Package ‘COTAN’

March 25, 2024

Type Package

Title COexpression Tables ANalysis

Version 2.2.3

Description Statistical and computational method to analyze the co-expression of gene pairs at single cell level. It provides the foundation for single-cell gene interactome analysis. The basic idea is studying the zero UMI counts' distribution instead of focusing on positive counts; this is done with a generalized contingency tables framework. COTAN can effectively assess the correlated or anti-correlated expression of gene pairs. It provides a numerical index related to the correlation and an approximate p-value for the associated independence test. COTAN can also evaluate whether single genes are differentially expressed, scoring them with a newly defined global differentiation index. Moreover, this approach provides ways to plot and cluster genes according to their co-expression pattern with other genes, effectively helping the study of gene interactions and becoming a new tool to identify cell-identity marker genes.

URL `https://github.com/seriph78/COTAN`

BugReports `https://github.com/seriph78/COTAN/issues`

Depends R (>= 4.2)

License GPL-3

Encoding UTF-8

RoxygenNote 7.2.3

Roxygen list(markdown = TRUE)

Imports stats, plyr, dplyr, methods, grDevices, Matrix, ggplot2, ggrepel, ggthemes, graphics, parallel, parallelly, tibble, tidyr, BiocSingular, PCAtools, parallelDist, ComplexHeatmap, circlize, grid, scales, RColorBrewer, utils, rlang, Rfast, stringr, Seurat, umap, dendextend, zeallot, assertthat, withr

Suggests testthat (>= 3.0.0), proto, spelling, knitr, data.table, gsubfn, R.utils, tidyverse, rmarkdown, htmlwidgets, MASS, Rtsne, plotly, BiocStyle, cowplot, qpdf, GEOquery, sf

Config/testthat/edition 3

Language en-US

biocViews SystemsBiology, Transcriptomics, GeneExpression, SingleCell

VignetteBuilder knitr

LazyData false

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

git_branch RELEASE_3_18

git_last_commit 34a23a9

git_last_commit_date 2024-01-02

Repository Bioconductor 3.18

Date/Publication 2024-03-25

Author Galfrè Silvia Giulia [aut, cre]
(<`https://orcid.org/0000-0002-2770-0344`>),
Morandin Francesco [aut] (<`https://orcid.org/0000-0002-2022-2300`>),
Fantozzi Marco [aut] (<`https://orcid.org/0000-0002-0708-5495`>),
Pietrosanto Marco [aut] (<`https://orcid.org/0000-0001-5129-6065`>),
Puttini Daniel [aut] (<`https://orcid.org/0009-0006-8401-9949`>),
Priami Corrado [aut] (<`https://orcid.org/0000-0002-3261-6235`>),
Cremisi Federico [aut] (<`https://orcid.org/0000-0003-4925-2703`>),
Helmer-Citterich Manuela [aut]
(<`https://orcid.org/0000-0001-9530-7504`>)

Maintainer Galfrè Silvia Giulia <`silvia.galfre@di.unipi.it`>

R topics documented:

`CalculatingCOEX` *Calculating the COEX matrix for genes and cells*

Description

These are the functions and methods used to calculate the **COEX** matrices according to the `COTAN` model. From there it is possible to calculate the associated *pValue* and the *GDI* (*Global Differential Expression*)

The **COEX** matrix is defined by following formula:

$$\frac{\sum_{i,j \in \{\text{Y, N}\}} (-1)^{\#\{i,j\}} \frac{O_{ij} - E_{ij}}{1 \vee E_{ij}}}{\sqrt{n \sum_{i,j \in \{\text{Y, N}\}} \frac{1}{1 \vee E_{ij}}}}$$

where O and E are the observed and expected contingency tables and n is the relevant numerosity (the number of genes/cells depending on given `actOnCells` flag).

The formula can be more effectively implemented as:

$$\sqrt{\frac{1}{n} \sum_{i,j \in \{\text{Y, N}\}} \frac{1}{1 \vee E_{ij}}} \, (O_{\text{YY}} - E_{\text{YY}})$$

once one notices that $O_{ij} - E_{ij} = (-1)^{\#\{i,j\}} \, r$ for some constant r for all $i, j \in \{\text{Y, N}\}$.

The latter follows from the fact that the relevant marginal sums of the the expected contingency tables were enforced to match the marginal sums of the observed ones.

Usage

```
## S4 method for signature 'COTAN'
getGenesCoex(
  objCOTAN,
  genes = vector(mode = "character"),
  zeroDiagonal = TRUE,
  ignoreSync = FALSE
)

## S4 method for signature 'COTAN'
getCellsCoex(
  objCOTAN,
  cells = vector(mode = "character"),
  zeroDiagonal = TRUE,
  ignoreSync = FALSE
)

## S4 method for signature 'COTAN'
```

```
dropGenesCoex(objCOTAN)

## S4 method for signature 'COTAN'
dropCellsCoex(objCOTAN)

## S4 method for signature 'COTAN'
calculateMu(objCOTAN)

observedContingencyTablesYY(
  objCOTAN,
  actOnCells = FALSE,
  asDspMatrices = FALSE
)

observedContingencyTables(objCOTAN, actOnCells = FALSE, asDspMatrices = FALSE)

expectedContingencyTablesNN(
  objCOTAN,
  actOnCells = FALSE,
  asDspMatrices = FALSE,
  optimizeForSpeed = TRUE
)

expectedContingencyTables(
  objCOTAN,
  actOnCells = FALSE,
  asDspMatrices = FALSE,
  optimizeForSpeed = TRUE
)

contingencyTables(objCOTAN, g1, g2)

## S4 method for signature 'COTAN'
calculateCoex(objCOTAN, actOnCells = FALSE, optimizeForSpeed = TRUE)

calculateS(
  objCOTAN,
  geneSubsetCol = vector(mode = "character"),
  geneSubsetRow = vector(mode = "character")
)

calculateG(
  objCOTAN,
  geneSubsetCol = vector(mode = "character"),
  geneSubsetRow = vector(mode = "character")
)
```

Arguments

`objCOTAN`	a `COTAN` object
`genes`	A vector of gene names. It will exclude any gene not on the list. By defaults the function will keep all genes.
`zeroDiagonal`	When `TRUE` sets the diagonal to zero.
`ignoreSync`	When `TRUE` ignores whether the `lambda/nu/dispersion` have been updated since the `COEX` matrix was calculated.
`cells`	A vector of cell names. It will exclude any cell not on the list. By defaults the function will keep all cells.
`actOnCells`	Boolean; when `TRUE` the function works for the cells, otherwise for the genes
`asDspMatrices`	Boolean; when `TRUE` the function will return only packed dense symmetric matrices
`optimizeForSpeed`	Boolean; when `TRUE` the function will use `Rfast` parallel algorithms that on the flip side use more memory
`g1`	a gene
`g2`	another gene
`geneSubsetCol`	an array of genes. It will be put in columns. If left empty the function will do it genome-wide.
`geneSubsetRow`	an array of genes. It will be put in rows. If left empty the function will do it genome-wide.

Details

`getGenesCoex()` extracts a complete (or a partial after genes dropping) genes' `COEX` matrix from the `COTAN` object.

`getCellsCoex()` extracts a complete (or a partial after cells dropping) cells' `COEX` matrix from the `COTAN` object.

`dropGenesCoex()` drops the `genesCoex` member from the given `COTAN` object

`dropCellsCoex()` drops the `cellsCoex` member from the given `COTAN` object

`calculateMu()` calculates the vector $\mu = \lambda \times \nu^T$

`observedContingencyTablesYY()` calculates observed *Yes/Yes* field of the contingency table

`observedContingencyTables()` calculates the observed contingency tables. When the parameter `asDspMatrices == TRUE`, the method will effectively throw away the lower half from the returned `observedYN` and `observedNY` matrices, but, since they are transpose one of another, the full information is still available.

`expectedContingencyTablesNN()` calculates the expected No/No field of the contingency table

`expectedContingencyTables()` calculates the expected values of contingency tables. When the parameter `asDspMatrices == TRUE`, the method will effectively throw away the lower half from the returned `expectedYN` and `expectedNY` matrices, but, since they are transpose one of another, the full information is still available.

`contingencyTables()` returns the observed and expected contingency tables for a given pair of genes. The implementation runs the same algorithms used to calculate the full observed/expected contingency tables, but restricted to only the relevant genes and thus much faster and less memory intensive

`calculateCoex()` estimates and stores the `COEX` matrix in the `cellCoex` or `genesCoex` field depending on given `actOnCells` flag. It also calculates the percentage of *problematic* genes/cells pairs. A pair is *problematic* when one or more of the expected counts were significantly smaller than 1 (< 0.5). These small expected values signal that scant information is present for such a pair.

`calculateS()` calculates the statistics **S** for genes contingency tables. It always has the diagonal set to zero.

`calculateG()` calculates the statistics *G-test* for genes contingency tables. It always has the diagonal set to zero. It is proportional to the genes' presence mutual information.

Value

`getGenesCoex()` returns the genes' `COEX` values

`getCellsCoex()` returns the cells' `COEX` values

`dropGenesCoex()` returns the updated `COTAN` object

`dropCellsCoex()` returns the updated `COTAN` object

`calculateMu()` returns the `mu` matrix

`observedContingencyTablesYY()` returns a `list` with the *Yes/Yes* observed contingency table as `matrix` and the *Yes* observed `vector`

`observedContingencyTables()` returns the observed contingency tables as named `list` with elements: "observedNN", "observedNY", "observedYN", "observedYY"

`expectedContingencyTablesNN()` returns a `list` with the *No/No* expected contingency table as `matrix` and the *No* expected `vector`

`expectedContingencyTables()` returns the expected contingency tables as named `list` with elements: "expectedNN", "expectedNY", "expectedYN", "expectedYY"

`contingencyTables()` returns a list containing the observed and expected contingency tables

`calculateCoex()` returns the updated `COTAN` object

`calculateS()` returns the `S` matrix

`calculateG()` returns the G matrix

Note

The sum of the matrices returned by the function `observedContingencyTables()` and `expectedContingencyTables()` will have the same value on all elements. This value is the number of genes/cells depending on the parameter `actOnCells` being `TRUE/FALSE`.

