbundanceVar, we compute log-normalized abundances for each cluster using normalizeCounts. We then compute the variance of the log-abundances across batches for each cluster. Large variances indicate that there are strong relative differences in abundance across batches, indicative of either imperfect mixing or genuine batch-specific subpopulations. The idea is to rank clusters by their variance to prioritize them for manual inspection to decide between these two possibilities. We use a large pseudo.count by default to avoid spuriously large variances when the counts are low.

Value

For clusterAbundanceTest, a named numeric vector of p-values from applying Pearson's chi-squared test on each cluster.

For clusterAbundanceVar, a named numeric vector of variances of log-abundances across batches for each cluster.

Author(s)

Aaron Lun

Examples

```
set.seed(1000)
means <- 2^rgamma(1000, 2, 1)
A1 <- matrix(rpois(10000, lambda=means), ncol=50) # Batch 1
A2 <- matrix(rpois(10000, lambda=means*runif(1000, 0, 2)), ncol=50) # Batch 2

B1 <- log2(A1 + 1)
B2 <- log2(A2 + 1)
out <- fastMNN(B1, B2)

cluster1 <- kmeans(t(B1), centers=10)$cluster
cluster2 <- kmeans(t(B2), centers=10)$cluster
merged.cluster <- kmeans(reducedDim(out, "corrected"), centers=10)$cluster

# Low p-values indicate unexpected differences in abundance.
clusterAbundanceTest(paste("Cluster", merged.cluster), out$batch)

# High variances indicate differences in normalized abundance.
clusterAbundanceVar(paste("Cluster", merged.cluster), out$batch)
```

---

divideIntoBatches *Divide into batches*

Description

Divide a single input object into multiple separate objects according to their batch of origin.

Usage

```
divideIntoBatches(x, batch, byrow = FALSE, restrict = NULL)
```

Arguments

x	A matrix-like object where one dimension corresponds to cells and another represents features.
batch	A factor specifying the batch to which each cell belongs.
byrow	A logical scalar indicating whether rows correspond to cells.
restrict	A subsetting vector specifying which cells should be used for correction.

Details

This function is intended for internal use and other package developers. It splits a single input object into multiple batches, allowing developers to use the same code for the scenario where batch is supplied with a single input.

Value

A list containing:

- batches, a named list of matrix-like objects where each element corresponds to a level of batch and contains all cells from that batch.
- reorder, an integer vector to be applied to the combined batches to recover the ordering of cells in x.
- restricted, a named list of integer vectors specifying which cells are to be used for correction. Set to NULL if the input restrict was also NULL.

Author(s)

Aaron Lun

Examples

```
X <- matrix(rnorm(1000), ncol=100)
out <- divideIntoBatches(X, sample(3, 100, replace=TRUE))
names(out)

# Recovering original order.
Y <- do.call(cbind, out$batches)
Z <- Y[,out$reorder]
all.equal(Z, X) # should be TRUE.
```

---

fastMNN *Fast mutual nearest neighbors correction*

Description

Correct for batch effects in single-cell expression data using a fast version of the mutual nearest neighbors (MNN) method.

Usage

```
fastMNN(
  ...,
  batch = NULL,
  k = 20,
  prop.k = NULL,
  restrict = NULL,
  cos.norm = TRUE,
  ndist = 3,
  d = 50,
  deferred = TRUE,
  weights = NULL,
```

```
  get.variance = FALSE,
  merge.order = NULL,
  auto.merge = FALSE,
  min.batch.skip = 0,
  subset.row = NULL,
  correct.all = FALSE,
  assay.type = "logcounts",
  BSPARAM = IrlbaParam(),
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam()
)
```

Arguments

`...`	One or more log-expression matrices where genes correspond to rows and cells correspond to columns. Alternatively, one or more SingleCellExperiment objects can be supplied containing a log-expression matrix in the `assay.type` assay. Each object should contain the same number of rows, corresponding to the same genes in the same order. Objects of different types can be mixed together. If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. If a single object is supplied, it is assumed to contain cells from all batches, so `batch` should also be specified. Alternatively, one or more lists of matrices or SingleCellExperiments can be provided; this is flattened as if the objects inside each list were passed directly to `...`.
`batch`	A vector or factor specifying the batch of origin for all cells when only a single object is supplied in `...`. This is ignored if multiple objects are present.
`k`	An integer scalar specifying the number of nearest neighbors to consider when identifying MNNs.
`prop.k`	A numeric scalar in (0, 1) specifying the proportion of cells in each dataset to use for mutual nearest neighbor searching. If set, the number of nearest neighbors used for the MNN search in each batch is redefined as `max(k, prop.k*N)` where `N` is the number of cells in that batch.
`restrict`	A list of length equal to the number of objects in `...`. Each entry of the list corresponds to one batch and specifies the cells to use when computing the correction.
`cos.norm`	A logical scalar indicating whether cosine normalization should be performed on the input data prior to PCA.
`ndist`	A numeric scalar specifying the threshold beyond which neighbours are to be ignored when computing correction vectors. Each threshold is defined as a multiple of the number of median distances.
`d`	Numeric scalar specifying the number of dimensions to use for dimensionality reduction in `multiBatchPCA`. If `NA`, no dimensionality reduction is performed and any matrices in `...` are used as-is.
`deferred`	Logical scalar indicating whether to defer centering/scaling, see `multiBatchPCA` for details.

weights, get.variance
: Further arguments to pass to multiBatchPCA.

merge.order
: An integer vector containing the linear merge order of batches in .... Alternatively, a list of lists representing a tree structure specifying a hierarchical merge order.

auto.merge
: Logical scalar indicating whether to automatically identify the "best" merge order.

min.batch.skip
: Numeric scalar specifying the minimum relative magnitude of the batch effect, below which no correction will be performed at a given merge step.

subset.row
: A vector specifying which features to use for correction.

correct.all
: Logical scalar indicating whether corrected expression values should be computed for genes not in subset.row. Only relevant if subset.row is not NULL.

assay.type
: A string or integer scalar specifying the assay containing the log-expression values. Only used for SingleCellExperiment inputs.

BSPARAM
: A BiocSingularParam object specifying the algorithm to use for PCA in multiBatchPCA.

BNPARAM
: A BiocNeighborParam object specifying the nearest neighbor algorithm.

BPPARAM
: A BiocParallelParam object specifying whether the PCA and nearest-neighbor searches should be parallelized.

Details

This function provides a variant of the mnnCorrect function, modified for speed and more robust performance. In particular:

- It performs a multi-sample PCA via multiBatchPCA and subsequently performs all calculations in the PC space. This reduces computational work and provides some denoising for improved neighbour detection. As a result, though, the corrected output cannot be interpreted on a gene level and is useful only for cell-level comparisons, e.g., clustering and visualization.
- The correction vector for each cell is directly computed from its k nearest neighbours in the same batch. Specifically, only the k nearest neighbouring cells that *also* participate in MNN pairs are used. Each MNN-participating neighbour is weighted by distance from the current cell, using a tricube scheme with bandwidth equal to the median distance multiplied by ndist. This ensures that the correction vector only uses information from the closest cells, improving the fidelity of local correction.
- Issues with "kissing" are avoided with a two-step procedure that removes variation along the batch effect vector. First, the average correction vector across all MNN pairs is computed. Cell coordinates are adjusted such that all cells in a single batch have the same position along this vector. The correction vectors are then recalculated with the adjusted coordinates (but the same MNN pairs).

The default setting of cos.norm=TRUE provides some protection against differences in scaling between log-expression matrices from batches that are normalized separately (see cosineNorm for details). However, if possible, we recommend using the output of multiBatchNorm as input to fastMNN. This will equalize coverage on the count level before the log-transformation, which is a more accurate rescaling than cosine normalization on the log-values.

The batch argument allows users to easily perform batch correction when all cells have already been combined into a single object. This avoids the need to manually split the matrix or SingleCellExperiment object into separate objects for input into fastMNN. In this situation, the order of input batches is defined by the order of levels in batch.

Value

A SingleCellExperiment is returned where each row is a gene and each column is a cell. This contains:

- A corrected matrix in the reducedDims slot, containing corrected low-dimensional coordinates for each cell. This has number of columns equal to d and number of rows equal to the total number of cells in ....
- A batch column in the colData slot, containing the batch of origin for each row (i.e., cell) in corrected.
- A rotation column the rowData slot, containing the rotation matrix used for the PCA. This has d columns and number of rows equal to the number of genes to report (see the "Choice of genes" section).
- A reconstructed matrix in the assays slot, containing the low-rank reconstruction of the expression matrix. This can be interpreted as per-gene corrected log-expression values (after cosine normalization, if cos.norm=TRUE) but should not be used for quantitative analyses. This has number of rows equal to the number of input genes if subset.row=NULL or correct.all=TRUE, otherwise each row corresponds to a gene in subset.row.

Cells in the output object are always ordered in the same manner as supplied in .... For a single input object, cells will be reported in the same order as they are arranged in that object. In cases with multiple input objects, the cell identities are simply concatenated from successive objects, i.e., all cells from the first object (in their provided order), then all cells from the second object, and so on. This is true regardless of the value of merge.order and auto.merge, which only affects the internal merge order of the batches.

Additionally, the metadata of the output object contains:

- merge.info, a DataFrame of diagnostic information about each merge step. See the "Merge diagnostics" section for more details.
- pca.info, a list of metadata produced by multiBatchPCA, such as the variance explained when get.variance=TRUE.

