Package ‘grandR’

February 15, 2024

Version 0.2.5

Title Comprehensive Analysis of Nucleotide Conversion Sequencing Data

Description Nucleotide conversion sequencing experiments have been developed to add a temporal dimension to RNA-seq and single-cell RNA-seq. Such experiments require specialized tools for primary processing such as GRAND-SLAM, (see 'Jürges et al' <doi:10.1093/bioinformatics/bty256>) and specialized tools for downstream analyses. 'grandR' provides a comprehensive toolbox for quality control, kinetic modeling, differential gene expression analysis and visualization of such data.

Author Florian Erhard [aut, cre] (<`https://orcid.org/0000-0002-3574-6983`>),
Teresa Rummel [ctb],
Lygeri Sakellaridi [ctb],
Kevin Berg [ctb]

Maintainer Florian Erhard <`Florian.Erhard@informatik.uni-regensburg.de`>

License Apache License (>= 2)

Encoding UTF-8

URL `https://github.com/erhard-lab/grandR`

BugReports `https://github.com/erhard-lab/grandR/issues`

Depends R (>= 3.5.0)

Imports stats, Matrix, rlang, ggplot2, grDevices, patchwork, RCurl, plyr, parallel, reshape2, MASS, scales, cowplot, minpack.lm, lfc, labeling, methods, utils, numDeriv

Suggests knitr, rmarkdown, circlize, Seurat, ComplexHeatmap, ggrepel, DESeq2, S4Vectors, data.table, clusterProfiler, biomaRt, msigdbr, fgsea, rclipboard, cubature, DT, shinyjs, shinyjqui, RColorBrewer, gsl, htmltools, matrixStats, monocle, VGAM, quantreg, graphics, shiny, ggrastr, viridisLite, deSolve

RoxygenNote 7.3.1

VignetteBuilder knitr

NeedsCompilation yes

Repository CRAN

Date/Publication 2024-02-15 15:30:02 UTC

R topics documented:

`Analyses` *Analysis table functions*

Description

Get analysis names and add or remove analyses

Usage

```
Analyses(data, description = FALSE)

AddAnalysis(
  data,
  name,
  table,
  by = NULL,
```

```
  warn.present = TRUE,
  warn.genes = TRUE
)

DropAnalysis(data, pattern = NULL)
```

Arguments

`data`	A grandR object
`description`	if TRUE, also return the column names of each analysis table (i.e. a list named according to the analyses)
`name`	The user-defined analysis name
`table`	The analysis table to add
`by`	Specify a column that contains gene names or symbols (see details)
`warn.present`	Warn if an analysis with the same name is already present (and then overwrite)
`warn.genes`	Warn if genes are missing (and then add NA)
`pattern`	A regular expression that is matched to analysis names

Details

The columns in the analysis tables are defined by the analysis method (e.g. "Synthesis","Half-life" and "rmse" by `FitKinetics`). A call to an analysis function might produce more than one table (e.g. because kinetic modeling is done for multiple Conditions). In this case, AddAnalysisTable produces more than one analysis table.

`AddAnalysis` is in most cases not called directly by the user, but is used by analysis methods to add their final result to a grandR object (e.g., FitKinetics,LikelihoodRatioTest,LFC,PairwiseDESeq2).

If it is called by the user (e.g. to add analysis results from external tools or from the literature, see pulse-chase vignette), then the user must make sure that either the rownames of the given table can be recognized as genes (names or symbols), or that there is a column in the table giving genes (this must be specified as the "by" parameter). The table does neither have to be sorted the same way the grandR object is, nor does it have to be complete. `AddAnalysis` will take care or reordering and inserting NA for missing genes (and it will issue a warning in case of missing genes).