See Also

ParametersEstimations for more details.

Examples

```
data("test.dataset")
objCOTAN <- COTAN(raw = test.dataset)
objCOTAN <- initializeMetaDataset(objCOTAN, GEO = "test_GEO",
                                  sequencingMethod = "distribution_sampling",
                                  sampleCondition = "reconstructed_dataset")
objCOTAN <- clean(objCOTAN)

objCOTAN <- estimateDispersionBisection(objCOTAN, cores = 12)

## Now the `COTAN` object is ready to calculate the genes' `COEX`

## mu <- calculateMu(objCOTAN)
## observedY <- observedContingencyTablesYY(objCOTAN, asDspMatrices = TRUE)
obs <- observedContingencyTables(objCOTAN, asDspMatrices = TRUE)

## expectedN <- expectedContingencyTablesNN(objCOTAN, asDspMatrices = TRUE)
exp <- expectedContingencyTables(objCOTAN, asDspMatrices = TRUE)

objCOTAN <- calculateCoex(objCOTAN, actOnCells = FALSE)
genesCoex <- getGenesCoex(objCOTAN)

## S <- calculateS(objCOTAN)
## G <- calculateG(objCOTAN)
## pValue <- calculatePValue(objCOTAN)
GDI <- calculateGDI(objCOTAN)

## Touching any of the lambda/nu/dispersino parameters invalidates the `COEX`
## matrix and derivatives, so it can be dropped it from the `COTAN` object
objCOTAN <- dropGenesCoex(objCOTAN)

objCOTAN <- estimateDispersionNuBisection(objCOTAN, cores = 12)

## Now the `COTAN` object is ready to calculate the cells' `COEX`
## In case one need to caclualte both it is more sensible to run the above
## before any `COEX` evaluation

g1 <- getGenes(objCOTAN)[sample(getNumGenes(objCOTAN), 1)]
g2 <- getGenes(objCOTAN)[sample(getNumGenes(objCOTAN), 1)]
tables <- contingencyTables(objCOTAN, g1 = g1, g2 = g2)
tables

objCOTAN <- calculateCoex(objCOTAN, actOnCells = TRUE)
cellsCoex <- getCellsCoex(objCOTAN)

objCOTAN <- dropCellsCoex(objCOTAN)
```

`ClustersList` Clusters *utilities*

Description

Handle *clusterization* <-> *clusters* `list` conversions, *clusters* grouping and merge

Usage

```
toClustersList(clusters)

fromClustersList(
  clustersList,
  elemNames = vector(mode = "character"),
  throwOnOverlappingClusters = TRUE
)

groupByClustersList(elemNames, clustersList, throwOnOverlappingClusters = TRUE)

groupByClusters(clusters)

mergeClusters(clusters, names, mergedName = "")

multiMergeClusters(clusters, namesList, mergedNames = NULL)
```

Arguments

`clusters`	A named `vector` or `factor` that defines the *clusters*
`clustersList`	A named `list` whose elements define the various clusters
`elemNames`	A `list` of names to which associate a cluster
`throwOnOverlappingClusters`	When `TRUE`, in case of overlapping clusters, the function `fromClustersList` and `groupByClustersList` will throw. This is the default. When FALSE, instead, in case of overlapping clusters, `fromClustersList` will return the last cluster to which each element belongs, while `groupByClustersList` will return a vector of positions that is longer than the given `elemNames`
`names`	A list of *clusters* names to be merged
`mergedName`	The name of the new merged clusters
`namesList`	A `list` of `lists` of *clusters* names to be respectively merged
`mergedNames`	The names of the new merged *clusters*

Details

`toClustersList()` given a *clusterization*, creates a `list` of *clusters* (i.e. for each *cluster*, which elements compose the *cluster*)

`fromClustersList()` given a `list` of *clusters* returns a *clusterization* (i.e. a named `vector` that for each element indicates to which cluster it belongs)

`groupByClusters()` given a *clusterization* returns a permutation, such that using the permutation on the input the *clusters* are grouped together `groupByClustersList()` given the elements' names and a `list` of *clusters* returns a permutation, such that using the permutation on the given names the *clusters* are grouped together.

`mergeClusters()` given a *clusterization*, creates a new one where the given *clusters* are merged.

`multiMergeClusters()` given a *clusterization*, creates a new one where the given sets of *clusters* are merged.

Value

`toClustersList()` returns a `list` of clusters

`fromClustersList()` returns a clusterization. If the given `elemNames` contain values not present in the `clustersList`, those will be marked as `"-1"`

`groupByClusters()` and `groupByClustersList()` return a permutation that groups the clusters together. For each cluster the positions are guaranteed to be in increasing order. In case, all elements not corresponding to any cluster are grouped together as the last group

`mergeClusters()` returns a new *clusterization* with the wanted *clusters* being merged. If less than 2 *cluster* names were passed the function will emit a warning and return the initial *clusterization*

`multiMergeClusters()` returns a new *clusterization* with the wanted *clusters* being merged by consecutive iterations of `mergeClusters()` on the given `namesList`

Examples

```
## create a clusterization
clusters <- paste0("",sample(7, 100, replace = TRUE))
names(clusters) <- paste0("E_",formatC(1:100,  width = 3, flag = "0"))

## create a clusters list from a clusterization
clustersList <- toClustersList(clusters)
head(clustersList, 1)

## recreate the clusterization from the cluster list
clusters2 <- fromClustersList(clustersList, names(clusters))
all.equal(factor(clusters), clusters2)

cl1Size <- length(clustersList[["1"]])

## establish the permutation that groups clusters together
perm <- groupByClusters(clusters)
!is.unsorted(head(names(clusters)[perm],cl1Size))
head(clusters[perm], cl1Size)

## it is possible to have the list of the element names different
## from the names in the clusters list
selectedNames <- paste0("E_",formatC(11:110,  width = 3, flag = "0"))
perm2 <- groupByClustersList(selectedNames, toClustersList(clusters))
all.equal(perm2[91:100], c(91:100))

## is is possible to merge a few clusters together
clustersMerged <- mergeClusters(clusters, names = c("7", "2"),
                                mergedName = "7__2")
```

```
sum(table(clusters)[c(2, 7)]) == table(clustersMerged)[["7__2"]]

## it is also possible to do multiple merges at once!
## Note the default new clusters' names
clustersMerged2 <-
  multiMergeClusters(clusters2, namesList = list(c("2", "7"),
                                                  c("1", "3", "5")))
table(clustersMerged2)
```

`COTAN` *COTAN*

Description

Constructor of the class `COTAN`

Usage

```
COTAN(raw = "ANY")
```

Arguments

`raw`	any object that can be converted to a matrix, but with row (genes) and column (cells) names

Value

a `COTAN` object

Examples

```
data("test.dataset")
obj <- COTAN(raw = test.dataset)
```

`COTAN-class` *Definition of the* `COTAN` *class*

Description

Definition of the `COTAN` class

Slots

`raw`	`dgCMatrix` - the raw UMI count matrix $n \times m$ (gene number × cell number)
`genesCoex`	`dspMatrix` - the correlation of `COTAN` between genes, $n \times n$
`cellsCoex`	`dspMatrix` - the correlation of `COTAN` between cells, $m \times m$
`metaDataset`	`data.frame`
`metaCells`	`data.frame`
`clustersCoex`	a `list` of `COEX` `data.frames` for each clustering in the metaCells

---

`COTANObjectCreation` *Automatic* `COTAN` *shortcuts*

Description

These functions take (or create) a `COTAN` object and run all the necessary steps until the genes' `COEX` matrix is calculated.

takes a newly created `COTAN` object (or the result of a call to `dropGenesCells()`) and applies all steps until the genes' `COEX` matrix is stored in the object

Usage

```
## S4 method for signature 'COTAN'
proceedToCoex(
  objCOTAN,
  calcCoex = TRUE,
  cores = 1L,
  saveObj = TRUE,
  outDir = "."
)

automaticCOTANObjectCreation(
  raw,
  GEO,
  sequencingMethod,
  sampleCondition,
  calcCoex = TRUE,
  cores = 1L,
  saveObj = TRUE,
  outDir = "."
)
```

Arguments

`objCOTAN`	a newly created `COTAN` object
`calcCoex`	a Boolean to determine whether to calculate the genes' `COEX` or stop just before at the `estimateDispersionBisection()` step

`cores`	number of cores to be used
`saveObj`	Boolean flag; when `TRUE` saves intermediate analyses and plots to file
`outDir`	an existing directory for the analysis output.
`raw`	a matrix or dataframe with the raw counts
`GEO`	a code reporting the GEO identification or other specific dataset code
`sequencingMethod`	a string reporting the method used for the sequencing
`sampleCondition`	a string reporting the specific sample condition or time point.

Details

`proceedToCoex()` takes a newly created `COTAN` object (or the result of a call to `dropGenesCells()`) and runs `calculateCoex()`

`automaticCOTANObjectCreation()` takes a raw dataset, creates and initializes a `COTAN` objects and runs proceedToCoex()

Value

`proceedToCoex()` returns the updated `COTAN` object with genes' `COEX` calculated. If asked to, it will also store the object, along all relevant clean-plots, in the output directory.

`automaticCOTANObjectCreation()` returns the new `COTAN` object with genes' `COEX` calculated. When asked, it will also store the object, along all relevant clean-plots, in the output directory.