Controlling the merge order

By default, batches are merged in the user-supplied order in ..., i.e., the first batch is merged with the second batch, the third batch is merged with the combined first-second batch, the fourth batch is merged with the combined first-second-third batch and so on. We refer to this approach as a progressive merge. When batch is supplied for a single object in ..., the ordering of batches in a progressive merge is determined by the ordering of factor levels in batch.

If merge.order is an integer vector, it is treated as an ordering permutation with which to perform a progressive merge. For example, if merge.order=c(4,1,3,2), batches 4 and 1 in ... are merged first; batch 3 is merged with the combined 4+1 batch; and then batch 2 is merged with the combined

4+1+3 batch. This is often more convenient than changing the order manually in ..., which would alter the order of batches in the output `corrected` matrix.

If `merge.order` is a character vector, it is treated as an ordering permutation for named batches. If `merge.order` is a factor, it is coerced into a character vector and also treated as a permutation of names.

If `merge.order` is a nested list, it is treated as a tree that specifies a hierarchical merge. Each element of the list should either be a string or integer scalar, corresponding to a leaf node that specifies a batch; or another list, corresponding to an internal node that should contain at least two children; or an integer or character vector of length 2 or more, again corresponding to an internal node.

- For example, `list(list(1,2), list(3,4))` indicates that batch 1 should be merged with batch 2; batch 3 should be merged with batch 4; and that, finally, the combined batches 1+2 and 3+4 should be merged.
- More than two children per node are supported and will result in a progressive merge within that node. For example, `list(list(1,2,3), list(4,5,6))` will merge batch 1 with 2, then 1+2 with 3; batch 4 with 5, and then 4+5 with 6; and finally, 1+2+3 with 4+5+6.
- The same approach can be used for integer or character vectors, e.g., `list(1:3, 4:6)` has the same effect as above.

Note that, while batches can be specified by name (character) or index (integer), users cannot use both in the same tree. Factors can be provided and are coerced to character vectors.

The merge order may occasionally be important as it determines the number of MNN pairs available at each merge step. MNN pairs results in greater stability of the batch vectors and increased likelihood of identifying shared subpopulations, which are important to the precision and accuracy of the MNN-based correction, respectively.

- In a progressive merge, the reference increases in size at each step, ensuring that more cells are available to identify MNN pairs in later merges. We suggest setting the largest, most heterogeneous batch as the first reference, which favors detection of sufficient MNN pairs between the first and other batches. Conversely, if two small batches without shared populations are supplied first, the wrong MNN pairs will be detected and the result of the merge will be incorrect.
- A merge tree is useful for merging together batches that are known to be more closely related (e.g., replicates) before attempting difficult merges involving more dissimilar batches. The idea is to increase the number of cells and thus MNN pairs prior to merging batches with few shared subpopulations. By comparison, performing the more difficult merges first is more likely to introduce errors whereby distinct subpopulations are incorrectly placed together, which is propagated to later steps as the initial merge is used as a reference for subsequent merges.
- If `auto.merge=TRUE`, merge steps are chosen to maximize the number of MNN pairs at each step. The aim is to improve the stability of the correction by first merging more similar batches with more MNN pairs. This can be somewhat time-consuming as MNN pairs need to be iteratively recomputed for all possible batch pairings.

The order of cells in the output is *never* affected by the setting of `merge.order`. It depends only on the order of objects in ... and the order of cells within each object.

Choice of genes

All genes are used with the default setting of `subset.row=NULL`. Users can set `subset.row` to subset the inputs to highly variable genes or marker genes. This improves the quality of the PCA and identification of MNN pairs by reducing the noise from irrelevant genes. Users should not be too restrictive with subsetting, as high dimensionality is required to satisfy the orthogonality assumption in MNN detection.

By default, only the selected genes are used to compute rotation vectors and a low-rank corrected expression matrix. However, setting `correct.all=TRUE` will return rotation vectors spanning all genes in the supplied input data. This ensures that corrected values are returned in `reconstructed` for all input genes, e.g., in `correctExperiments`. Note that this setting will not alter the corrected low-dimension coordinates, nor the rotation values for the selected genes.

If `d=NA`, any gene *not* in `subset.row` will have reconstructed values of zero when `correct.all=TRUE`. Without a PCA, we cannot easily extrapolate the correction to other genes.

Using restriction

See `?"batchelor-restrict"` for a description of the `restrict` argument. Specifically, `fastMNN` will only use the restricted subset of cells in each batch to identify MNN pairs and the center of the orthogonalization. It will then extrapolate the correction to all cells in each batch.

Note that *all* cells are used to perform the PCA, regardless of whether `restrict` is set. This is generally desirable in applications where `restrict` is useful. For example, constructing the projection vectors with only control cells will not guarantee resolution of unique non-control populations in each batch.

However, this also means that the corrected values for the restricted cells will differ from the output when the inputs are directly subsetted to only contain the restricted cells. If this is not desirable, users can perform the PCA manually and apply `reducedMNN` instead.

Merge diagnostics

We can consider `fastMNN`'s operation in terms of pairwise merge steps. Each merge step involves two mutually exclusive sets of cells, a "left" set and "right" set. Each set may consist of cells from different batches if those batches were merged in a previous step. The merge will then create a new set of cells that combines the left and right sets. Iteratively repeating this process with the newly formed sets will eventually merge all batches together.

The output metadata contains `merge.info`, a DataFrame where each row corresponds to a merge step. It contains the following fields:

- `left`, a List of integer or character vectors. Each vector specifies the batches in the left set at a given merge step.
- `right`, a similar List of integer or character vectors. Each vector specifies the batches in the right set at a given merge step.
- `pairs`, a List of DataFrames specifying which pairs of cells were identified as MNNs at each step. In each DataFrame, each row corresponds to a single MNN pair and specifies the paired cells that were in the left and right sets, respectively. Note that the indices refer to those paired cells in the *output* ordering of cells, i.e., users can identify the paired cells at each step by column-indexing the output of the `fastMNN` function.

- batch.size, a numeric vector specifying the relative magnitude of the batch effect at each merge, see "Orthogonalization details".
- skipped, a logical vector indicating whether the correction was skipped if the magnitude of the batch effect was below min.batch.skip.
- lost.var, a numeric matrix specifying the percentage of variance lost due to orthogonalization at each merge step. This is reported separately for each batch (columns, ordered according to the input order, *not* the merge order).

Specifying the number of neighbors

The threshold to define nearest neighbors is defined by k, which is passed to findMutualNN to identify MNN pairs. The size of k can be roughly interpreted as the anticipated minimum size of a shared subpopulation in each batch. If a batch has fewer than k cells of a shared subpopulation, there is an increased risk that its counterparts in other batches will form incorrect MNN pairs.

From the perspective of the algorithm, larger values allow for more MNN pairs to be obtained, which improves the stability of the correction vectors. Larger values also increase robustness against non-orthogonality, by ignoring a certain level of biological variation when identifying pairs. This can be used to avoid the kissing problem where MNN pairs are only detected on the "surface" of the distribution. However, values of k should not be too large, as this would result in MNN pairs being inappropriately identified between biologically distinct populations.

In practice, increasing k will generally result in more aggressive merging as the algorithm is more generous in matching subpopulations across batches. We suggest starting with the default k and increasing it if one is confident that the same cell types are not adequately merged across batches. This is better than starting with a large k as incorrect merging is much harder to diagnose than insufficient merging.

An additional consideration is that the effect of any given k will vary with the number of cells in each batch. With more cells, a larger k may be preferable to achieve better merging in the presence of non-orthogonality. We can achieve this by setting prop.k, e.g., prop.k=0.05 will set k to 5% of the number of cells in each batch. This allows the choice of k to adapt to the size of each batch at each merge step and handles asymmetry in batch sizes (via the k1 and k2 arguments in findMutualNN).

Orthogonalization details

fastMNN will compute the percentage of variance that is lost from each batch during orthogonalization at each merge step. This represents the variance in each batch that is parallel to the average correction vectors (and hence removed during orthogonalization) at each merge step. Large proportions suggest that there is biological structure that is parallel to the batch effect, corresponding to violations of the assumption that the batch effect is orthogonal to the biological subspace. A rule of thumb is that more than 10% of lost variance is cause for closer examination, though this is highly dependent on the context, e.g., a large lost proportion may be fine if the population structure is still approximately preserved while a small lost proportion may be unacceptable if an important rare subpopulation can no longer be distinguished.

In rare cases, orthogonalization may cause problems in the *absence* of a batch effect, resulting in large losses of variance. To avoid this, fastMNN will not perform any correction if the relative magnitude of the batch effect is less than min.batch.skip. The relative magnitude is defined as the L2 norm of the average correction vector divided by the root-mean-square of the L2 norms of

the per-MNN pair correction vectors. This will be large when the per-pair vectors are all pointing in the same direction, and small when the per-pair vectors point in random directions due to the absence of a consistent batch effect. If a large loss of variance is observed along with a small batch effect in a given merge step, users can set `min.batch.skip` to simply skip correction in that step.