Value

Either the analysis names or a grandR data with added/removed slots or the metatable to be used with AddAnalysis

Functions

- `Analyses()`: Obtain the analyses names
- `AddAnalysis()`: Add an analysis table
- `DropAnalysis()`: Remove analyses from the grandR object

See Also

Slots, DefaultSlot

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))

sars <- Normalize(sars)    # default behavior is to update the default slot; this calls AddSlot
Slots(sars)
DefaultSlot(sars)
sars <- DropSlot(sars,"norm")
sars                         # note that the default slot reverted to count
```

`AnalyzeGeneSets` *Gene set analysis*

Description

Perform gene-set enrichment and overrepresentation analysis (GSEA/ORA) for a specified set of genes

Usage

```
AnalyzeGeneSets(
  data,
  analysis = Analyses(data)[1],
  criteria = LFC,
  genes = NULL,
  species = NULL,
  category = NULL,
  subcategory = NULL,
  verbose = TRUE,
  minSize = 10,
  maxSize = 500,
  process.genesets = NULL
)
```

Arguments

`data`	the grandR object that contains the data to analyze
`analysis`	the analysis to use, can be more than one and can be regexes (see details)
`criteria`	an expression to define criteria for GSEA/ORA (see details)
`genes`	specify genes directly (use analysis and criteria if NULL; see details)
`species`	the species the genes belong to (eg "Homo sapiens"); can be NULL, then the species is inferred from gene ids (see details)
`category`	the category defining gene sets (see ListGeneSets)
`subcategory`	the category defining gene sets (see ListGeneSets)

`verbose`	Print status messages
`minSize`	The minimal size of a gene set to be considered
`maxSize`	The maximal size of a gene set to be considered
`process.genesets`	a function to process geneset names; can be NULL (see details)

Details

The analysis parameter (just like for GetAnalysisTable can be a regex (that will be matched against all available analysis names). It can also be a vector (of regexes). Be careful with this, if more than one table e.g. with column LFC ends up in here, only the first is used (if criteria=LFC).

The criteria parameter can be used to define how analyses are performed. The criteria must be an expression that either evaluates into a numeric or logical vector. In the first case, GSEA is performed, in the latter it is ORA. The columns of the given analysis table(s) can be used to build this expression.

If no species is given, a very simple automatic inference is done, which will only work when having human or mouse ENSEMBL identifiers as gene ids.

The process.genesets parameters can be function that takes the character vector representing the names of all gene sets. The original names are replaced by the return value of this function.

Value

the clusterprofile object representing the analysis results.

See Also

GSEA,enricher,msigdbr

Examples

```
# See the differential-expression vignette!
```

`ApplyContrasts` *Apply a function over contrasts*

Description

Helper function to run many pairwise comparisons using a contrast matrix

Usage

```
ApplyContrasts(
  data,
  analysis,
  name.prefix,
  contrasts,
  mode.slot = NULL,
  genes = NULL,
  verbose = FALSE,
  FUN,
  ...
)
```

Arguments

`data`	the grandR object
`analysis`	a plain name, only used for status messages
`name.prefix`	the prefix for the new analysis name; a dot and the column names of the contrast matrix are appended; can be NULL (then only the contrast matrix names are used)
`contrasts`	contrast matrix that defines all pairwise comparisons, generated using GetContrasts
`mode.slot`	which slot to take expression values from
`genes`	restrict analysis to these genes; NULL means all genes
`verbose`	print status messages?
`FUN`	a function taking 1. the data matrix, 2. a logical vector indicating condition A and 3. a logical vector indicating condition B
`...`	further parameters forward to FUN

Details

To implement most pairwise analyses, you only have to define FUN; see the source code of LFC for an example!

Value

a new grandR object with added analysis tables (that were returned by FUN)

See Also

LFC,PairwiseDESeq2,GetContrasts

`as.Seurat.grandR` *Create Seurat object from a grandR object*

Description

Create Seurat object from a grandR object

Usage