Examples

```
data("test.dataset")

## In case one needs to run more steps to clean the datatset the following
## might apply
##
## objCOTAN <- COTAN(raw = test.dataset)
## objCOTAN <- initializeMetaDataset(objCOTAN,
##                                   GEO = "test",
##                                   sequencingMethod = "artificial",
##                                   sampleCondition = "test dataset")
## # in case the genes' `COEX` is not needed it can be skipped
## # (e.g. for [cellsUniformClustering()])
## objCOTAN <- proceedToCoex(objCOTAN, calcCoex = FALSE,
##                           cores = 12, saveObj = FALSE)


## Otherwise it is possible to run all at once.
objCOTAN <- automaticCOTANObjectCreation(
  raw = test.dataset,
  GEO = "code",
  sequencingMethod = "10X",
  sampleCondition = "mouse_dataset",
  calcCoex = TRUE,
```

```
    saveObj = FALSE,
    outDir = tempdir(),
    cores = 12)
```

`Datasets` *Data-sets*

Description

Simple data-sets included in the package

Usage

```
data(raw.dataset)

data(ERCCraw)

data(test.dataset)

data(test.dataset.clusters1)

data(test.dataset.clusters2)
```

Format

`raw.dataset` is a data frame with 2000 genes and 815 cells

`ERCCRaw` is a `data.frame`

`test.dataset` is a `data.frame` with 600 genes and 1200 cells

`test.dataset.clusters1` is a `character array`

`test.dataset.clusters2` is a `character array`

Details

`raw.dataset` is a sub-sample of a real *scRNA-seq* data-set

`ERCCRaw` dataset

`test.dataset` is an artificial data set obtained by sampling target negative binomial distributions on a set of 600 genes on 2 two cells *clusters* of 600 cells each. Each *clusters* has its own set of parameters for the distributions even, but a fraction of the genes has the same expression in both *clusters*.

`test.dataset.clusters1` is the *clusterization* obtained running `cellsUniformClustering()` on the `test.dataset`

`test.dataset.clusters2` is the *clusterization* obtained running `mergeUniformCellsClusters()` on the `test.dataset` using the previous *clusterization*

Source

GEO GSM2861514

ERCC

`estimateNuLinearByCluster,COTAN-method`

Handling cells' clusterization *and related functions*

Description

These functions manage the *clusterizations* and their associated *cluster* `COEX` `data.frames`.

A *clusterization* is any partition of the cells where to each cell it is assigned a **label**; a group of cells with the same label is called *cluster*.

For each *cluster* is also possible to define a `COEX` value for each gene, indicating its increased or decreased expression in the *cluster* compared to the whole background. A `data.frame` with these values listed in a column for each *cluster* is stored separately for each *clusterization* in the `clustersCoex` member.

The formulae for this *In/Out* `COEX` are similar to those used in the `calculateCoex()` method, with the **role** of the second gene taken by the *In/Out* status of the cells with respect to each *cluster*.

Usage

```
## S4 method for signature 'COTAN'
estimateNuLinearByCluster(objCOTAN, clName = "", clusters = NULL)

## S4 method for signature 'COTAN'
getClusterizations(objCOTAN, dropNoCoex = FALSE, keepPrefix = FALSE)

## S4 method for signature 'COTAN'
getClusterizationName(objCOTAN, clName = "", keepPrefix = FALSE)

## S4 method for signature 'COTAN'
getClusterizationData(objCOTAN, clName = "")

getClusters(objCOTAN, clName = "")

## S4 method for signature 'COTAN'
getClustersCoex(objCOTAN)

## S4 method for signature 'COTAN'
addClusterization(
  objCOTAN,
  clName,
  clusters,
  coexDF = data.frame(),
```

```
  override = FALSE
)

## S4 method for signature 'COTAN'
addClusterizationCoex(objCOTAN, clName, coexDF)

## S4 method for signature 'COTAN'
dropClusterization(objCOTAN, clName)

DEAOnClusters(objCOTAN, clName = "", clusters = NULL)

pValueFromDEA(coexDF, numCells)

UMAPPlot(df, clusters = NULL, elements = NULL, title = "")

clustersDeltaExpression(objCOTAN, clName = "", clusters = NULL)

clustersMarkersHeatmapPlot(
  objCOTAN,
  groupMarkers,
  clName = "",
  clusters = NULL,
  kCuts = 3L,
  condNameList = NULL,
  conditionsList = NULL
)

clustersSummaryData(
  objCOTAN,
  clName = "",
  clusters = NULL,
  condName = "",
  conditions = NULL
)

clustersSummaryPlot(
  objCOTAN,
  clName = "",
  clusters = NULL,
  condName = "",
  conditions = NULL,
  plotTitle = ""
)

clustersTreePlot(
  objCOTAN,
  kCuts,
  clName = "",
```

```
  distance = "cosine",
  hclustMethod = "ward.D2"
)

findClustersMarkers(
  objCOTAN,
  n = 10L,
  markers = NULL,
  clName = "",
  clusters = NULL,
  coexDF = NULL,
  pValueDF = NULL,
  deltaExp = NULL,
  method = "bonferroni"
)

geneSetEnrichment(clustersCoex, groupMarkers)

reorderClusterization(
  objCOTAN,
  clName = "",
  clusters = NULL,
  coexDF = NULL,
  reverse = FALSE,
  keepMinusOne = TRUE,
  distance = "cosine",
  hclustMethod = "ward.D2"
)
```

Arguments

`objCOTAN`	a `COTAN` object
`clName`	The name of the *clusterization*. If not given the last available *clusterization* will be used, as it is probably the most significant!
`clusters`	A *clusterization* to use. If given it will take precedence on the one indicated by `clName`
`dropNoCoex`	When `TRUE` drops the names from the *clusterizations* with empty associated coex `data.frame`
`keepPrefix`	When `TRUE` returns the internal name of the *clusterization*: the one with the `CL_` prefix.
`coexDF`	a `data.frame` where each column indicates the `COEX` for each of the *clusters* of the *clusterization*
`override`	When `TRUE` silently allows overriding data for an existing *clusterization* name. Otherwise the default behavior will avoid potential data losses
`numCells`	the number of overall cells in all *clusters*
`df`	the `data.frame` to plot. It must have a row names containing the given elements

`elements`	a named `list` of elements to label. Each array in the list will have different color
`title`	a string giving the plot title. Will default to UMAP Plot if not specified
`groupMarkers`	a named `list` with an element for each group comprised of one or more marker genes
`kCuts`	the number of estimated *cluster* (this defines the height for the tree cut)
`condNameList`	a `list` of *conditions*' names to be used for additional columns in the final plot. When none are given no new columns will be added using data extracted via the function `clustersSummaryData()`
`conditionsList`	a `list` of *conditions* to use. If given they will take precedence on the ones indicated by `condNameList`
`condName`	The name of a condition in the `COTAN` object to further separate the cells in more sub-groups. When no condition is given it is assumed to be the same for all cells (no further sub-divisions)
`conditions`	The *conditions* to use. If given it will take precedence on the one indicated by `condName` that will only indicate the relevant column name in the returned `data.frame`
`plotTitle`	The title to use for the returned plot
`distance`	type of distance to use (default is `"cosine"`, `"euclidean"` and the others from `parallelDist::parDist()` are also available)
`hclustMethod`	It defaults is `"ward.D2"` but can be any of the methods defined by the `stats::hclust()` function.
`n`	the number of extreme `COEX` values to return
`markers`	a `list` of marker genes
`pValueDF`	a `data.frame` with *In/Out p-value* based on the `COEX`. E.G. the result of a call to `pValueFromDEA()`
`deltaExp`	a `data.frame` with the *delta-expression* in a *cluster*. E.G. the result of a call to `clustersDeltaExpression()`
`method`	*p-value* adjustment method. Defaults to `"bonferroni"`
`clustersCoex`	the `COEX` `data.frame`
`reverse`	a flag to the output order
`keepMinusOne`	a flag to decide whether to keep the cluster `"-1"` (representing the non-clustered cells) untouched

Details

`estimateNuLinearByCluster()` does a linear estimation of nu: cells' counts averages normalized *cluster* by *cluster*

`getClusterizations()` extracts the list of the *clusterizations* defined in the `COTAN` object.

`getClusterizationName()` normalizes the given *clusterization* name or, if none were given, returns the name of last available *clusterization* in the `COTAN` object. It can return the *clusterization* **internal name** if needed `getClusterizationData()` extracts the asked *clusterization* and its associated `COEX` `data.frame` from the `COTAN` object

`getClusters()` extracts the asked *clusterization* from the `COTAN` object

`getClustersCoex()` extracts the full `clusterCoex` member `list`

`addClusterization()` adds a *clusterization* to the current `COTAN` object, by adding a new column in the `metaCells` `data.frame` and adding a new element in the `clustersCoex` `list` using the passed in `COEX` `data.frame` or an empty `data.frame` if none were passed in.

`addClusterizationCoex()` adds a *clusterization* `COEX` `data.frame` to the current `COTAN` object. It requires the named *clusterization* to be already present.

`dropClusterization()` drops a *clusterization* from the current `COTAN` object, by removing the corresponding column in the `metaCells` `data.frame` and the corresponding `COEX` `data.frame` from the `clustersCoex` `list`.

`DEAOnClusters()` is used to run the Differential Expression analysis using the `COTAN` contingency tables on each *cluster* in the given *clusterization*

`pValueFromDEA()` is used to convert to *p-value* the Differential Expression analysis using the `COTAN` contingency tables on each *cluster* in the given *clusterization*

`UMAPPlot()` plots the given `data.frame` containing genes information related to clusters after applying the UMAP transformation.

`clustersDeltaExpression()` estimates the change in genes' expression inside the *cluster* compared to the average situation in the data set.

`clustersMarkersHeatmapPlot()` returns the heatmap plot of a summary score for each *cluster* and each gene marker list in the given *clusterization*. It also returns the numerosity and percentage of each *cluster* on the right and a gene *clusterization* dendogram on the left (as returned by the function `geneSetEnrichment()`) that allows to estimate which markers groups are more or less expressed in each *cluster* so it is easier to derive the *clusters*' cell types.

`clustersSummaryData()` calculates various statistics about each cluster (with an optional further `condition` to separate the cells).

`clustersSummaryPlot()` calculates various statistics about each cluster via `clustersSummaryData()` and puts them together into a plot.

`clustersTreePlot()` returns the `dendogram` plot where the given *clusters* are placed on the base of their relative distance. Also if needed calculates and stores the `DEA` of the relevant *clusterization*.