Author(s)

Aaron Lun

References

Haghverdi L, Lun ATL, Morgan MD, Marioni JC (2018). Batch effects in single-cell RNA-sequencing data are corrected by matching mutual nearest neighbors. *Nat. Biotechnol.* 36(5):421

Lun ATL (2018). Further MNN algorithm development. https://MarioniLab.github.io/FurtherMNN2018/theory/description.html

See Also

`cosineNorm` and `multiBatchPCA`, to obtain the values to be corrected.

`reducedMNN`, for a version of the function that operates in low-dimensional space.

`mnnCorrect`, for the "classic" version of the MNN correction algorithm.

`clusterMNN`, for the cluster-based version of this approach.

`multiModalMNN` from the **mumosa** package, which extends this to data in the `altExps`.

`mnnDeltaVariance`, to obtain further per-gene diagnostics on the behavior of the correction.

Examples

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

# Corrected values for use in clustering, etc.
str(reducedDim(out))

# Extracting corrected expression values for gene 10.
summary(assay(out)[10,])
```

`intersectRows` *Take the intersection of rows across batches*

Description

Subset multiple batches so that they have the same number and order of rows.

Usage

```
intersectRows(..., subset.row = NULL, keep.all = FALSE)
```

Arguments

...	One or more matrix-like objects containing single-cell gene expression matrices. Alternatively, one or more SingleCellExperiment objects can be supplied.
subset.row	A vector specifying the subset of genes to retain. Defaults to NULL, in which case all genes are retained.
keep.all	Logical scalar indicating whether the data should actually be subsetted to subset.row. If FALSE, subset.row is checked but not used.

Details

If any entry of ... contains no row names or if the intersection is empty, an error is raised.

If all entries of ... already have the same row names in the same order, this function is a no-op. In such cases, subset.row can be a character, integer or logical vector.

If entries of ... do not have identical row names, subset.row can only be a character vector. Any other type will lead to an error as the interpretation of the subset is not well defined.

Setting keep.all=TRUE option gives the same result as having subset.row=NULL in the first place. However, it is still useful for checking that subset.row is a character vector. This ensures that downstream applications can safely use subset.row in conjunction with, e.g., correct.all=TRUE for fastMNN.

Value

A list containing the row-subsetted contents of ..., each of which have the same number and order of rows.

Author(s)

Aaron Lun

Examples

```
X <- rbind(A=c(1,2), B=c(3,4))
Y <- rbind(a=c(1,2), B=c(3,4))
intersectRows(X, Y) # Only B is retained.

# Error is raised when no genes are retained:
X <- rbind(A=c(1,2), B=c(3,4))
Y <- rbind(a=c(1,2), b=c(3,4))
try(intersectRows(X, Y))

# Error is raised for non-character subset.row
# when row names are not identical:
X <- rbind(A=c(1,2), B=c(3,4), C=c(5,6))
Y <- rbind(a=c(1,2), B=c(3,4), C=c(5,6))
intersectRows(X, Y)
```

```
intersectRows(X, Y, subset.row="B")
try(intersectRows(X, Y, subset.row=1))

# Setting keep.all=TRUE only checks but does not apply subset.row.
intersectRows(X, Y, subset.row="B", keep.all=TRUE)
try(intersectRows(X, Y, subset.row=1, keep.all=TRUE))
```

`mnnCorrect` *Mutual nearest neighbors correction*

Description

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

Usage

```
mnnCorrect(
  ...,
  batch = NULL,
  restrict = NULL,
  k = 20,
  prop.k = NULL,
  sigma = 0.1,
  cos.norm.in = TRUE,
  cos.norm.out = TRUE,
  svd.dim = 0L,
  var.adj = TRUE,
  subset.row = NULL,
  correct.all = FALSE,
  merge.order = NULL,
  auto.merge = FALSE,
  assay.type = "logcounts",
  BSPARAM = ExactParam(),
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam()
)
```

Arguments

`...`	One or more log-expression matrices where genes correspond to rows and cells correspond to columns. Alternatively, one or more SingleCellExperiment objects can be supplied containing a log-expression matrix in the `assay.type` assay. Each object should contain the same number of rows, corresponding to the same genes in the same order. Objects of different types can be mixed together. If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. If a single object is supplied, it is assumed to contain cells from all batches, so `batch` should also be specified.

	Alternatively, one or more lists of matrices or SingleCellExperiments can be provided; this is flattened as if the objects inside each list were passed directly to ...
batch	A vector or factor specifying the batch of origin for all cells when only a single object is supplied in ... This is ignored if multiple objects are present.
restrict	A list of length equal to the number of objects in ... Each entry of the list corresponds to one batch and specifies the cells to use when computing the correction.
k	An integer scalar specifying the number of nearest neighbors to consider when identifying MNNs.
prop.k	A numeric scalar in (0, 1) specifying the proportion of cells in each dataset to use for mutual nearest neighbor searching. If set, the number of nearest neighbors used for the MNN search in each batch is redefined as max(k, prop.k*N) where N is the number of cells in that batch.
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.
subset.row	A vector specifying which features to use for correction.
correct.all	A logical scalar specifying whether correction should be applied to all genes, even if only a subset is used for the MNN calculations.
merge.order	An integer vector containing the linear merge order of batches in ... Alternatively, a list of lists representing a tree structure specifying a hierarchical merge order.
auto.merge	Logical scalar indicating whether to automatically identify the "best" merge order.
assay.type	A string or integer scalar specifying the assay containing the log-expression values. Only used for SingleCellExperiment inputs.
BSPARAM	A BiocSingularParam object specifying the algorithm to use for PCA in multiBatchPCA.
BNPARAM	A BiocNeighborParam object specifying the nearest neighbor algorithm.
BPPARAM	A BiocParallelParam object specifying whether the PCA and 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 log-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 (log-)expression values are returned for use in clustering and dimensionality reduction.

For each MNN pair, a pairwise correction vector is computed based on the difference in the log-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 SingleCellExperiment object containing the `corrected` assay. This contains corrected expression values for each gene (row) in each cell (column) in each batch. A `batch` field is present in the column data, specifying the batch of origin for each cell.

Cells in the output object are always ordered in the same manner as supplied in `...`. For a single input object, cells will be reported in the same order as they are arranged in that object. In cases with multiple input objects, the cell identities are simply concatenated from successive objects, i.e., all cells from the first object (in their provided order), then all cells from the second object, and so on.

The metadata of the SingleCellExperiment contains `merge.info`, a DataFrame where each row corresponds to a merge step. See "Merge diagnostics" for more information.

Choosing the gene set

All genes are used with the default setting of `subset.row=NULL`. Users can set `subset.row` to subset the inputs to highly variable genes or marker genes. This may provide more meaningful identification of MNN pairs by reducing the noise from irrelevant genes. Note that users should not be too restrictive with subsetting, as high dimensionality is required to satisfy the orthogonality assumption in MNN detection.

If `subset.row` is specified and `correct.all=TRUE`, 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 for all genes in the supplied matrices. Note that setting `correct.all=TRUE` will not alter the corrected expression values for the subsetted genes.

Expected type of input data

The input expression values should generally be log-transformed, e.g., log-counts, see `logNormCounts` for details. They should also be normalized within each data set to remove cell-specific biases in capture efficiency and sequencing depth. Users may also consider using the `multiBatchNorm` functino to mitigate the effect of differences in sequencing depth between batches.

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.

The cosine normalization is achieved using the cosineNorm function.

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 to obtain svd.dim dimensions. (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.

See ?"batchelor-restrict" for a description of the restrict argument. Specifically, mnnCorrect will only use the restricted subset of cells in each batch to identify MNN pairs (and to perform variance adjustment, if var.adj=TRUE), and then apply the correction to all cells in each batch.

Merge diagnostics

Each merge step combines two mutually exclusive sets of cells, a "left" set and "right" set. The metadata thus contains the following fields:

- left, a List of integer or character vectors. Each vector specifies the batches in the left set at a given merge step.
- right, a similar List of integer or character vectors. Each vector specifies the batches in the right set at a given merge step.
- pairs, a List of DataFrames specifying which pairs of cells were identified as MNNs at each step. In each DataFrame, each row corresponds to a single MNN pair and specifies the paired cells that were in the left and right sets, respectively. Note that the indices refer to those paired cells in the *output* ordering of cells, i.e., users can identify the paired cells at each step by column-indexing the output of the mnnCorrect function.

Specifying the number of neighbors

The threshold to define nearest neighbors is defined by k, which is passed to findMutualNN to identify MNN pairs. The size of k can be roughly interpreted as the anticipated minimum size of

a shared subpopulation in each batch. If a batch has fewer than k cells of a shared subpopulation, there is an increased risk that its counterparts in other batches will form incorrect MNN pairs.

From the perspective of the algorithm, larger values allow for more MNN pairs to be obtained, which improves the stability of the correction vectors. Larger values also increase robustness against non-orthogonality, by ignoring a certain level of biological variation when identifying pairs. This can be used to avoid the kissing problem where MNN pairs are only detected on the "surface" of the distribution. However, values of k should not be too large, as this would result in MNN pairs being inappropriately identified between biologically distinct populations.

In practice, increasing k will generally result in more aggressive merging as the algorithm is more generous in matching subpopulations across batches. We suggest starting with the default k and increasing it if one is confident that the same cell types are not adequately merged across batches. This is better than starting with a large k as incorrect merging is much harder to diagnose than insufficient merging.

An additional consideration is that the effect of any given k will vary with the number of cells in each batch. With more cells, a larger k may be preferable to achieve better merging in the presence of non-orthogonality. We can achieve this by setting prop.k, e.g., prop.k=0.05 will set k to 5% of the number of cells in each batch. This allows the choice of k to adapt to the size of each batch at each merge step and handles asymmetry in batch sizes (via the k1 and k2 arguments in findMutualNN).

Controlling the merge order

By default, batches are merged in the user-supplied order in ..., i.e., the first batch is merged with the second batch, the third batch is merged with the combined first-second batch, the fourth batch is merged with the combined first-second-third batch and so on. We refer to this approach as a progressive merge. When batch is supplied for a single object in ..., the ordering of batches in a progressive merge is determined by the ordering of factor levels in batch.

If merge.order is an integer vector, it is treated as an ordering permutation with which to perform a progressive merge. For example, if merge.order=c(4,1,3,2), batches 4 and 1 in ... are merged first; batch 3 is merged with the combined 4+1 batch; and then batch 2 is merged with the combined 4+1+3 batch. This is often more convenient than changing the order manually in ..., which would alter the order of batches in the output corrected matrix.

If merge.order is a character vector, it is treated as an ordering permutation for named batches. If merge.order is a factor, it is coerced into a character vector and also treated as a permutation of names.

If merge.order is a nested list, it is treated as a tree that specifies a hierarchical merge. Each element of the list should either be a string or integer scalar, corresponding to a leaf node that specifies a batch; or another list, corresponding to an internal node that should contain at least two children; or an integer or character vector of length 2 or more, again corresponding to an internal node.

- For example, list(list(1,2), list(3,4)) indicates that batch 1 should be merged with batch 2; batch 3 should be merged with batch 4; and that, finally, the combined batches 1+2 and 3+4 should be merged.
- More than two children per node are supported and will result in a progressive merge within that node. For example, list(list(1,2,3), list(4,5,6)) will merge batch 1 with 2, then 1+2 with 3; batch 4 with 5, and then 4+5 with 6; and finally, 1+2+3 with 4+5+6.

- The same approach can be used for integer or character vectors, e.g., `list(1:3, 4:6)` has the same effect as above.

Note that, while batches can be specified by name (character) or index (integer), users cannot use both in the same tree. Factors can be provided and are coerced to character vectors.

The merge order may occasionally be important as it determines the number of MNN pairs available at each merge step. MNN pairs results in greater stability of the batch vectors and increased likelihood of identifying shared subpopulations, which are important to the precision and accuracy of the MNN-based correction, respectively.

- In a progressive merge, the reference increases in size at each step, ensuring that more cells are available to identify MNN pairs in later merges. We suggest setting the largest, most heterogeneous batch as the first reference, which favors detection of sufficient MNN pairs between the first and other batches. Conversely, if two small batches without shared populations are supplied first, the wrong MNN pairs will be detected and the result of the merge will be incorrect.
- A merge tree is useful for merging together batches that are known to be more closely related (e.g., replicates) before attempting difficult merges involving more dissimilar batches. The idea is to increase the number of cells and thus MNN pairs prior to merging batches with few shared subpopulations. By comparison, performing the more difficult merges first is more likely to introduce errors whereby distinct subpopulations are incorrectly placed together, which is propagated to later steps as the initial merge is used as a reference for subsequent merges.
- If `auto.merge=TRUE`, merge steps are chosen to maximize the number of MNN pairs at each step. The aim is to improve the stability of the correction by first merging more similar batches with more MNN pairs. This can be somewhat time-consuming as MNN pairs need to be iteratively recomputed for all possible batch pairings.

The order of cells in the output is *never* affected by the setting of `merge.order`. It depends only on the order of objects in `...` and the order of cells within each object.

Author(s)

Laleh Haghverdi, with modifications by Aaron Lun

References

Haghverdi L, Lun ATL, Morgan MD, Marioni JC (2018). Batch effects in single-cell RNA-sequencing data are corrected by matching mutual nearest neighbors. *Nat. Biotechnol.* 36(5):421

See Also

`fastMNN` for a faster equivalent.

Examples

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

`mnnDeltaVariance` *Computes the variance of the paired MNN deltas*

Description

Computes the variance of the paired MNN deltas

Usage