```
as.Seurat.grandR(
  data,
  modalities = c(RNA = "total", newRNA = "new"),
  hls = NULL,
  time = NULL,
  mode = c("assay", "cells", "genes", "list")
)
```

Arguments

`data`	a grandR object
`modalities`	vector defining modalities to include in the Seurat object (see details)
`hls`	half-lives for computing previous RNA, only required for "prev" modality (see details)
`time`	labeling time, only required for "prev" modality (see details)
`mode`	how to integrate modalities into seurat object (see details)

Details

Modalities must be a named character vector. The only allowed elements are "total" (total counts), "new" (new counts), "old" (old counts), "prev" (estimated previous time point counts). The names of the elements are further used depending on mode.

To compute the previous time point counts, a vector of half lives and the labeling time is required. The half-lives must be given in the correct order (same as in the grandR object).

The mode parameter defines how the defined modalities are represented in the Seurat object. "assay" means that for each modality, the Seurat object will contain an assay (named according to the corresponding name in modalities). "cells" means that cells will be copied for each modality and cell names are prefixed by the corresponding name in modalities (i.e., if the grandR object has 1000 cells named c1,...,c1000, and modalities=c(RNA="total",newRNA="new"), the Seurat object will have 2000 cells named RNA.c1,...,RNA.c1000,newRNA.c1,...,newRNA.c1000). "genes" means that genes fill be copied for each modality and gene names are prefixed by the corresponding name in modalities. "list" means that instead of a single Seurat object, a list of Seurat objects is returned.

Value

a Seurat object

`CalibrateEffectiveLabelingTimeKineticFit` *Uses the kinetic model to calibrate the effective labeling time.*

Description

The NTRs of each sample might be systematically too small (or large). This function identifies such systematic deviations and computes labeling durations without systematic deviations.

Usage

```
CalibrateEffectiveLabelingTimeKineticFit(
  data,
  slot = DefaultSlot(data),
  time = Design$dur.4sU,
  time.name = "calibrated_time",
  time.conf.name = "calibrated_time_conf",
  CI.size = 0.95,
  compute.confidence = FALSE,
  n.estimate = 1000,
  n.iter = 10000,
  verbose = FALSE,
  ...
)
```

Arguments

`data`	A grandR object
`slot`	The data slot to take expression values from
`time`	The column in the column annotation table representing the labeling duration
`time.name`	The name in the column annotation table to put the calibrated labeling durations
`time.conf.name`	The name in the column annotation table to put the confidence values for the labeling durations (half-size of the confidence interval)
`CI.size`	The level for confidence intervals
`compute.confidence`	should CIs be computed or not?
`n.estimate`	the times are calibrated with the top n expressed genes
`n.iter`	the maximal number of iterations for the numerical optimization
`verbose`	verbose output
`...`	forwarded to FitKinetics

Details

There are many reasons why the nominal (wall-clock) time of 4sU labeling might be distinct from the effective labeling time. Most importantly, 4sU needs some time to enter the cells and get activated to be ready for transcription. Therefore, the 4sU concentration (relative to the U concentration) rises, based on observations, over the timeframe of 1-2h. GRAND-SLAM assumes a constant 4sU incorporation rate, i.e. specifically new RNA made early during the labeling is underestimated. This, especially for short labeling (<2h), the effective labeling duration might be significantly less than the nominal labeling duration.

It is impossible to obtain a perfect absolute calibration, i.e. all durations might be off by a factor.

Value

A new grandR object containing the calibrated durations in the column data annotation

See Also

FitKinetics

`CalibrateEffectiveLabelingTimeMatchHalflives`

Calibrate the effective labeling time by matching half-lives to a .reference

Description

The NTRs of each sample might be systematically too small (or large). This function identifies such systematic deviations and computes labeling durations without systematic deviations.

Usage

```
CalibrateEffectiveLabelingTimeMatchHalflives(
  data,
  reference.halflives = NULL,
  reference.columns = NULL,
  slot = DefaultSlot(data),
  time.labeling = Design$dur.4sU,
  time.experiment = NULL,
  time.name = "calibrated_time",
  n.estimate = 1000,
  verbose = FALSE
)
```

Arguments

`data`	A grandR object
`reference.halflives`	a vector of reference Half-lives named by genes
`reference.columns`	the reference column description
`slot`	The data slot to take expression values from
`time.labeling`	the column in the column annotation table denoting the labeling duration or the labeling duration itself
`time.experiment`	the column in the column annotation table denoting the experimental time point (can be NULL, see details)
`time.name`	The name in the column annotation table to put the calibrated labeling durations
`n.estimate`	the times are calibrated with the top n expressed genes
`verbose`	verbose output

Value

A new grandR object containing the calibrated durations in the column data annotation

See Also

FitKineticsGeneSnapshot

`check.analysis` *Internal functions to check for a valid analysis or slot names.*

Description

Internal functions to check for a valid analysis or slot names.

Usage