`findClustersMarkers()` takes in a `COTAN` object and a *clusterization* and produces a `data.frame` with the `n` most positively enriched and the `n` most negatively enriched genes for each *cluster*. The function also provides whether and the found genes are in the given `markers` list or not. It also returns the *p-value* and the *adjusted p-value* using the `stats::p.adjust()`

`geneSetEnrichment()` returns a cumulative score of enrichment in a *cluster* over a gene set. In formulae it calculates $\frac{1}{n}\sum_i(1-e^{-\theta X_i})$, where the X_i are the positive values from `DEAOnClusters()` and $\theta = -\frac{1}{0.1}\ln(0.25)$

`reorderClusterization()` takes in a *clusterizations* and reorder its labels so that in the new order near labels indicate near clusters according to a `DEA` based distance

Value

`estimateNuLinearByCluster()` returns the updated `COTAN` object

`getClusterizations()` returns a vector of *clusterization* names, usually without the `CL_` prefix

`getClusterizationName()` returns the normalized *clusterization* name or `NULL` if no *clusterizations* are present

`getClusterizationData()` returns a `list` with 2 elements:

- `"clusters"` the named cluster labels array
- `"coex"` the associated `COEX` `data.frame`. This will be an **empty** `data.frame` when not specified for the relevant *clusterization*

`getClusters()` returns the named cluster labels array

`getClustersCoex()` returns the list with a `COEX` `data.frame` for each *clusterization*. When not empty, each `data.frame` contains a `COEX` column for each *cluster*.

`addClusterization()` returns the updated `COTAN` object

`addClusterizationCoex()` returns the updated `COTAN` object

`dropClusterization()` returns the updated `COTAN` object

`DEAOnClusters()` returns the co-expression `data.frame` for the genes in each *cluster*

`pValueFromDEA()` returns a `data.frame` with the *p-values* corresponding to the given *coex*

`UMAPPlot()` returns a `ggplot2` object

`clustersDeltaExpression()` returns a `data.frame` with the weighted discrepancy of the expression of each gene within the *cluster* against model expectations

`clustersMarkersHeatmapPlot()` returns a list with:

- `"heatmapPlot"` the complete heatmap plot
- `"dataScore"` the `data.frame` with the score values

`clustersSummaryData()` returns a `data.frame` with the following statistics: The calculated statistics are:

- `"clName"` the *cluster* **labels**
- `"condName"` the relevant condition (that sub-divides the *clusters*)
- `"CellNumber"` the number of cells in the group
- `"MeanUDE"` the average "UDE" in the group of cells
- `"MedianUDE"` the median "UDE" in the group of cells
- `"ExpGenes25"` the number of genes expressed in at the least 25% of the cells in the group
- `"ExpGenes"` the number of genes expressed at the least once in any of the cells in the group
- `"CellPercentage"` fraction of the cells with respect to the total cells

`clustersSummaryPlot()` returns a `list` with a `data.frame` and a `ggplot` objects

- `"data"` contains the data,
- `"plot"` is the returned plot

`clustersTreePlot()` returns a list with 2 objects:

- `"dend"` a `ggplot2` object representing the `dendrogram` plot
- `"objCOTAN"` the updated `COTAN` object

`findClustersMarkers()` returns a `data.frame` containing `n` top/bottom `COEX` scores for each *cluster*

`geneSetEnrichment()` returns a `data.frame` with the cumulative score

`reorderClusterization()` returns a `list` with 2 elements:

- `"clusters"` the newly reordered cluster labels array
- `"coex"` the associated `COEX` `data.frame`

Examples

```
data("test.dataset")
objCOTAN <- COTAN(raw = test.dataset)
objCOTAN <- clean(objCOTAN)
objCOTAN <- estimateDispersionBisection(objCOTAN, cores = 12)

data("test.dataset.clusters1")
clusters <- test.dataset.clusters1

coexDF <- DEAOnClusters(objCOTAN, clusters = clusters)

groupMarkers <- list(G1 = c("g-000010", "g-000020", "g-000030"),
                     G2 = c("g-000300", "g-000330"),
                     G3 = c("g-000510", "g-000530", "g-000550",
                            "g-000570", "g-000590"))

umapPlot <- UMAPPlot(coexDF, clusters = NULL, elements = groupMarkers)
plot(umapPlot)

objCOTAN <- addClusterization(objCOTAN, clName = "first_clusterization",
                              clusters = clusters, coexDF = coexDF)

objCOTAN <- estimateNuLinearByCluster(objCOTAN, clusters = clusters)

clSummaryPlotAndData <-
  clustersSummaryPlot(objCOTAN, clName = "first_clusterization",
                      plotTitle = "first clusterization")
##plot(clSummaryPlotAndData[["plot"]])

##objCOTAN <- dropClusterization(objCOTAN, "first_clusterization")

clusterizations <- getClusterizations(objCOTAN, dropNoCoex = TRUE)

enrichment <- geneSetEnrichment(clustersCoex = coexDF,
                                groupMarkers = groupMarkers)

clHeatmapPlotAndData <- clustersMarkersHeatmapPlot(objCOTAN, groupMarkers)
##plot(clHeatmapPlotAndData[["heatmapPlot"]])
```

```
conditions <- as.integer(substring(getCells(objCOTAN), 3L))
conditions <- factor(ifelse(conditions <= 600, "L", "H"))
names(conditions) <- getCells(objCOTAN)

clHeatmapPlotAndData2 <-
  clustersMarkersHeatmapPlot(objCOTAN, groupMarkers, kCuts = 2,
                             condNameList = list("High/Low"),
                             conditionsList = list(conditions))
##plot(clHeatmapPlotAndData2[["heatmapPlot"]])

clName <- getClusterizationName(objCOTAN)

clusterDataList <- getClusterizationData(objCOTAN, clName = clName)

clusters <- getClusters(objCOTAN, clName = clName)

allClustersCoexDF <- getClustersCoex(objCOTAN)


deltaExpression <- clustersDeltaExpression(objCOTAN, clusters = clusters)

summaryData <- clustersSummaryData(objCOTAN)

treePlotAndObj <- clustersTreePlot(objCOTAN, 2)
objCOTAN <- treePlotAndObj[["objCOTAN"]]
plot(treePlotAndObj[["dend"]])

clMarkers <- findClustersMarkers(objCOTAN, clusters = clusters)
```

`funProbZero` *funProbZero*

Description

Private function that gives the probability of a sample gene count being zero given the given the dispersion and mu

Usage

```
funProbZero(disp, mu)
```

Arguments

`disp`	the estimated dispersion (can be a n-sized vector)
`mu`	the lambda times nu value (can be a $n \times m$ matrix)

Details

Using d for `disp` and μ for `mu`, it returns: $(1 + d\mu)^{-\frac{1}{d}}$ when $d > 0$ and $\exp((d-1)\mu)$ otherwise. The function is continuous in $d = 0$, increasing in d and decreasing in μ. It returns 0 when $d = -\infty$ or $\mu = \infty$. It returns 1 when $\mu = 0$.

Value

the probability (matrix) that a count is identically zero

`GenesCoexSpace` *Local Differentiation Index*

Description

To make the `GDI` more specific, it may be desirable to restrict the set of genes against which `GDI` is computed to a selected subset, with the recommendation to include a consistent fraction of cell-identity genes, and possibly focusing on markers specific for the biological question of interest (for instance neural cortex layering markers). In this case we denote it as *Local Differentiation Index* (LDI) relative to the selected subset.

Usage

```
genesCoexSpace(objCOTAN, primaryMarkers, numGenesPerMarker = 25L)

establishGenesClusters(
  objCOTAN,
  groupMarkers,
  numGenesPerMarker = 25L,
  kCuts = 6L,
  distance = "cosine",
  hclustMethod = "ward.D2"
)
```

Arguments

`objCOTAN`	a `COTAN` object
`primaryMarkers`	A vector of primary marker names.
`numGenesPerMarker`	the number of correlated genes to keep as other markers (default 25)
`groupMarkers`	a named `list` with an element for each group comprised of one or more marker genes
`kCuts`	the number of estimated *cluster* (this defines the height for the tree cut)
`distance`	type of distance to use (default is `"cosine"`, `"euclidean"` and the others from `parallelDist::parDist()` are also available)
`hclustMethod`	default is "ward.D2" but can be any method defined by `stats::hclust()` function

Details

`genesCoexSpace()` calculates genes groups based on the primary markers and uses them to prepare the genes' `COEX` space `data.frame`.

`establishGenesClusters()` perform the genes' clustering based on a pool of gene markers, using the genes' `COEX` space

Value

`genesCoexSpace()` returns a `list` with:

- `"SecondaryMarkers"` a named `list` that for each secondary marker, gives the `list` of primary markers that selected for it
- `"GCS"` the `COEX` `data.frame`
- `"rankGenes"` a `data.frame` with the rank of each gene according to its *p-value*

`establishGenesClusters()` a `list` of:

- `"g.space"` the genes' `COEX` space `data.frame`
- `"plot.eig"` the eigenvalues plot
- `"pca_clusters"` the *pca* components `data.frame`
- `"tree_plot"` the tree plot for the genes' `COEX` space

Examples

```
data("test.dataset")
objCOTAN <- COTAN(raw = test.dataset)
objCOTAN <- proceedToCoex(objCOTAN, cores = 12, saveObj = FALSE)

markers <- getGenes(objCOTAN)[sample(getNumGenes(objCOTAN), 10)]
GCS <- genesCoexSpace(objCOTAN, primaryMarkers = markers,
                      numGenesPerMarker = 15)

groupMarkers <- list(G1 = c("g-000010", "g-000020", "g-000030"),
                     G2 = c("g-000300", "g-000330"),
                     G3 = c("g-000510", "g-000530", "g-000550",
                            "g-000570", "g-000590"))

resList <-  establishGenesClusters(objCOTAN, groupMarkers = groupMarkers,
                                   numGenesPerMarker = 11)
```

`GenesStatistics` *Calculations of genes statistics*

Description

A collection of functions returning various statistics associated to the genes. In particular the *discrepancy* between the expected probabilities of zero and their actual occurrences, both at single gene level or looking at genes' pairs

Usage

```
calculateGenesCE(objCOTAN)

calculateGDI(objCOTAN, statType = "S")

calculatePValue(
  objCOTAN,
  statType = "S",
  geneSubsetCol = vector(mode = "character"),
  geneSubsetRow = vector(mode = "character")
)

calculatePDI(
  objCOTAN,
  statType = "S",
  geneSubsetCol = vector(mode = "character"),
  geneSubsetRow = vector(mode = "character")
)
```

Arguments

`objCOTAN`	a `COTAN` object
`statType`	Which statistics to use to compute the p-values. By default it will use the "S" (Pearson's χ^2 test) otherwise the "G" (G-test)
`geneSubsetCol`	an array of genes. It will be put in columns. If left empty the function will do it genome-wide.
`geneSubsetRow`	an array of genes. It will be put in rows. If left empty the function will do it genome-wide.