```
mnnDeltaVariance(
  ...,
  pairs,
  cos.norm = FALSE,
  subset.row = NULL,
  compute.all = FALSE,
  assay.type = "logcounts",
  BPPARAM = SerialParam(),
  trend.args = list()
)
```

Arguments

`...`	One or more log-expression matrices where genes correspond to rows and cells correspond to columns. Alternatively, one or more SingleCellExperiment objects can be supplied containing a log-expression matrix in the `assay.type` assay. Each object should contain the same number of rows, corresponding to the same genes in the same order. Objects of different types can be mixed together. If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. If a single object is supplied, it is assumed to contain cells from all batches, so `batch` should also be specified. Alternatively, one or more lists of matrices or SingleCellExperiments can be provided; this is flattened as if the objects inside each list were passed directly to `...`.
`pairs`	A DataFrame or list of DataFrames containing MNN pairing information. Each row of each DataFrame specifies an MNN pair; each DataFrame should have two columns containing the column indices of paired cells. This is typically produced by `fastMNN`, see that documentation for more information.
`cos.norm`	A logical scalar indicating whether cosine normalization should be performed on the expression values.
`subset.row`	A vector specifying which genes to use for the calculation. This is typically the same as the `subset.row` passed to `fastMNN`, if any.
`compute.all`	Logical scalar indicating whether statistics should be returned for all genes when `subset.row` is specified, see DEtails.
`assay.type`	A string or integer scalar specifying the assay containing the log-expression values. Only used for SingleCellExperiment inputs.

BPPARAM	A BiocParallelParam object specifying whether the calculations should be parallelized.
trend.args	Named list of further arguments to pass to fitTrendVar from the **scran** package.

Details

The "MNN delta" is defined as the difference in the *uncorrected* expression values of MNN-paired cells. We compute the variance of these values across all pairs for each gene; genes with highly variable deltas are strongly affected by the correction beyond a simple shift. By looking through the top genes, we may conclude that the correction did something unusual if, e.g., we find important markers in the top set. Conversely, if key genes do not have unusually high variances in their delta, we may gain some confidence in the reliability of the correction - at least with respect to the behavior of those genes.

We compute the variance of the deltas rather than using their magnitude, as a shift in expression space is not particularly concerning (and is, in fact, par for the course for batch correction). Rather, we are interested in identifying non-linear changes that might be indicative of correction errors, e.g., inappropriate merging of two different subpopulations. This would manifest as high variances for the relevant marker genes. Of course, whether or not this is actually an error can be a subjective decision - loss of some biological heterogeneity is often an acceptable cost for flexible correction.

To eliminate the mean-variance relationship, we fit a trend to the variances across all genes with respect to the mean log-expression in each MNN pair. We then define an adjusted variance based on the residuals of the fitted curve; this is a more appropriate measure to use for ranking affected genes, as it ensures that genes are not highly ranked due to their abundance alone. Fitting is done using the fitTrendVar function from the **scran** package - further arguments can be passed via trend.args.

If a list is passed to pairs, each entry of the list is assumed to correspond to a merge step. The variance and trend fitting is done separately for each merge step, and the results are combined by computing the average variance across all steps. This avoids considering the variance of the deltas across merge steps, which is not particularly concerning, e.g., if different batches require different translations.

Value

A DataFrame with one row per gene in ... (or as specified by subset.row), containing the following columns:

- mean, the mean of the mean log-expressions across all MNN pairs. This may contain repeated contributions from the same cell if it is involved in many MNN pairs.
- total, the total variance of the deltas across all MNN pairs.
- trend, the fitted values of the trend in total with respect to mean.
- adjusted, the adjusted variance, i.e., the residuals of the fitted trend.

The metadata contains the trend fitting statistics returned by fitTrendVar.

If pairs is a list of length greater than 1, the returned DataFrame will also contain per.block, a nested DataFrames of nested DataFrames. Each one of these contains statistics for the individual merge steps and has the same structure as that described above.

Improving consistency with fastMNN

If cos.norm=TRUE, cosine normalization is performed on each cell in ... with cosineNorm. This mimics what is done inside fastMNN for greater consistency with that function's internal calculations. Some further scaling is performed so that the magnitude of the normalized values is mostly unaffected.

If subset.row is specified, variances are only computed for the requested subset of genes, most typically the set of highly variable genes used in fastMNN. This also implies that the normalization and trend fitting is limited to the specified subset. However, if compute.all=TRUE, the scaling factor and fitted trend are extrapolated to compute adjusted variances for all other genes. This is useful for picking up genes outside of the subset used in the correction.

In most applications, it is not necessary to have strict consistency with the internal values computed by fastMNN. Nonetheless, it may be helpful for diagnostics in more difficult situations.

Author(s)

Aaron Lun

See Also

fastMNN and related functions, to obtain the MNN pairs in the first place.

Examples

```
means <- 2^rexp(200) * 10
B1 <- matrix(rpois(10000, means), ncol=50) # Batch 1
B2 <- matrix(rpois(10000, means), ncol=50) # Batch 2

# Spiking in some population structure. Each batch is split into two halves
# corresponding to 2 subpopulations. These populations match up across batches
# but the group in batch 1 also upregulates the first gene.
B1[1:2,1:25] <- B1[1:2,1:25] * 50
B2[2,1:25] <- B2[2,1:25] * 50