```
check.analysis(data, analyses, regex)

check.slot(data, slot, allow.ntr = TRUE)

check.mode.slot(data, mode.slot, allow.ntr = TRUE)
```

Arguments

`data`	a grandR object
`analyses`	a regex to be matched to analysis names
`regex`	interpret as regular expression
`slot`	a slot name
`allow.ntr`	whether to allow for the value "ntr" (and throw an error in case)
`mode.slot`	a mode.slot

Details

A mode.slot is a mode followed by a dot followed by a slot name, or just a slot name. A mode is either *total*, *new* or *old*.

Value

Whether or not the given name is valid and unique for the grandR object

---

`ClassifyGenes` *Build the type column for the gene info table.*

Description

Returns a function to be used as `classify.genes` parameter for `ReadGRAND`.

Usage

```
ClassifyGenes(
  ...,
  use.default = TRUE,
  drop.levels = TRUE,
  name.unknown = "Unknown"
)
```

Arguments

`...`	additional functions to define types (see details)
`use.default`	if TRUE, use the default type inference (priority after the user defined ones); see details
`drop.levels`	if TRUE, drop unused types from the factor that is generated
`name.unknown`	the type to be used for all genes where no type was identified

Details

This function returns a function. Usually, you do not use it yourself but `ClassifyGenes` is usually as `classify.genes` parameter for `ReadGRAND` to build the *Type* column in the `GeneInfo` table. See the example to see how to use it directly.

Each ... parameter must be a function that receives the gene info table and must return a logical vector, indicating for each row in the gene info table, whether it matches to a specific type. The name of the parameter is used as the type name.

If a gene matches to multiple type, the first function returning TRUE for a row in the table is used.

By default, this function will recognize mitochondrial genes (MT prefix of the gene symbol), ERCC spike-ins, and Ensembl gene identifiers (which it will call "cellular"). These three are the last functions to be checked (in case a user defined type via ...) also matches to, e.g., an Ensembl gene).

Value

a function that takes the original GeneInfo table and adds the Type column

See Also

ReadGRAND

Examples

```
viral.genes <- c('ORF3a','E','M','ORF6','ORF7a','ORF7b','ORF8','N','ORF10','ORF1ab','S')
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate),
                  classify.genes=ClassifyGenes(`SARS-CoV-2`=
                             function(gene.info) gene.info$Symbol %in% viral.genes),
                  verbose=TRUE)
table(GeneInfo(sars)$Type)

fun<-ClassifyGenes(viral=function(gene.info) gene.info$Symbol %in% viral.genes)
table(fun(GeneInfo(sars)))
```

`Coldata` *Get the column annotation table or add additional columns to it*

Description

The columns of a grandR object are samples or cells. The column annotation table contains meta information for the columns of a grandR object. When loaded from the GRAND-SLAM output, this this constructed from the sample/cell names by `MakeColdata`

Usage

```
Coldata(data, column = NULL, value = NULL)

Coldata(data, column) <- value
```

Arguments

`data`	A grandR object
`column`	The name of the additional annotation column; can also be a data frame (then value is ignored and the data frame is added)
`value`	The additional annotation per sample or cell

Details

A new column can be added either by `data<-Coldata(data,name,values)` or by `Coldata(data,name)<-values`.

Several new columns can be added by `data<-Coldata(data,df)` where df is either a data frame or matrix.

The column named *Condition* has a special meaning in this table: It is used by several functions to stratify the columns during the analysis (e.g. to estimate separate kinetic parameters with `FitKinetics` or it is used as covariate for `LFC` or `LikelihoodRatioTest`). For that reason there are special functions to set and get this column.

Value

Either the column annotation table or a new grandR object having an updated column annotation table

See Also

GeneInfo, MakeColdata, Condition

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))

head(GeneInfo(sars))
GeneInfo(sars,"LengthCategory")<-cut(GeneInfo(sars)$Length,c(0,1500,2500,Inf),
                                      labels=c("Short","Medium","Long"))
table(GeneInfo(sars)$LengthCategory)
```