Details

`calculateGenesCE()` is used to calculate the discrepancy between the expected probability of zero and the observed zeros across all cells for each gene as *cross-entropy*: $-\sum_c \mathbb{1}_{X_c==0}\log(p_c) - \mathbb{1}_{X_c!=0}\log(1-p_c)$ where X_c is the observed count and p_c the probability of zero

`calculateGDI()` produces a `data.frame` with the *GDI* for each gene based on the `COEX` matrix `calculatePValue()` computes the p-values for genes in the `COTAN` object. It can be used genome-wide or by setting some specific genes of interest. By default it computes the *p-values* using the `S` statistics (χ^2)

`calculatePDI()` computes the p-values for genes in the `COTAN` object using `calculatePValue()` and takes their $\log\left(-\log\left(\cdot\right)\right)$ to calculate the genes' *Pair Differential Index*

Value

`calculateGenesCE()` returns a named `array` with the *cross-entropy* of each gene

`calculateGDI()` returns a `data.frame` with the *GDI* data

`calculatePValue()` returns a *p-value* `matrix` as `dspMatrix`

`calculatePDI()` returns a *Pair Differential Index* `matrix` as `dspMatrix`

`getColorsVector` *getColorsVector*

Description

This function returns a list of colors based on the `brewer.pal()` function

Usage

```
getColorsVector(numNeededColors = 0L)
```

Arguments

`numNeededColors`	The number of returned colors. If omitted it returns all available colors

Details

The colors are taken from the `brewer.pal.info()` sets with `Set1`, `Set2`, `Set3` placed first.

Value

an array of `RGB` colors of the wanted size

Examples

```
colorsVector <- getColorsVector(17)
```

`HandleMetaData` *Handling* meta-data *in* `COTAN` *objects*

Description

Much of the information stored in the `COTAN` object is compacted into three `data.frames`:

- `"metaDataset"` - contains all general information about the data-set
- `"metaGenes"` - contains genes' related information along the `lambda` and `dispersion` vectors and the fully-expressed flag
- `"metaCells"` - contains cells' related information along the `nu` vector, the fully-expressing flag, the *clusterizations* and the *conditions*

Usage

```
## S4 method for signature 'COTAN'
getMetadataDataset(objCOTAN)

## S4 method for signature 'COTAN'
getMetadataElement(objCOTAN, tag)

## S4 method for signature 'COTAN'
getMetadataGenes(objCOTAN)

## S4 method for signature 'COTAN'
getMetadataCells(objCOTAN)

## S4 method for signature 'COTAN'
getDims(objCOTAN)

datasetTags()

## S4 method for signature 'COTAN'
initializeMetaDataset(objCOTAN, GEO, sequencingMethod, sampleCondition)

## S4 method for signature 'COTAN'
addElementToMetaDataset(objCOTAN, tag, value)

setColumnInDF(df, colToSet, colName, rowNames = vector(mode = "character"))
```

Arguments

`objCOTAN`	a `COTAN` object
`tag`	the new information tag
`GEO`	a code reporting the GEO identification or other specific data-set code

`sequencingMethod`	a string reporting the method used for the sequencing
`sampleCondition`	a string reporting the specific sample condition or time point
`value`	a value (or an array) containing the information
`df`	the `data.frame`
`colToSet`	the the column to add
`colName`	the name of the new or existing column in the `data.frame`
`rowNames`	when not empty, if the input `data.frame` has no real row names, the new row names of the resulting `data.frame`

Details

`getMetadataDataset()` extracts the meta-data stored for the current data-set.

`getMetadataElement()` extracts the value associated with the given tag if present or an empty string otherwise.

`getMetadataGenes()` extracts the meta-data stored for the genes

`getMetadataCells()` extracts the meta-data stored for the cells

`getDims()` extracts the sizes of all slots of the `COTAN` object

`datasetTags()` defines a list of short names associated to an enumeration. It also defines the relative long names as they appear in the meta-data

`initializeMetaDataset()` initializes meta-data data-set

`addElementToMetaDataset()` is used to add a line of information to the meta-data `data.frame`. If the tag was already used it will update the associated value(s) instead

`setColumnInDF()` is a function to append, if missing, or resets, if present, a column into a `data.frame`, whether the `data.frame` is empty or not. The given `rowNames` are used only in the case the `data.frame` has only the default row numbers, so this function cannot be used to override row names

Value

`getMetadataDataset()` returns the meta-data `data.frame`

`getMetadataElement()` returns a string with the relevant value

`getMetadataGenes()` returns the genes' meta-data `data.frame`

`getMetadataCells()` returns the cells' meta-data `data.frame`

`getDims()` returns a named `list` with the sizes of the slots

`datasetTags()` a named `character array` with the standard labels used in the `metaDataset` of the `COTAN` objects

`initializeMetaDataset()` returns the given `COTAN` object with the updated `metaDataset`

`addElementToMetaDataset()` returns the updated `COTAN` object

`setColumnInDF()` returns the updated, or the newly created, `data.frame`

Examples

```
data("test.dataset")
objCOTAN <- COTAN(raw = test.dataset)

objCOTAN <- initializeMetaDataset(objCOTAN, GEO = "test_GEO",
                                  sequencingMethod = "distribution_sampling",
                                  sampleCondition = "reconstructed_dataset")

objCOTAN <- addElementToMetaDataset(objCOTAN, "Test",
                                    c("These are ", "some values"))

dataSetInfo <- getMetadataDataset(objCOTAN)

numInitialCells <- getMetadataElement(objCOTAN, "cells")

metaGenes <- getMetadataGenes(objCOTAN)

metaCells <- getMetadataCells(objCOTAN)

allSizes <- getDims(objCOTAN)
```

`HandlingConditions` *Handling cells'* conditions *and related functions*

Description

These functions manage the *conditions*.

A *condition* is a set of **labels** that can be assigned to cells: one **label** per cell. This is especially useful in cases when the `data-set` is the result of merging multiple experiments' raw data

Usage

```
## S4 method for signature 'COTAN'
getAllConditions(objCOTAN, keepPrefix = FALSE)

## S4 method for signature 'COTAN'
getConditionName(objCOTAN, condName = "", keepPrefix = FALSE)

## S4 method for signature 'COTAN'
getCondition(objCOTAN, condName = "")

normalizeNameAndLabels(objCOTAN, name = "", labels = NULL, isCond = FALSE)

## S4 method for signature 'COTAN'
addCondition(objCOTAN, condName, conditions, override = FALSE)

## S4 method for signature 'COTAN'
dropCondition(objCOTAN, condName)
```

Arguments

`objCOTAN`	a `COTAN` object
`keepPrefix`	When `TRUE` returns the internal name of the *condition*: the one with the `COND_` prefix.
`condName`	the name of an existing *condition*.
`name`	the name of the *clusterization*/*condition*. If not given the last available *clusterization* will be used, or no *conditions*
`labels`	a *clusterization*/*condition* to use. If given it will take precedence on the one indicated by `name`
`isCond`	a Boolean to indicate whether the function is dealing with *clusterizations* `FALSE` or *conditions* `TRUE`
`conditions`	a (factors) array of *condition* **labels**
`override`	When `TRUE` silently allows overriding data for an existing *condition* name. Otherwise the default behavior will avoid potential data losses

Details

`getAllConditions()` extracts the list of the *conditions* defined in the `COTAN` object.

`getConditionName()` normalizes the given *condition* name or, if none were given, returns the name of last available *condition* in the `COTAN` object. It can return the *condition* **internal name** if needed

`getCondition()` extracts the asked *condition* from the `COTAN` object

`normalizeNameAndLabels()` takes a pair of name/labels and normalize them based on the available information in the `COTAN` object

`addCcondition()` adds a *condition* to the current `COTAN` object, by adding a new column in the `metaCells` `data.frame`

`dropCondition()` drops a *condition* from the current `COTAN` object, by removing the corresponding column in the `metaCells` `data.frame`

Value

`getAllConditions()` returns a vector of *conditions* names, usually without the `COND_` prefix

`getConditionName()` returns the normalized *condition* name or `NULL` if no *conditions* are present

`getCondition()` returns a named `factor` with the *condition*

`normalizeNameAndLabels()` returns a `list` with:

- `"name"` the relevant name
- `"labels"` the relevant *clusterization*/*condition*

`addCondition()` returns the updated `COTAN` object

`dropCondition()` returns the updated `COTAN` object

Examples

```
data("test.dataset")
objCOTAN <- COTAN(raw = test.dataset)

genre <- rep(c("F", "M"), getNumCells(objCOTAN) / 2)
objCOTAN <- addCondition(objCOTAN, condName = "Genre", conditions = genre)

##objCOTAN <- dropCondition(objCOTAN, "Genre")

conditionsNames <- getAllConditions(objCOTAN)

condName <- getConditionName(objCOTAN)

condition <- getCondition(objCOTAN, condName = condName)
isa(condition, "factor")

nameAndCond <- normalizeNameAndLabels(objCOTAN, name = condName,
                                      isCond = TRUE)
isa(nameAndCond[["labels"]], "factor")
```

`HeatmapPlots` *Heatmap Plots*

Description

These functions create heatmap `COEX` plots.