# Quick log-transformation and correction. We'll omit the multiBatchNorm as the
# sequencing depth is the same across batches anyway.
library(scuttle)
B1 <- normalizeCounts(B1)
B2 <- normalizeCounts(B2)
out <- fastMNN(B1, B2)

# First gene should have high variance of deltas, as the upregulation in B1 is
# removed to enable the alignment of subpopulations across batches.
mnnDeltaVariance(B1, B2, pairs=metadata(out)$merge.info$pairs[[1]])
```

`multiBatchNorm` *Per-batch scaling normalization*

Description

Perform scaling normalization within each batch to provide comparable results to the lowest-coverage batch.

Usage

```
multiBatchNorm(
  ...,
  batch = NULL,
  norm.args = list(),
  min.mean = 1,
  subset.row = NULL,
  normalize.all = FALSE,
  preserve.single = TRUE,
  assay.type = "counts",
  BPPARAM = SerialParam()
)
```

Arguments

`...`	One or more SingleCellExperiment objects containing counts and size factors. Each object should contain the same number of rows, corresponding to the same genes in the same order. If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. If a single object is supplied, `batch` should also be specified. Alternatively, one or more lists of SingleCellExperiments can be provided; this is flattened as if the objects inside were passed directly to `...`.
`batch`	A factor specifying the batch of origin for all cells when only a single object is supplied in `...`. This is ignored if multiple objects are present.
`norm.args`	A named list of further arguments to pass to `logNormCounts`.
`min.mean`	A numeric scalar specifying the minimum (library size-adjusted) average count of genes to be used for normalization.
`subset.row`	A vector specifying which features to use for normalization.
`normalize.all`	A logical scalar indicating whether normalized values should be returned for all genes.
`preserve.single`	A logical scalar indicating whether to combine the results into a single matrix if only one object was supplied in `...`.
`assay.type`	A string specifying which assay values contains the counts.
`BPPARAM`	A BiocParallelParam object specifying whether calculations should be parallelized.

Details

When performing integrative analyses of multiple batches, it is often the case that different batches have large differences in sequencing depth. This function removes systematic differences in coverage across batches to simplify downstream comparisons. It does so by resaling the size factors using median-based normalization on the ratio of the average counts between batches. This is roughly equivalent to the between-cluster normalization described by Lun et al. (2016).

This function will adjust the size factors so that counts in high-coverage batches are scaled *downwards* to match the coverage of the most shallow batch. The `logNormCounts` function will then add the same pseudo-count to all batches before log-transformation. By scaling downwards, we favour stronger squeezing of log-fold changes from the pseudo-count, mitigating any technical differences in variance between batches. Of course, genuine biological differences will also be shrunk, but this is less of an issue for upregulated genes with large counts.

Only genes with library size-adjusted average counts greater than `min.mean` will be used for computing the rescaling factors. This improves precision and avoids problems with discreteness. By default, we use `min.mean=1`, which is usually satisfactory but may need to be lowered for very sparse datasets.

Users can also set `subset.row` to restrict the set of genes used for computing the rescaling factors. By default, normalized values will only be returned for genes specified in the subset. Setting `normalize.all=TRUE` will return normalized values for all genes.

Value

A list of SingleCellExperiment objects with normalized log-expression values in the `"logcounts"` assay (depending on values in `norm.args`). Each object contains cells from a single batch.

If `preserve.single=TRUE` and `...` contains only one SingleCellExperiment, that object is returned with an additional `"logcounts"` assay containing normalized log-expression values. The order of cells is not changed.

Comparison to other normalization strategies

For comparison, imagine if we ran `logNormCounts` separately in each batch prior to correction. Size factors will be computed within each batch, and batch-specific application in `logNormCounts` will not account for scaling differences between batches. In contrast, `multiBatchNorm` will rescale the size factors so that they are comparable across batches. This removes at least one difference between batches to facilitate easier correction.

`cosineNorm` performs a similar role of equalizing the scale of expression values across batches. However, the advantage of `multiBatchNorm` is that its output is more easily interpreted - the normalized values remain on the log-scale and differences can still be interpreted (roughly) as log-fold changes. The output can then be fed into downstream analysis procedures (e.g., HVG detection) in the same manner as typical log-normalized values from `logNormCounts`.

Author(s)

Aaron Lun

References

Lun ATL (2018). Further MNN algorithm development. https://MarioniLab.github.io/FurtherMNN2018/theory/description.html

See Also

mnnCorrect and fastMNN, for methods that can benefit from rescaling.

logNormCounts for the calculation of log-transformed normalized expression values.

applyMultiSCE, to apply this function over the altExps in x.

Examples

```
d1 <- matrix(rnbinom(50000, mu=10, size=1), ncol=100)
sce1 <- SingleCellExperiment(list(counts=d1))
sizeFactors(sce1) <- runif(ncol(d1))

d2 <- matrix(rnbinom(20000, mu=50, size=1), ncol=40)
sce2 <- SingleCellExperiment(list(counts=d2))
sizeFactors(sce2) <- runif(ncol(d2))

out <- multiBatchNorm(sce1, sce2)
summary(sizeFactors(out[[1]]))
summary(sizeFactors(out[[2]]))
```

---

multiBatchPCA *Multi-batch PCA*

Description

Perform a principal components analysis across multiple gene expression matrices to project all cells to a common low-dimensional space.

Usage

```
multiBatchPCA(
  ...,
  batch = NULL,
  d = 50,
  subset.row = NULL,
  weights = NULL,
  get.all.genes = FALSE,
  get.variance = FALSE,
  preserve.single = FALSE,
  assay.type = "logcounts",
  BSPARAM = IrlbaParam(),
  deferred = TRUE,
  BPPARAM = SerialParam()
)
```

Arguments

...	Two or more matrices containing expression values (usually log-normalized). Each matrix is assumed to represent one batch and should contain the same number of rows, corresponding to the same genes in the same order. Alternatively, two or more SingleCellExperiment objects containing these matrices. Note the same restrictions described above for gene expression matrix inputs. Alternatively, one matrix or SingleCellExperiment can be supplied containing cells from all batches. This requires batch to also be specified. Alternatively, one or more lists of matrices or SingleCellExperiments can be provided; this is flattened as if the objects inside were passed directly to ....
batch	A factor specifying the batch identity of each cell in the input data. Ignored if ... contains more than one argument.
d	An integer scalar specifying the number of dimensions to keep from the PCA. Alternatively NA, in which case the PCA step is omitted entirely - see details below.
subset.row	A vector specifying which features to use for correction.
weights	Numeric vector, logical scalar or list specifying the weighting scheme to use, see below for details.
get.all.genes	A logical scalar indicating whether the reported rotation vectors should include genes that are excluded by a non-NULL value of subset.row.
get.variance	A logical scalar indicating whether to return the (weighted) variance explained by each PC.
preserve.single	A logical scalar indicating whether to combine the results into a single matrix if only one object was supplied in ....
assay.type	A string or integer scalar specifying the assay containing the expression values, if SingleCellExperiment objects are present in ....
BSPARAM	A BiocSingularParam object specifying the algorithm to use for PCA, see runSVD for details.
deferred	A logical scalar used to overwrite the deferred status of BSPARAM for greater speed. Set to NULL to use the supplied status in BSPARAM directly.
BPPARAM	A BiocParallelParam object specifying whether the SVD should be parallelized.

Details

This function is roughly equivalent to cbinding all matrices in ... and performing PCA on the merged matrix. The main difference is that each sample is forced to contribute equally to the identification of the rotation vectors. Specifically, the mean vector used for centering is defined as the grand mean of the mean vectors within each batch. Each batch's contribution to the gene-gene covariance matrix is also divided by the number of cells in that batch.

Our approach is to effectively weight the cells in each batch to mimic the situation where all batches have the same number of cells. This ensures that the low-dimensional space can distinguish subpopulations in smaller batches. Otherwise, batches with a large number of cells would dominate

the PCA, i.e., the definition of the mean vector and covariance matrix. This may reduce resolution of unique subpopulations in smaller batches that differ in a different dimension to the subspace of the larger batches.

It is usually recommended to set `subset.row` to a subset of interesting features (e.g., highly variable genes). This reduces computational time and eliminates uninteresting noise that could interfere with identification of the most relevant axes of variation. Setting `get.all.genes=TRUE` will report rotation vectors that span all genes, even when only a subset of genes are used for the PCA. This is done by projecting all non-used genes into the low-dimensional "cell space" defined by the first `d` components.

Value

A List of numeric matrices is returned where each matrix corresponds to a batch and contains the first `d` PCs (columns) for all cells in the batch (rows).

If `preserve.single=TRUE` and `...` contains a single object, the List will only contain a single matrix. This contains the first `d` PCs (columns) for all cells in the same order as supplied in the single input object.

The metadata contains `rotation`, a matrix of rotation vectors, which can be used to construct a low-rank approximation of the input matrices. This has number of rows equal to the number of genes after any subsetting, except if `get.all.genes=TRUE`, where the number of rows is equal to the genes before subsetting.

If `get.variance=TRUE`, the metadata will also contain `var.explained`, a numeric vector containing the weighted variance explained by each of `d` PCs; and `var.total`, the total variance across all components after weighting.

Tuning the weighting

By default, `weights=NULL` or `TRUE` will use the default weights, i.e., the reciprocal of the number of cells in each batch. This equalizes the contribution of batches with different numbers of cells as described above.

If `weights=FALSE`, no weighting is performed. This means that larger batches will drive the PCA, which may be desirable when dealing with technical replicates where there is no concern about unique subpopulations in smaller batches.

If `weights` is a numeric vector, it is expected to be of the same length (and, if named, have the same names) as the entries in `...`. Each entry of the vector is used to scale the default weight of the corresponding batch. This allows users to fine-tune the contribution of each batch in situations with multiple levels of heterogeneity. For example, consider merging data from multiple donors where each donor contains a variable number of batches. In such cases, it may be more appropriate to ensure that each donor has equal weight, rather than each batch. This is done by assigning a value of `weights` to each replicate that is inversely proportional to the number of batches for the same donor - see Examples.

Alternatively, `weights` can be a list representing a tree-like structure, identical to the tree controlling the merge order in `fastMNN`. Here, weights are scaled so that each partition in the tree yields subtrees with identical scaled weights (summed across each subtree's children). This allows us to easily adjust the weighting scheme for hierarchical batch structures like the replicate-study scenario described above.

Performing the PCA

Most of the parameters related to the PCA are controlled by BSPARAM, which determines the choice and parameterization of SVD algorithms. The default is to use IrlbaParam though RandomParam is often faster (at the cost of some accuracy). Most choices of BSPARAM will interactly sanely with BPPARAM to achieve parallelization during the PCA itself.

With the default deferred=TRUE, the per-gene centering and per-cell scaling will be deferred during matrix multiplication in approximate SVD algorithms. This is much faster when the input matrices are sparse, as deferred operations avoids loss of sparsity at the cost of numerical precision. If deferred=NULL, the use of deferred scaling is determined by the setting within BSPARAM itself - see ?bsdeferred for details.

If d=NA, no PCA is performed. Instead, the centered matrices are transposed and returned directly, while the rotation matrix is set to an identity matrix. This allows developers to easily switch between PCA-based approximations versus the underlying dataset in their functions by simply changing d. (In some settings, one can even interpret d=NA as the maximum d, as Euclidean distance calculations between cells will be identical to a full-rank PC matrix.) Note that, in this mode, no projection will be done with get.all.genes=TRUE; rather, genes not in subset.row will simply have rotation values of zero. If get.variance=TRUE, the weighted variances of the individual genes are returned.

Author(s)

Aaron Lun

See Also

runSVD

Examples