`ComputeAbsolute` *Compute absolute expression using ERCC spike ins*

Description

Compute absolute expression in a grandR object and puts the normalized data into a new slot

Usage

```
ComputeAbsolute(
  data,
  dilution = 40000,
  volume = 10,
  slot = "tpm",
  name = "absolute"
)
```

Arguments

`data`	the grandR object
`dilution`	the dilution of the spikein transcript in the lysis reaction mix
`volume`	the approximate volume of the lysis chamber (nanoliters)
`slot`	the slot containing relative expression values
`name`	the name of the new slot to put absolute expression values in

Value

a new grandR object with an additional slot

See Also

relative2abs

`ComputeColumnStatistics` *Compute statistics for all columns (i.e. samples or cells)*

Description

Compute statistics for all columns (i.e. samples or cells)

Usage

```
ComputeColumnStatistics(data, verbose = TRUE)
```

Arguments

`data`	a grandR object
`verbose`	output status messages

Value

a new grandR object containing additional columns in the Coldata table:

- p.conv.X: the T-to-C mismatch frequency in the given ("X") subread category
- percent.new: new overall percentage of new RNA
- total.reads: the total number of reads (or UMIs, if UMIs were sequences)
- total.genes: the total number of genes detected
- percentage per type: the percentage (up to 100!) of the counts of each type in the GeneInfo

---

`ComputeExpressionPercentage` *Expression percentage computation*

Description

Compute the expression percentage for a particular set of genes.

Usage

```
ComputeExpressionPercentage(
  data,
  name,
  genes = Genes(data),
  mode.slot = DefaultSlot(data),
  genes.total = Genes(data),
  mode.slot.total = mode.slot,
  multiply.by.100 = TRUE
)
```

Arguments

`data`	the grandR object
`name`	the new name by which this is added to the Coldata
`genes`	define the set of genes to compute the percentage for
`mode.slot`	which mode.slot to take the values for computing the percentage from
`genes.total`	define the set of genes defining the total value
`mode.slot.total`	which mode.slot to take the values for computing the total
`multiply.by.100`	if TRUE, compute percentage values, otherwise fractions between 0 and 1

Details

The percentages are computed for the given genes with the given mode.slot, w.r.t the mode.slot.total from the genes.total. Thus to compute the percentage of mitochondrial gene expression in total RNA (unnormalized), only set genes=Genes(data,"^MT-",regex=TRUE). To compute the percentage of new RNA among all genes, set mode.slot="new.count" and mode.slot.total="count".

Genes can be referred to by their names, symbols, row numbers in the gene table, or a logical vector referring to the gene table rows.

To refer to data slots, the mode.slot syntax can be used: Each name is either a data slot, or one of (new,old,total) followed by a dot followed by a slot. For new or old, the data slot value is multiplied by ntr or 1-ntr. This can be used e.g. to filter by *new counts*.

Value

a new grandR object having the expression percentage in its Coldata table

See Also

`Coldata`

`ComputeNonConstantParam` *Compute and evaluate functions for non constant rates*

Description

For simplicity, non constant rates here have the following form $o+f*t^e$.

Usage

```
ComputeNonConstantParam(start, end = start, exponent = 1, end.time = 2)

EvaluateNonConstantParam(t, param)
```

Arguments

`start`	the value at t=0
`end`	the value at t=end.time
`exponent`	the exponent (e above)
`end.time`	the end time
`t`	vector of times
`param`	output of `ComputeNonConstantParam()`, only a single row!

Value

data frame containing either the parameters o, f and e (ComputeNonConstantParam), or containing the value of $o+f*t^e$ for the given times (EvaluateNonConstantParam).