Usage

```
heatmapPlot(genesLists, sets, conditions, dir, pValueThreshold = 0.01)

genesHeatmapPlot(
  objCOTAN,
  primaryMarkers,
  secondaryMarkers = vector(mode = "character"),
  pValueThreshold = 0.01,
  symmetric = TRUE
)

cellsHeatmapPlot(objCOTAN, cells = NULL, clusters = NULL)

plotTheme(plotKind = "common", textSize = 14L)
```

Arguments

`genesLists`	A `list` of genes' `arrays`. The first `array` defines the genes in the columns
`sets`	A numeric array indicating which fields in the previous `list` should be used

`conditions`	An array of prefixes indicating the different files
`dir`	The directory in which are all `COTAN` files (corresponding to the previous prefixes)
`pValueThreshold`	The p-value threshold. Default is 0.01
`objCOTAN`	a `COTAN` object
`primaryMarkers`	A set of genes plotted as rows
`secondaryMarkers`	A set of genes plotted as columns
`symmetric`	A Boolean: default `TRUE`. When `TRUE` the union of `primaryMarkers` and `secondaryMarkers` is used for both rows and column genes
`cells`	Which cells to plot (all if no argument is given)
`clusters`	Use this clusterization to select/reorder the cells to plot
`plotKind`	a string indicating the plot kind
`textSize`	axes and strip text size (default=14)

Details

`heatmapPlot()` creates the `heatmap` of one or more `COTAN` objects

`genesHeatmapPlot()` is used to plot an *heatmap* made using only some genes, as markers, and collecting all other genes correlated with these markers with a p-value smaller than the set threshold. Than all relations are plotted. Primary markers will be plotted as groups of rows. Markers list will be plotted as columns.

`cellsHeatmapPlot()` creates the heatmap plot of the cells' `COEX` matrix

`plotTheme()` returns the appropriate theme for the selected plot kind. Supported kinds are: `"common"`, `"pca"`, `"genes"`, `"UDE"`, `"heatmap"`, `"GDI"`, `"UMAP"`, `"size-plot"`

Value

`heatmapPlot()` returns a `ggplot2` object

`genesHeatmapPlot()` returns a `ggplot2` object

`cellsHeatmapPlot()` returns the cells' `COEX` *heatmap* plot

`plotTheme()` returns a `ggplot2::theme` object

See Also

`ggplot2::theme()` and `ggplot2::ggplot()`

Examples

```
data("test.dataset")
objCOTAN <- COTAN(raw = test.dataset)
objCOTAN <- clean(objCOTAN)
objCOTAN <- estimateDispersionNuBisection(objCOTAN, cores = 12)
objCOTAN <- calculateCoex(objCOTAN, actOnCells = FALSE)
```

```
objCOTAN <- calculateCoex(objCOTAN, actOnCells = TRUE)

## Save the `COTAN` object to file
data_dir <- tempdir()
saveRDS(objCOTAN, file = file.path(data_dir, "test.dataset.cotan.RDS"))

## some genes
primaryMarkers <- c("g-000010", "g-000020", "g-000030")

## an example of named list of different gene set
groupMarkers <- list(G1 = primaryMarkers,
                     G2 = c("g-000300", "g-000330"),
                     G3 = c("g-000510", "g-000530", "g-000550",
                            "g-000570", "g-000590"))

hPlot <- heatmapPlot(genesLists = groupMarkers, sets = c(2, 3),
                     pValueThreshold = 0.05, conditions = c("test.dataset"),
                     dir = paste0(data_dir, "/"))
plot(hPlot)

ghPlot <- genesHeatmapPlot(objCOTAN, primaryMarkers = primaryMarkers,
                           secondaryMarkers = groupMarkers,
                           pValueThreshold = 0.05, symmetric = FALSE)
plot(ghPlot)

clusters <- c(rep_len("1", getNumCells(objCOTAN)/2),
              rep_len("2", getNumCells(objCOTAN)/2))
names(clusters) <- getCells(objCOTAN)

chPlot <- cellsHeatmapPlot(objCOTAN, clusters = clusters)
plot(chPlot)

theme <- plotTheme("pca")
```

`LegacyFastSymmMatrix` *Handle symmetric matrix <-> vector conversions*

Description

Converts a symmetric matrix into a compacted symmetric matrix and vice-versa.

Usage

```
vec2mat_rfast(x, genes = "all")

mat2vec_rfast(mat)
```

Arguments

`x`	a `list` formed by two arrays: `genes` with the unique gene names and `values` with all the values.
`genes`	an array with all wanted genes or the string `"all"`. When equal to `"all"` (the default), it recreates the entire matrix.
`mat`	a square (possibly symmetric) matrix with all genes as row and column names.

Details

This is a legacy function related to old `scCOTAN` objects. Use the more appropriate `Matrix::dspMatrix` type for similar functionality.

`mat2vec_rfast`will forcibly make its argument symmetric.

Value

`vec2mat_rfast` returns the reconstructed symmetric matrix

`mat2vec_rfast` a `list` formed by two arrays:

- `genes` with the unique gene names,
- `values` with all the values.

Examples

```
v <- list("genes" = paste0("gene_", c(1:9)), "values" = c(1:45))

M <- vec2mat_rfast(v)
all.equal(rownames(M), v[["genes"]])
all.equal(colnames(M), v[["genes"]])

genes <- paste0("gene_", sample.int(ncol(M), 3))

m <- vec2mat_rfast(v, genes)
all.equal(rownames(m), v[["genes"]])
all.equal(colnames(m), genes)

v2 <- mat2vec_rfast(M)
all.equal(v, v2)
```

`LoggingFunctions` *Logging in the* `COTAN` *package*

Description

Logging is currently supported for all `COTAN` functions. It is possible to see the output on the terminal and/or on a log file. The level of output on terminal is controlled by the `COTAN.LogLevel` option while the logging on file is always at its maximum verbosity

Usage

```
setLoggingLevel(newLevel = 1L)

setLoggingFile(logFileName)

logThis(msg, logLevel = 2L, appendLF = TRUE)
```

Arguments

`newLevel`	the new default logging level. It defaults to 1
`logFileName`	the log file.
`msg`	the message to print
`logLevel`	the logging level of the current message. It defaults to 2
`appendLF`	whether to add a new-line character at the end of the message

Details

`setLoggingLevel()` sets the `COTAN` logging level. It set the `COTAN.LogLevel` options to one of the following values:

- 0 - Always on log messages
- 1 - Major log messages
- 2 - Minor log messages
- 3 - All log messages

`setLoggingFile()` sets the log file for all `COTAN` output logs. By default no logging happens on a file (only on the console). Using this function `COTAN` will use the indicated file to dump the logs produced by all `logThis()` commands, independently from the log level. It stores the `connection` created by the call to `bzfile()` in the option: `COTAN.LogFile`

`logThis()` prints the given message string if the current log level is greater or equal to the given log level (it always prints its message on file if active). It uses `message()` to actually print the messages on the `stderr()` connection, so it is subject to `suppressMessages()`

Value

`setLoggingLevel()` returns the old logging level or default level if not set yet.

`logThis()` returns TRUE if the message has been printed on the terminal

Examples

```
setLoggingLevel(3) # for debugging purposes only

setLoggingFile("./COTAN_Test1.log") # for debugging purposes only
logThis("Some log message")
setLoggingFile("") # closes the log file

logThis("LogLevel 0 messages will always show, ",
```

```
        logLevel = 0, appendLF = FALSE)
suppressMessages(logThis("unless all messages are suppressed",
                         logLevel = 0))
```

`ParametersEstimations` *Estimation of the* `COTAN` *model's parameters*

Description

These functions are used to estimate the `COTAN` model's parameters. That is the average count for each gene (lambda) the average count for each cell (nu) and the dispersion parameter for each gene to match the probability of zero.

The estimator methods are named `Linear` if they can be calculated as a linear statistic of the raw data or `Bisection` if they are found via a parallel bisection solver.

Usage

```
## S4 method for signature 'COTAN'
estimateLambdaLinear(objCOTAN)

## S4 method for signature 'COTAN'
estimateNuLinear(objCOTAN)

## S4 method for signature 'COTAN'
estimateDispersionBisection(
  objCOTAN,
  threshold = 0.001,
  cores = 1L,
  maxIterations = 100L,
  chunkSize = 1024L
)

## S4 method for signature 'COTAN'
estimateNuBisection(
  objCOTAN,
  threshold = 0.001,
  cores = 1L,
  maxIterations = 100L,
  chunkSize = 1024L
)

## S4 method for signature 'COTAN'
estimateDispersionNuBisection(
  objCOTAN,
  threshold = 0.001,
  cores = 1L,
```

```
  maxIterations = 100L,
  chunkSize = 1024L,
  enforceNuAverageToOne = TRUE
)

## S4 method for signature 'COTAN'
estimateDispersionNuNlminb(
  objCOTAN,
  threshold = 0.001,
  maxIterations = 50L,
  chunkSize = 1024L,
  enforceNuAverageToOne = TRUE
)

## S4 method for signature 'COTAN'
getNormalizedData(objCOTAN)

## S4 method for signature 'COTAN'
getNu(objCOTAN)

## S4 method for signature 'COTAN'
getLambda(objCOTAN)

## S4 method for signature 'COTAN'
getDispersion(objCOTAN)
```

Arguments

`objCOTAN`	a `COTAN` object
`threshold`	minimal solution precision
`cores`	number of cores to use. Default is 1.
`maxIterations`	max number of iterations (avoids infinite loops)
`chunkSize`	number of genes to solve in batch in a single core. Default is 1024.
`enforceNuAverageToOne`	a Boolean on whether to keep the average `nu` equal to 1

Details

`estimateLambdaLinear()` does a linear estimation of lambda (genes' counts averages)

`estimateNuLinear()` does a linear estimation of nu (normalized cells' counts averages)

`estimateDispersionBisection()` estimates the negative binomial dispersion factor for each gene (a). Determines the `dispersion` such that, for each gene, the probability of zero count matches the number of observed zeros. It assumes `estimateNuLinear()` being already run.

`estimateNuBisection()` estimates the `nu` vector of a `COTAN` object by bisection. It determines the `nu` parameters such that, for each cell, the probability of zero counts matches the number of observed zeros. It assumes `estimateDispersionBisection()` being already run. Since this breaks the assumption that the average `nu` is `1`, it is recommended not to run this in isolation but use `estimateDispersionNuBisection()` instead.

`estimateDispersionNuBisection()` estimates the `dispersion` and `nu` field of a `COTAN` object by running sequentially a bisection for each parameter.

`estimateDispersionNuNlminb()` estimates the `nu` and `dispersion` parameters to minimize the discrepancy between the observed and expected probability of zero. It uses the `stats::nlminb()` solver, but since the joint parameters have too high dimensionality, it converges too slowly to be actually useful in real cases.