```
d1 <- matrix(rnorm(5000), ncol=100)
d1[1:10,1:10] <- d1[1:10,1:10] + 2 # unique population in d1
d2 <- matrix(rnorm(2000), ncol=40)
d2[11:20,1:10] <- d2[11:20,1:10] + 2 # unique population in d2

# PCA defaults to IRLBA, so we need to set the seed.
set.seed(10)
out <- multiBatchPCA(d1, d2, d=10)

# Examining results.
xlim <- range(c(out[[1]][,1], out[[2]][,1]))
ylim <- range(c(out[[1]][,2], out[[2]][,2]))
plot(out[[1]][,1], out[[1]][,2], col="red", xlim=xlim, ylim=ylim)
points(out[[2]][,1], out[[2]][,2], col="blue")

# Using the weighting scheme, assuming that 'd2' and 'd3'
# are replicates and should contribute the same combined
# weight as 'd1'.
d3 <- d2 + 5
set.seed(10)
out <- multiBatchPCA(d1, d2, d3, d=10, weights=c(1, 0.5, 0.5))
```

```
# Alternative method for specifying weights as a tree.
set.seed(10)
alt <- multiBatchPCA(d1, d2, d3, d=10, weights=list(1, list(2, 3)))
```

`noCorrect` *No correction*

Description

Provides a no-correction method that has the same interface as the correction functions. This allows users to easily swap function calls to examine the effect of correction.

Usage

```
noCorrect(
  ...,
  batch = NULL,
  subset.row = NULL,
  correct.all = FALSE,
  assay.type = "logcounts"
)
```

Arguments

`...`	One or more log-expression matrices where genes correspond to rows and cells correspond to columns. Alternatively, one or more SingleCellExperiment objects can be supplied containing a log-expression matrix in the `assay.type` assay. Each object should contain the same number of rows, corresponding to the same genes in the same order. Objects of different types can be mixed together. If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. If a single object is supplied, it is assumed to contain cells from all batches, so `batch` should also be specified. Alternatively, one or more lists of matrices or SingleCellExperiments can be provided; this is flattened as if the objects inside each list were passed directly to `....`
`batch`	A vector or factor specifying the batch of origin for all cells when only a single object is supplied in `....` This is ignored if multiple objects are present.
`subset.row`	A vector specifying which features to use for correction.
`correct.all`	Logical scalar indicating whether corrected expression values should be computed for genes not in `subset.row`. Only relevant if `subset.row` is not `NULL`.
`assay.type`	A string or integer scalar specifying the assay containing the log-expression values. Only used for SingleCellExperiment inputs.

Details

This function is effectively equivalent to `cbind`ing the matrices together without any correction. The aim is to provide a consistent interface that allows users to simply combine batches without additional operations. This is often desirable as a negative control to see if the transformation is actually beneficial. It also allows for convenient downstream analyses that are based on the uncorrected data, e.g., differential expression.

Setting `correct.all=TRUE` is equivalent to forcing `subset.row=NULL`, given that no correction is being performed anyway.

In the case of a single object in `...`, `batch` has no effect beyond being stored in the `colData` of the output.

Value

A SingleCellExperiment is returned where each row is a gene and each column is a cell. This contains:

- A `merged` matrix in the `assays` slot, containing the merged expression values from all elements of `...`.
- A `batch` column in the `colData` slot, containing the batch of origin for each row (i.e., cell) in `corrected`.

Author(s)

Aaron Lun

Examples

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

# Same as combining the expression values.
stopifnot(all(assay(out)==cbind(B1, B2)))

# Specifies which cell came from which batch:
str(out$batch)
```

---

quickCorrect *Quickly perform batch correction*

Description

Quickly perform batch correction by intersecting the gene features, normalizing and log-transforming, modelling per-gene variances and identifying highly variable genes, and then applying the correction algorithm of choice.

Usage

```
quickCorrect(
  ...,
  batch = NULL,
  restrict = NULL,
  correct.all = TRUE,
  assay.type = "counts",
  PARAM = FastMnnParam(),
  multi.norm.args = list(),
  precomputed = NULL,
  model.var.args = list(),
  hvg.args = list(n = 5000)
)
```

Arguments

`...`	One or more matrix-like objects containing single-cell gene expression matrices. Alternatively, one or more SingleCellExperiment objects can be supplied. If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. Objects of different types can be mixed together. If a single object is supplied, `batch` should also be specified.
`batch`	A factor specifying the batch of origin for each cell if only one batch is supplied in `...`. This will be ignored if two or more batches are supplied.
`restrict`	A list of length equal to the number of objects in `...`. Each entry of the list corresponds to one batch and specifies the cells to use when computing the correction.
`correct.all`	A logical scalar indicating whether to return corrected expression values for all genes, even if `subset.row` is set. Used to ensure that the output is of the same dimensionality as the input.
`assay.type`	A string or integer scalar specifying the assay to use for correction. Only used for SingleCellExperiment inputs.
`PARAM`	A BatchelorParam object specifying the batch correction method to dispatch to, and the parameters with which it should be run. ClassicMnnParam will dispatch to `mnnCorrect`; FastMnnParam will dispatch to `fastMNN`; RescaleParam will dispatch to `rescaleBatches`; and RegressParam will dispatch to `regressBatches`.
`multi.norm.args`	Named list of further arguments to pass to `multiBatchNorm`.
`precomputed`	List of DataFrames containing precomputed variance modelling results. This should be a list of the same length as the number of entries in `...`, and each should have the same row names as the corresponding entry of `...`.
`model.var.args`	Named list of further arguments to pass to `modelGeneVar`.
`hvg.args`	Named list of further arguments to pass to `getTopHVGs`. By default, we take the top 5000 genes with the highest variances.

Details

This function wraps the sequence of typical steps required to obtain corrected expression values.'

1. Intersecting each batch to the universe of common features with `intersectRows`.
2. Applying normalization and log-transformation to the batches with `multiBatchNorm`.
3. Modelling the per-gene variance with `modelGeneVar`. If `precomputed` is supplied, the precomputed results are used instead.
4. Identifying highly variable genes with `getTopHVGs`. These genes will be used in the correction, though corrected values for all genes can be returned by setting `correct.all=TRUE`.
5. Applying the batch correction algorithm of choice with `batchCorrect`, as specified by `PARAM`.

The default of `correct.all=TRUE` differs from that of other functions. This is because the subsetting to HVGs is done internally here, and we avoid surprises by returning results for all genes in the input object(s). In contrast, the other functions require explicit subsetting via `subset.row=` and it is expected that users will set `correct.all=` if all genes are desired.

Value

A list containing:

- `dec`, a DataFrame containing the combined variance modelling results across all batches.
- `hvgs`, a character vector of the genes selected to use in the correction.
- `corrected`, a SingleCellExperiment containing the corrected values across all cells in all batches.

Author(s)

Aaron Lun

See Also

`intersectRows`, for the intersection to obtain a universe of genes.

`multiBatchNorm`, to perform the normalization.

`modelGeneVar` and `getTopHVGs`, to identify the top HVGs for use in correction.

`batchCorrect`, to dispatch to the desired correction algorithm.

Examples

```
d1 <- matrix(rnbinom(50000, mu=10, size=1), ncol=100)
sce1 <- SingleCellExperiment(list(counts=d1))
sizeFactors(sce1) <- runif(ncol(d1))
rownames(sce1) <- paste0("GENE", 1:500)

d2 <- matrix(rnbinom(20000, mu=50, size=1), ncol=40)
sce2 <- SingleCellExperiment(list(counts=d2))
sizeFactors(sce2) <- runif(ncol(d2))
rownames(sce2) <- paste0("GENE", 201:700)
```

```
# Fire and forget:
set.seed(1000)
output <- quickCorrect(sce1, sce2)
output$corrected
```

`reducedMNN` *MNN correction in reduced dimensions*

Description

MNN correction in reduced dimensions

Usage

```
reducedMNN(
  ...,
  batch = NULL,
  k = 20,
  prop.k = NULL,
  restrict = NULL,
  ndist = 3,
  merge.order = NULL,
  auto.merge = FALSE,
  min.batch.skip = 0,
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam()
)
```

Arguments

`...`	One or more matrices of low-dimensional representations where rows are cells and columns are dimensions. Each object should contain the same number of columns, corresponding to the same dimensions. These should have been generated by a single call to `multiBatchPCA`. If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. If a single object is supplied, `batch` should also be specified. Alternatively, any number of lists of such objects. this is flattened as if the objects inside each list were passed directly to `...`.
`batch`	A vector or factor specifying the batch of origin for all cells when only a single object is supplied in `...`. This is ignored if multiple objects are present.
`k`	An integer scalar specifying the number of nearest neighbors to consider when identifying MNNs.
`prop.k`	A numeric scalar in (0, 1) specifying the proportion of cells in each dataset to use for mutual nearest neighbor searching. If set, the number of nearest neighbors used for the MNN search in each batch is redefined as `max(k, prop.k*N)` where `N` is the number of cells in that batch.

restrict	A list of length equal to the number of objects in .... Each entry of the list corresponds to one batch and specifies the cells to use when computing the correction.
ndist	A numeric scalar specifying the threshold beyond which neighbours are to be ignored when computing correction vectors. Each threshold is defined as a multiple of the number of median distances.
merge.order	An integer vector containing the linear merge order of batches in .... Alternatively, a list of lists representing a tree structure specifying a hierarchical merge order.
auto.merge	Logical scalar indicating whether to automatically identify the "best" merge order.
min.batch.skip	Numeric scalar specifying the minimum relative magnitude of the batch effect, below which no correction will be performed at a given merge step.
BNPARAM	A BiocNeighborParam object specifying the nearest neighbor algorithm.
BPPARAM	A BiocParallelParam object specifying whether the PCA and nearest-neighbor searches should be parallelized.

Details

reducedMNN performs the same operations as fastMNN but assumes that the PCA has already been performed. This is useful as the PCA (via multiBatchPCA) is often the most time-consuming step. By performing the PCA once, reducedMNN allows the MNN correction to be quickly repeated with different parameters.

reducedMNN operates on the same principles as fastMNN, so users are referred to the documentation for the latter for more details on the effect of each of the arguments. Obviously, any arguments pertaining to gene-based steps in fastMNN are not relevant here.

Note that multiBatchPCA will not perform cosine-normalization, so it is the responsibility of the user to cosine-normalize each batch beforehand with cosineNorm to recapitulate results of fastMNN with cos.norm=TRUE. In addition, multiBatchPCA must be run on all samples at once, to ensure that all cells are projected to the same low-dimensional space.

Value

A DataFrame is returned where each row corresponds to a cell, containing:

- corrected, the matrix of corrected low-dimensional coordinates for each cell.
- batch, the Rle specifying the batch of origin for each row.

Cells in the output object are always ordered in the same manner as supplied in .... The metadata on this object is the same as that in the output of fastMNN.

Author(s)

Aaron Lun

See Also

multiBatchPCA, to obtain the values to be corrected.

fastMNN, for the version that operates on gene-expression values.

clusterMNN, for an application on cluster centroids.

Examples

```
B1 <- matrix(rnorm(10000), nrow=50) # Batch 1
B2 <- matrix(rnorm(10000), nrow=50) # Batch 2

# Corrected values equivalent to fastMNN().
cB1 <- cosineNorm(B1)
cB2 <- cosineNorm(B2)
pcs <- multiBatchPCA(cB1, cB2)
mnn.out <- reducedMNN(pcs[[1]], pcs[[2]])

mnn.out
```

---

regressBatches *Regress out batch effects*

Description

Fit a linear model to each gene regress out uninteresting factors of variation, returning a matrix of residuals.

Usage

```
regressBatches(
  ...,
  batch = NULL,
  design = NULL,
  keep = NULL,
  restrict = NULL,
  subset.row = NULL,
  correct.all = FALSE,
  d = NA,
  deferred = TRUE,
  assay.type = "logcounts",
  BSPARAM = IrlbaParam(),
  BPPARAM = SerialParam()
)
```

Arguments

...	One or more log-expression matrices where genes correspond to rows and cells correspond to columns. Alternatively, one or more SingleCellExperiment objects can be supplied containing a log-expression matrix in the `assay.type` assay. Each object should contain the same number of rows, corresponding to the same genes in the same order. Objects of different types can be mixed together. If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. If a single object is supplied, it is assumed to contain cells from all batches, so `batch` should also be specified. Alternatively, one or more lists of matrices or SingleCellExperiments can be provided; this is flattened as if the objects inside each list were passed directly to `...`.
batch	A vector or factor specifying the batch of origin for all cells when only a single object is supplied in `...`. This is ignored if multiple objects are present.
design	A numeric design matrix with number of rows equal to the total number of cells, specifying the experimental factors to remove. Each row corresponds to a cell in the order supplied in `...`.
keep	Integer vector specifying the coefficients of `design` to *not* regress out, see the `ResidualMatrix` constructor for more details.
restrict	A list of length equal to the number of objects in `...`. Each entry of the list corresponds to one batch and specifies the cells to use when computing the correction.
subset.row	A vector specifying which features to use for correction.
correct.all	Logical scalar indicating whether corrected expression values should be computed for genes not in `subset.row`. Only relevant if `subset.row` is not `NULL`.
d	Numeric scalar specifying the number of dimensions to use for PCA via `multiBatchPCA`. If `NA`, no PCA is performed.
deferred	Logical scalar indicating whether to defer centering/scaling, see `multiBatchPCA` for details.
assay.type	A string or integer scalar specifying the assay containing the log-expression values. Only used for SingleCellExperiment inputs.
BSPARAM	A BiocSingularParam object specifying the algorithm to use for PCA in `multiBatchPCA`.
BPPARAM	A BiocParallelParam object specifying whether the PCA should be parallelized.

Details

This function fits a linear model to the log-expression values for each gene and returns the residuals. By default, the model is parameterized as a one-way layout with the batch of origin, so the residuals represent the expression values after correcting for the batch effect. The novelty of this function is that it returns a ResidualMatrix in as the `"corrected"` assay. This avoids explicitly computing the residuals, which would result in a loss of sparsity or similar problems. Rather, residuals are either computed as needed or are never explicitly computed at all (e.g., during matrix multiplication). This means that `regressBatches` is faster and lighter than naive regression or even `rescaleBatches`.

More complex designs should be explicitly specified with the `design` argument, e.g., to regress out a covariate. This can be any full-column-rank matrix that is typically constructed with `model.matrix`.

If design is specified with a single object in ..., batch is ignored. If design is specified with multiple objects, regression is applied to the matrix obtained by cbinding all of those objects together; this means that the first few rows of design correspond to the cells from the first object, then the next rows correspond to the second object and so on.

Like rescaleBatches, this function assumes that the batch effect is orthogonal to the interesting factors of variation. For example, each batch is assumed to have the same composition of cell types. The same reasoning applies to any uninteresting factors specified in design, including continuous variables. For example, if one were to use this function to regress out cell cycle, the assumption is that all cell types are similarly distributed across cell cycle phases. If this is not true, the correction will not only be incomplete but can introduce spurious differences.

See ?"batchelor-restrict" for a description of the restrict argument. Specifically, this function will compute the model coefficients using only the specified subset of cells. The regression will then be applied to all cells in each batch.

If set, the d option will perform a PCA via multiBatchPCA. This is provided for convenience as efficiently executing a PCA on a ResidualMatrix is not always intuitive. (Specifically, BiocSingularParam objects must be set up with deferred=TRUE for best performance.) The arguments BSPARAM, deferred and BPPARAM only have an effect when d is set to a non-NA value.

All genes are used with the default setting of subset.row=NULL. If a subset of genes is specified, residuals are only returned for that subset. Similarly, if d is set, only the genes in the subset are used to perform the PCA. If additionally correct.all=TRUE, residuals are returned for all genes but only the subset is used for the PCA.

Value

A SingleCellExperiment object containing the corrected assay. This contains the computed residuals for each gene (row) in each cell (column) in each batch. A batch field is present in the column data, specifying the batch of origin for each cell.