`getNormalizedData()` extracts the *normalized* count table (i.e. divided by `nu`)

`getNu()` extracts the nu array (normalized cells' counts averages)

`getLambda()` extracts the lambda array (mean expression for each gene)

`getDispersion()` extracts the dispersion array (a)

Value

`estimateLambdaLinear()` returns the updated `COTAN` object

`estimateNuLinear()` returns the updated `COTAN` object

`estimateDispersionBisection()` returns the updated `COTAN` object

`estimateNuBisection()` returns the updated `COTAN` object

`estimateDispersionNuBisection()` returns the updated `COTAN` object

`estimateDispersionNuNlminb()` returns the updated `COTAN` object

`getNormalizedData()` returns the normalized count `data.frame`

`getNu()` returns the nu array

`getLambda()` returns the lambda array

`getDispersion()` returns the dispersion array

Examples

```
data("test.dataset")
objCOTAN <- COTAN(raw = test.dataset)

objCOTAN <- estimateLambdaLinear(objCOTAN)
lambda <- getLambda(objCOTAN)

objCOTAN <- estimateNuLinear(objCOTAN)
nu <- getNu(objCOTAN)

objCOTAN <- estimateDispersionBisection(objCOTAN, cores = 12)
dispersion <- getDispersion(objCOTAN)

objCOTAN <- estimateDispersionNuBisection(objCOTAN, cores = 12,
                                          enforceNuAverageToOne = TRUE)
nu <- getNu(objCOTAN)
dispersion <- getDispersion(objCOTAN)

rawNorm <- getNormalizedData(objCOTAN)
```

`RawDataCleaning` *Raw data cleaning*

Description

These methods are to be used to clean the raw data. That is drop any number of genes/cells that are too sparse or too present to allow proper calibration of the `COTAN` model.

We call genes that are expressed in all cells *Fully-Expressed* while cells that express all genes in the data are called *Fully-Expressing*. In case it has been made quite easy to exclude the flagged genes/cells in the user calculations.

Usage

```
## S4 method for signature 'COTAN'
flagNotFullyExpressedGenes(objCOTAN)

## S4 method for signature 'COTAN'
flagNotFullyExpressingCells(objCOTAN)

## S4 method for signature 'COTAN'
getFullyExpressedGenes(objCOTAN)

## S4 method for signature 'COTAN'
getFullyExpressingCells(objCOTAN)

## S4 method for signature 'COTAN'
findFullyExpressedGenes(objCOTAN, cellsThreshold = 0.99)

## S4 method for signature 'COTAN'
findFullyExpressingCells(objCOTAN, genesThreshold = 0.99)

## S4 method for signature 'COTAN'
dropGenesCells(
  objCOTAN,
  genes = vector(mode = "character"),
  cells = vector(mode = "character")
)

ECDPlot(objCOTAN, yCut)

## S4 method for signature 'COTAN'
clean(
  objCOTAN,
  cellsCutoff = 0.003,
  genesCutoff = 0.002,
```

```
  cellsThreshold = 0.99,
  genesThreshold = 0.99
)

cleanPlots(objCOTAN, includePCA = TRUE)

cellSizePlot(objCOTAN, splitPattern = " ", numCol = 2L)

genesSizePlot(objCOTAN, splitPattern = " ", numCol = 2L)

mitochondrialPercentagePlot(
  objCOTAN,
  splitPattern = " ",
  numCol = 2L,
  genePrefix = "^MT-"
)

scatterPlot(objCOTAN, splitPattern = " ", numCol = 2L, splitSamples = FALSE)
```

Arguments

`objCOTAN`	a `COTAN` object
`cellsThreshold`	any gene that is expressed in more cells than threshold times the total number of cells will be marked as **fully-expressed**. Default threshold is 0.99 (99.0%)
`genesThreshold`	any cell that is expressing more genes than threshold times the total number of genes will be marked as **fully-expressing**. Default threshold is 0.99 (99.0%)
`genes`	an array of gene names
`cells`	an array of cell names
`yCut`	y threshold of library size to drop
`cellsCutoff`	`clean()` will delete from the `raw` data any gene that is expressed in less cells than threshold times the total number of cells. Default cutoff is 0.003 (0.3%)
`genesCutoff`	`clean()` will delete from the `raw` data any cell that is expressing less genes than threshold times the total number of genes. Default cutoff is 0.002 (0.2%)
`includePCA`	a `Boolean` flag to determine whether to calculate the *PCA* associated with the normalized matrix. When `TRUE` the first four elements of the returned list will be `NULL`
`splitPattern`	Pattern used to extract, from the column names, the sample field (default " ")
`numCol`	Once the column names are split by splitPattern, the column number with the sample name (default 2)
`genePrefix`	Prefix for the mitochondrial genes (default "^MT-" for Human, mouse "^mt-")
`splitSamples`	Boolean. Whether to plot each sample in a different panel (default `FALSE`)

Details

`flagNotFullyExpressedGenes()` returns a Boolean array with TRUE for those genes that are not fully-expressed.

`flagNotFullyExpressingCells()`returns a Boolean vector with TRUE for those cells that are not expressing all genes

`getFullyExpressedGenes()` returns the genes expressed in all cells of the dataset

`getFullyExpressingCells()` returns the cells that did express all genes of the dataset

`findFullyExpressedGenes()` determines the fully-expressed genes inside the raw data

`findFullyExpressingCells()` determines the cells that are expressing all genes in the dataset

`dropGenesCells()` removes an array of genes and/or cells from the current `COTAN` object.

`ECDPlot()` plots the empirical distribution function of library sizes (UMI number). It helps to define where to drop "cells" that are simple background signal.

`clean()` is the main method that can be used to check and clean the dataset. It will discard any genes that has less than 3 non-zero counts per thousand cells and all cells expressing less than 2 per thousand genes. It also produces and stores the estimators for nu and lambda

`cleanPlots()` creates the plots associated to the output of the `clean()` method.

`cellSizePlot()` plots the raw library size for each cell and sample.

`genesSizePlot()` plots the raw gene number (reads > 0) for each cell and sample

`mitochondrialPercentagePlot()` plots the raw library size for each cell and sample.

`scatterPlot()` creates a plot that check the relation between the library size and the number of genes detected.

Value

`flagNotFullyExpressedGenes()` returns a Booleans array with TRUE for genes that are not fully-expressed

`flagNotFullyExpressingCells()` returns an array of Booleans with TRUE for cells that are not expressing all genes

`getFullyExpressedGenes()` returns an array containing all genes that are expressed in all cells

`getFullyExpressingCells()` returns an array containing all cells that express all genes

`findFullyExpressedGenes()` returns the given `COTAN` object with updated **fully-expressed** genes' information

`findFullyExpressingCells()` returns the given `COTAN` object with updated **fully-expressing** cells' information

`dropGenesCells()` returns a completely new `COTAN` object with the new raw data obtained after the indicated genes/cells were expunged. All remaining data is dropped too as no more relevant with the restricted matrix. Exceptions are:

- the meta-data for the data-set that gets kept unchanged
- the meta-data of genes/cells that gets restricted to the remaining elements. The columns calculated via `estimate` and `find` methods are dropped too

`ECDPlot()` returns an ECD plot

`clean()` returns the updated `COTAN` object

`cleanPlots()` returns a `list` of `ggplot2` plots:

- `"pcaCells"` is for pca cells
- `"pcaCellsData"` is the data of the pca cells (can be plotted)
- `"genes"` is for `B` group cells' genes
- `"UDE"` is for cells' UDE against their pca
- `"nu"` is for cell *nu*
- `"zoomedNu"` is the same but zoomed on the left and with an estimate for the low *nu* threshold that defines problematic cells

`cellSizePlot()` returns the `violin-boxplot` plot

`genesSizePlot()` returns the `violin-boxplot` plot

`mitochondrialPercentagePlot()` returns a `list` with:

- `"plot"` a `violin-boxplot` object
- `"sizes"` a sizes `data.frame`

`scatterPlot()` returns the scatter plot

Examples

```
library(zeallot)

data("test.dataset")
objCOTAN <- COTAN(raw = test.dataset)

genes.to.rem <- getGenes(objCOTAN)[grep('^MT', getGenes(objCOTAN))]
cells.to.rem <- getCells(objCOTAN)[which(getCellsSize(objCOTAN) == 0)]
objCOTAN <- dropGenesCells(objCOTAN, genes.to.rem, cells.to.rem)

objCOTAN <- clean(objCOTAN)

objCOTAN <- findFullyExpressedGenes(objCOTAN)
goodPos <- flagNotFullyExpressedGenes(objCOTAN)

objCOTAN <- findFullyExpressingCells(objCOTAN)
goodPos <- flagNotFullyExpressingCells(objCOTAN)

feGenes <- getFullyExpressedGenes(objCOTAN)

feCells <- getFullyExpressingCells(objCOTAN)

## These plots might help to identify genes/cells that need to be dropped
ecdPlot <- ECDPlot(objCOTAN, yCut = 100)
plot(ecdPlot)

# This creates many infomative plots useful to determine whether
# there is still something to drop...
# Here we use the tuple-like assignment feature of the `zeallot` package
c(pcaCellsPlot, ., genesPlot, UDEPlot, ., zNuPlot) %<-% cleanPlots(objCOTAN)
plot(pcaCellsPlot)
plot(UDEPlot)
```

```
plot(zNuPlot)

lsPlot <- cellSizePlot(objCOTAN)
plot(lsPlot)

gsPlot <- genesSizePlot(objCOTAN)
plot(gsPlot)

mitPercPlot <-
  mitochondrialPercentagePlot(objCOTAN, genePrefix = "g-0000")[["plot"]]
plot(mitPercPlot)

scPlot <- scatterPlot(objCOTAN)
plot(scPlot)
```

`RawDataGetters` *Raw data* `COTAN` *accessors*

Description

These methods extract information out of a just created `COTAN` object. The accessors have **read-only** access to the object.

Usage

```
## S4 method for signature 'COTAN'
getRawData(objCOTAN)

## S4 method for signature 'COTAN'
getNumCells(objCOTAN)

## S4 method for signature 'COTAN'
getNumGenes(objCOTAN)

## S4 method for signature 'COTAN'
getCells(objCOTAN)

## S4 method for signature 'COTAN'
getGenes(objCOTAN)

## S4 method for signature 'COTAN'
getZeroOneProj(objCOTAN)

## S4 method for signature 'COTAN'
getCellsSize(objCOTAN)

## S4 method for signature 'COTAN'
```

```
getNumExpressedGenes(objCOTAN)

## S4 method for signature 'COTAN'
getGenesSize(objCOTAN)

## S4 method for signature 'COTAN'
getNumOfExpressingCells(objCOTAN)
```

Arguments

`objCOTAN`	a `COTAN` object

Details

`getRawData()` extracts the raw count table.

`getNumCells()` extracts the number of cells in the sample (m)

`getNumGenes()` extracts the number of genes in the sample (n)

`getCells()` extract all cells in the dataset.

`getGenes()` extract all genes in the dataset.

`getZeroOneProj()` extracts the raw count table where any positive number has been replaced with `1`

`getCellsSize()` extracts the cell raw library size.

`getNumExpressedGenes()` extracts the number of genes expressed for each cell. Exploits a feature of Matrix::CsparseMatrix

`getGenesSize()` extracts the genes raw library size.

`getNumOfExpressingCells()` extracts, for each gene, the number of cells that are expressing it. Exploits a feature of Matrix::CsparseMatrix

Value

`getRawData()` returns the raw count sparse matrix

`getNumCells()` returns the number of cells in the sample (m)

`getNumGenes()` returns the number of genes in the sample (n)

`getCells()` returns a character array with the cells' names

`getGenes()` returns a character array with the genes' names

`getZeroOneProj()` returns the raw count matrix projected to `0` or `1`

`getCellsSize()` returns an array with the library sizes

`getNumExpressedGenes()` returns an array with the library sizes

`getGenesSize()` returns an array with the library sizes

`getNumOfExpressingCells()` returns an array with the library sizes

Examples

```
data("test.dataset")
objCOTAN <- COTAN(raw = test.dataset)

rawData <- getRawData(objCOTAN)

numCells <- getNumCells(objCOTAN)

numGenes <- getNumGenes(objCOTAN)

cellsNames <- getCells(objCOTAN)

genesNames <- getGenes(objCOTAN)

zeroOne <- getZeroOneProj(objCOTAN)

cellsSize <- getCellsSize(objCOTAN)

numExpGenes <- getNumExpressedGenes(objCOTAN)

genesSize <- getGenesSize(objCOTAN)

numExpCells <- getNumOfExpressingCells(objCOTAN)
```

`scCOTAN-class` *scCOTAN-class (for legacy usage)*

Description

Define `scCOTAN` structure

Value

a `scCOTAN` object

Slots

`raw` `ANY`. To store the raw data matrix

`raw.norm` `ANY`. To store the raw data matrix divided for the cell efficiency estimated (nu)

`coex` `ANY`. The coex matrix

`nu` `vector`.

`lambda` `vector`.

`a` `vector`.

`hk` `vector`.

`n_cells` `numeric`.

`meta` `data.frame`.

`yes_yes` `ANY`. Unused and deprecated. Kept for backward compatibility only

`clusters` `vector`.

`cluster_data` `data.frame`.

`UniformClusters` *Uniform Clusters*

Description

This group of functions takes in input a `COTAN` object and handle the task of dividing the dataset into **Uniform Clusters**, that is *clusters* that have an homogeneous genes' expression. This condition is checked by calculating the `GDI` of the *cluster* and verifying that no more than a small fraction of the genes have their `GDI` level above the given `GDIThreshold`

Usage

```
GDIPlot(
  objCOTAN,
  genes,
  condition = "",
  statType = "S",
  GDIThreshold = 1.4,
  GDIIn = NULL
)

cellsUniformClustering(
  objCOTAN,
  GDIThreshold = 1.4,
  cores = 1L,
  maxIterations = 25L,
  initialResolution = 0.8,
  distance = "cosine",
  hclustMethod = "ward.D2",
  saveObj = TRUE,
  outDir = "."
)

checkClusterUniformity(
  objCOTAN,
  cluster,
  cells,
  GDIThreshold = 1.4,
  cores = 1L,
  saveObj = TRUE,
  outDir = "."
```

```
)

mergeUniformCellsClusters(
  objCOTAN,
  clusters = NULL,
  GDIThreshold = 1.4,
  batchSize = 10L,
  cores = 1L,
  distance = "cosine",
  hclustMethod = "ward.D2",
  saveObj = TRUE,
  outDir = "."
)
```

Arguments

`objCOTAN`	a `COTAN` object
`genes`	a named `list` of genes to label. Each array will have different color.
`condition`	a string corresponding to the condition/sample (it is used only for the title).
`statType`	type of statistic to be used. Default is "S": Pearson's chi-squared test statistics. "G" is G-test statistics
`GDIThreshold`	the threshold level that discriminates uniform clusters. It defaults to 1.4
`GDIIn`	when the `GDI` data frame was already calculated, it can be put here to speed up the process (default is `NULL`)
`cores`	number cores used
`maxIterations`	max number of re-clustering iterations. It defaults to 25
`initialResolution`	a number indicating how refined are the clusters before checking for **uniformity**. It defaults to 0.8, the same as `Seurat::FindClusters()`
`distance`	type of distance to use (default is `"cosine"`, `"euclidean"` and the others from `parallelDist::parDist()` are also available)
`hclustMethod`	It defaults is `"ward.D2"` but can be any of the methods defined by the `stats::hclust()` function.
`saveObj`	Boolean flag; when `TRUE` saves intermediate analyses and plots to file
`outDir`	an existing directory for the analysis output. The effective output will be paced in a sub-folder.
`cluster`	the tag of the *cluster*
`cells`	the cells belonging to the *cluster*
`clusters`	The *clusterization* to merge. If not given the last available *clusterization* will be used, as it is probably the most significant!
`batchSize`	Number pairs to test in a single round. If none of them succeeds the merge stops

Details

`GDIPlot()` directly evaluates and plots the `GDI` for a sample.

`cellsUniformClustering()` finds a **Uniform** *clusterizations* by means of the `GDI`. Once a preliminary *clusterization* is obtained from the `Seurat-package` methods, each *cluster* is checked for **uniformity** via the function `checkClusterUniformity()`. Once all *clusters* are checked, all cells from the **non-uniform** clusters are pooled together for another iteration of the entire process, until all *clusters* are deemed **uniform**. In the case only a few cells are left out (≤ 50), those are flagged as `"-1"` and the process is stopped.

`checkClusterUniformity()` takes a `COTAN` object and a cells' *cluster* and checks whether the latter is **uniform** by `GDI`. The function runs `COTAN` to check whether the `GDI` is lower than the given `GDIThreshold` for the 99% of the genes. If the `GDI` results to be too high for too many genes, the *cluster* is deemed **non-uniform**.

`mergeUniformCellsClusters()` takes in a **uniform** *clusterization* and iteratively checks whether merging two *near clusters* would form a **uniform** *cluster* still. This function uses the *cosine distance* to establish the *nearest clusters pairs*. It will use the `checkClusterUniformity()` function to check whether the merged *clusters* are **uniform**. The function will stop once no *near pairs* of clusters are mergeable in a single batch

Value

`GDIPlot()` returns a ggplot2 object

`cellsUniformClustering()` returns a `list` with 2 elements:

- `"clusters"` the newly found cluster labels array
- `"coex"` the associated `COEX` `data.frame`

`checkClusterUniformity` returns a list with:

- `"isUniform"`: a flag indicating whether the *cluster* is **uniform**
- `"fractionAbove"`: the percentage of genes with `GDI` above the threshold
- `"1stPercentile"`: the quantile associated to the highest percentile

a `list` with:

- `"clusters"` the merged cluster labels array
- `"coex"` the associated `COEX` `data.frame`

Examples

```
data("test.dataset")

objCOTAN <- automaticCOTANObjectCreation(raw = test.dataset,
                                         GEO = "S",
                                         sequencingMethod = "10X",
                                         sampleCondition = "Test",
                                         cores = 12L,
                                         saveObj = FALSE)
```

```
groupMarkers <- list(G1 = c("g-000010", "g-000020", "g-000030"),
                     G2 = c("g-000300", "g-000330"),
                     G3 = c("g-000510", "g-000530", "g-000550",
                            "g-000570", "g-000590"))
gdiPlot <- GDIPlot(objCOTAN, genes = groupMarkers, cond = "test")
plot(gdiPlot)

## Here we override the default GDI threshold as a way to speed-up
## calculations as higher threshold implies less stringent uniformity
## It real applications it might be appropriate to change the threshold
## in cases of relatively low genes/cells number, or in cases when an
## rough clusterization is needed in the early satges of the analysis
##

splitList <- cellsUniformClustering(objCOTAN, cores = 12,
                                    initialResolution = 0.8,
                                    GDIThreshold = 1.5, saveObj = FALSE)

clusters <- splitList[["clusters"]]

firstCluster <- getCells(objCOTAN)[clusters %in% clusters[[1L]]]
checkClusterUniformity(objCOTAN,
                       GDIThreshold = 1.5,
                       cluster = clusters[[1L]],
                       cells = firstCluster,
                       cores = 12L,
                       saveObj = FALSE)

objCOTAN <- addClusterization(objCOTAN,
                              clName = "split",
                              clusters = clusters)

objCOTAN <- addClusterizationCoex(objCOTAN,
                                  clName = "split",
                                  coexDF = splitList[["coex"]])

identical(reorderClusterization(objCOTAN)[["clusters"]], clusters)

mergedList <- mergeUniformCellsClusters(objCOTAN,
                                        GDIThreshold = 1.5,
                                        batchSize = 5L,
                                        clusters = clusters,
                                        cores = 12L,
                                        distance = "cosine",
                                        hclustMethod = "ward.D2",
                                        saveObj = FALSE)

objCOTAN <- addClusterization(objCOTAN,
                              clName = "merged",
                              clusters = mergedList[["clusters"]],
                              coexDF = mergedList[["coex"]])

identical(reorderClusterization(objCOTAN), mergedList)
```

Index