Cells in the output object are always ordered in the same manner as supplied in .... For a single input object, cells will be reported in the same order as they are arranged in that object. In cases with multiple input objects, the cell identities are simply concatenated from successive objects, i.e., all cells from the first object (in their provided order), then all cells from the second object, and so on.

If d is not NA, a PCA is performed on the residual matrix via multiBatchPCA, and an additional corrected field is present in the reducedDims of the output object.

Author(s)

Aaron Lun

See Also

rescaleBatches, for another approach to regressing out the batch effect.

The ResidualMatrix class, for the class of the residual matrix.

applyMultiSCE, to apply this across multiple altExps.

Examples

```
means <- 2^rgamma(1000, 2, 1)
A1 <- matrix(rpois(10000, lambda=means), ncol=50) # Batch 1
A2 <- matrix(rpois(10000, lambda=means*runif(1000, 0, 2)), ncol=50) # Batch 2

B1 <- log2(A1 + 1)
B2 <- log2(A2 + 1)
out <- regressBatches(B1, B2)
```

`rescaleBatches` *Scale counts across batches*

Description

Scale counts so that the average count within each batch is the same for each gene.

Usage

```
rescaleBatches(
  ...,
  batch = NULL,
  restrict = NULL,
  log.base = 2,
  pseudo.count = 1,
  subset.row = NULL,
  correct.all = FALSE,
  assay.type = "logcounts"
)
```

Arguments

`...`	One or more log-expression matrices where genes correspond to rows and cells correspond to columns. Alternatively, one or more SingleCellExperiment objects can be supplied containing a log-expression matrix in the `assay.type` assay. Each object should contain the same number of rows, corresponding to the same genes in the same order. Objects of different types can be mixed together. If multiple objects are supplied, each object is assumed to contain all and only cells from a single batch. If a single object is supplied, it is assumed to contain cells from all batches, so `batch` should also be specified. Alternatively, one or more lists of matrices or SingleCellExperiments can be provided; this is flattened as if the objects inside each list were passed directly to `...`.
`batch`	A vector or factor specifying the batch of origin for all cells when only a single object is supplied in `...`. This is ignored if multiple objects are present.

restrict	A list of length equal to the number of objects in .... Each entry of the list corresponds to one batch and specifies the cells to use when computing the correction.
log.base	A numeric scalar specifying the base of the log-transformation.
pseudo.count	A numeric scalar specifying the pseudo-count used for the log-transformation.
subset.row	A vector specifying which features to use for correction.
correct.all	Logical scalar indicating whether corrected expression values should be computed for genes not in subset.row. Only relevant if subset.row is not NULL.
assay.type	A string or integer scalar specifying the assay containing the log-expression values. Only used for SingleCellExperiment inputs.

Details

This function assumes that the log-expression values were computed by a log-transformation of normalized count data, plus a pseudo-count. It reverses the log-transformation and scales the underlying counts in each batch so that the average (normalized) count is equal across batches. The assumption here is that each batch contains the same population composition. Thus, any scaling difference between batches is technical and must be removed.

This function is approximately equivalent to centering in log-expression space, the simplest application of linear regression methods for batch correction. However, by scaling the raw counts, it avoids loss of sparsity that would otherwise result from centering. It also mitigates issues with artificial differences in variance due to log-transformation. This is done by always downscaling to the lowest average expression for each gene such that differences in variance are dampened by the addition of the pseudo-count.

Use of rescaleBatches assumes that the uninteresting factors described in design are orthogonal to the interesting factors of variation. For example, each batch is assumed to have the same composition of cell types. If this is not true, the correction will not only be incomplete but may introduce spurious differences.

The output values are always re-log-transformed with the same log.base and pseudo.count. These can be used directly in place of the input values for downstream operations.

All genes are used with the default setting of subset.row=NULL. Users can set subset.row to subset the inputs, though this is purely for convenience as each gene is processed independently of other genes.

See ?"batchelor-restrict" for a description of the restrict argument. Specifically, the function will compute the scaling differences using only the specified subset of cells, and then apply the re-scaling to all cells in each batch.

Value

A SingleCellExperiment object containing the corrected assay. This contains corrected log-expression values for each gene (row) in each cell (column) in each batch. A batch field is present in the column data, specifying the batch of origin for each cell.

Cells in the output object are always ordered in the same manner as supplied in .... For a single input object, cells will be reported in the same order as they are arranged in that object. In cases with multiple input objects, the cell identities are simply concatenated from successive objects, i.e., all cells from the first object (in their provided order), then all cells from the second object, and so on.

Author(s)

Aaron Lun

See Also

regressBatches, for a residual calculation based on a fitted linear model.

applyMultiSCE, to apply this function over multiple altExps.

Examples

```
means <- 2^rgamma(1000, 2, 1)
A1 <- matrix(rpois(10000, lambda=means), ncol=50) # Batch 1
A2 <- matrix(rpois(10000, lambda=means*runif(1000, 0, 2)), ncol=50) # Batch 2

B1 <- log2(A1 + 1)
B2 <- log2(A2 + 1)
out <- rescaleBatches(B1, B2)
```

Index

altExps, *4*, *30*, *43*, *55*, *58*
applyMultiSCE, 2, *43*, *55*, *58*
applySCE, *2–4*
assays, *18*

batchCorrect, 5, *10*, *17–19*, *50*
batchCorrect,ClassicMnnParam-method *(batchCorrect)*, 5
batchCorrect,FastMnnParam-method *(batchCorrect)*, 5
batchCorrect,NoCorrectParam-method *(batchCorrect)*, 5
batchCorrect,RegressParam-method *(batchCorrect)*, 5
batchCorrect,RescaleParam-method *(batchCorrect)*, 5
batchelor-restrict, 8
BatchelorParam, *7, 8*, *18*, *49*
BatchelorParam-class, 9
BiocNeighborParam, *13*, *25*, *33*, *52*
BiocParallelParam, *13*, *19*, *25*, *33*, *39*, *41*, *44*, *52*, *54*
BiocSingularParam, *13*, *25*, *33*, *44*, *54, 55*
bsdeferred, *46*

checkBatchConsistency, 11
checkIfSCE *(checkBatchConsistency)*, 11
checkRestrictions *(checkBatchConsistency)*, 11
chisq.test, *21*
ClassicMnnParam, *7*, *18*, *49*
ClassicMnnParam *(BatchelorParam-class)*, 9
ClassicMnnParam-class *(BatchelorParam-class)*, 9
clusterAbundanceTest *(diagnostics-cluster)*, 21
clusterAbundanceVar *(diagnostics-cluster)*, 21
clusterMNN, 12, *30*, *53*
clusterRows, *14*
colData, *14*, *16*, *18*, *48*
convertPCsToSCE, 15
correctExperiments, *8*, 17, *28*
cosineNorm, 19, *25*, *30*, *35*, *40*, *42*, *52*

DataFrame, *15*, *26*, *38, 39*, *49, 50*, *52*
DelayedArray, *19*
DelayedMatrix, *20*
diagnostics-cluster, 21
divideIntoBatches, *12*, 22

fastMNN, *7*, *9*, *14*, *16*, *18*, *20*, 23, *31*, *37, 38*, *40*, *43*, *45*, *49*, *52, 53*
FastMnnParam, *7*, *18*, *49*
FastMnnParam *(BatchelorParam-class)*, 9
FastMnnParam-class *(BatchelorParam-class)*, 9
findMutualNN, *29*, *35, 36*

getTopHVGs, *49, 50*

intersectRows, 30, *50*
IrlbaParam, *46*

lapply, *3*
List, *28*, *35*, *45*
logNormCounts, *34*, *41–43*
LowRankMatrix, *16*

mapply, *3*
mcols, *18*
metadata, *14–16*, *18*, *39*
mnnCorrect, *7*, *9*, *18*, *20*, *25*, *30*, 32, *43*, *49*
mnnDeltaVariance, *30*, 38
model.matrix, *54*
modelGeneVar, *49, 50*
multiBatchNorm, *25*, *34*, 41, *42*, *49, 50*
multiBatchPCA, *15, 16*, *24–26*, *30*, *33*, 43, *51–55*

noCorrect, *19*, 47
NoCorrectParam *(*BatchelorParam-class*)*, 9
NoCorrectParam-class
*(*BatchelorParam-class*)*, 9
normalizeCounts, *21*

quickCorrect, 48

RandomParam, *46*
reducedDims, *16*, *18*, *55*
reducedMNN, *13–16*, *28*, *30*, 51
regressBatches, *7*, *9*, *18*, *49*, 53, *58*
RegressParam, *7*, *18*, *49*
RegressParam *(*BatchelorParam-class*)*, 9
RegressParam-class
*(*BatchelorParam-class*)*, 9
rescaleBatches, *7*, *9*, *18*, *49*, *54, 55*, 56
RescaleParam, *7*, *18*, *49*
RescaleParam *(*BatchelorParam-class*)*, 9
RescaleParam-class
*(*BatchelorParam-class*)*, 9
ResidualMatrix, *54, 55*
rowData, *16*, *18*
rowRanges, *18*
runSVD, *44*, *46*

SimpleList, *10*
simplifyToSCE, *4*
SingleCellExperiment, *2, 3*, *7*, *11, 12*,
14–17, *24*, *26*, *31, 32*, *34*, *38*, *41*, *44*,
47–50, *54–57*