Functions

- `ComputeNonConstantParam()`: compute a data frame containing the parameters for non constant rates
- `EvaluateNonConstantParam()`: compute a data frame containing the rates for the given parameter set (computed from `ComputeNonConstantParam`)

`ComputeNtrPosteriorQuantile` *Compute NTR quantiles*

Description

Computes quantiles from the NTR posterior and puts them into a new slot

Usage

```
ComputeNtrPosteriorQuantile(data, quantile, name)

ComputeNtrCI(data, CI.size = 0.95, name.lower = "lower", name.upper = "upper")

ComputeNtrPosteriorLower(data, CI.size = 0.95, name = "lower")

ComputeNtrPosteriorUpper(data, CI.size = 0.95, name = "upper")
```

Arguments

`data`	the grandR object
`quantile`	which quantile to compute
`name`	the name of the new slot to put quantile values in
`CI.size`	A number between 0 and 1 representing the size of the credible interval
`name.lower`	the name of the new slot to put the lower bound of the CI in
`name.upper`	the name of the new slot to put the upper bound of the CI in

Details

The NTR posterior distribution can be approximated by a beta distribution.

ComputeNtrPosteriorQuantile computes any quantile from this Beta approximation

ComputeNtrPosteriorLower computes the (1-CI.size)/2 quantile

ComputeNtrPosteriorUpper computes the 1-(1-CI.size)/2 quantile

ComputeNtrCI computes both of these quantiles.

Value

a new grandR object containing an additional slot

`ComputePseudoNtr` *Compute pseudo NTRs from two count matrices*

Description

NTRs can be computed from given new and total counts.

Usage

```
ComputePseudoNtr(
  data,
  new.slot,
  total.slot = DefaultSlot(data),
  detection.rate = 1
)
```

Arguments

`data`	a grandR object
`new.slot`	the slot containing new RNA counts
`total.slot`	the slot containing total RNA counts
`detection.rate`	the detection rate of T-to-C mismatch reads (see details)

Details

To correct for some bias, a detection rate (as suggested by Cao et al., Nature Biotech 2020) should be provided. This detection rate defines, how much new RNA is detected on average using the T-to-C mismatch reads.

Value

a new grandR object

`ComputeSteadyStateHalfLives` *Steady state half-lives for each sample*

Description

Transforms each NTR to a half-life value (assuming steady state gene expression) and puts them into a new slot or adds an analysis

Usage

```
ComputeSteadyStateHalfLives(
  data,
  time = Design$dur.4sU,
  name = "HL",
  columns = NULL,
  max.HL = 48,
  CI.size = 0.95,
  compute.CI = FALSE,
  as.analysis = FALSE
)
```

Arguments

`data`	the grandR object
`time`	either a number indicating the labeling time, or a name of the Coldata table
`name`	the name of the new slot/analysis to put half-life values in
`columns`	which columns (i.e. samples or cells) to return; sets as.analysis to TRUE (see details)
`max.HL`	all values above this will be set to this
`CI.size`	A number between 0 and 1 representing the size of the credible interval
`compute.CI`	it TRUE, credible intervals are computed, this also sets as.analysis to TRUE
`as.analysis`	if TRUE add the results as analysis and not as data slot

Details

An NTR value p can be transformed into an RNA half-live using the equation log(2)/(-1/t*log(1-p)) This is described in our GRAND-SLAM paper (Juerges et al., Bioinformatics 2018).

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment havin the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

Value

a new grandR object with an additional slot or analysis

`ComputeSummaryStatistics` *Compute summary statistics*

Description

Summary statistics are computed for all samples (or cells).

Usage

```
ComputeSummaryStatistics(
  data,
  pairs = Findno4sUPairs(data),
  coldata = FALSE,
  do.bootstrap = FALSE,
  seed = 1337
)
```

Arguments

`data`	a grandR object
`pairs`	a no4sU pairs list as generated by Findno4sUPairs
`coldata`	if TRUE, add the coldata table
`do.bootstrap`	if TRUE, also report standard errors of the 4sU dropout estimated via bootstrapping
`seed`	the seed for the random number generator for bootstrapping

Value

a table of summary statistics including:

- Mean LFC: the mean absolute log2 fold change of each sample vs the corresponding 4sU naive sample
- 4sU dropout: the estimated 4sU dropout percentage
- p.conv: The 4sU incorporation frequency estimated by GRAND-SLAM
- Frction labelled: the global NTR

`Condition` *Get or set the conditions in the column annotation table.*

Description

The conditions column from the column annotation table is used by several functions to stratify the columns (samples or cells) during the analysis (e.g. to estimate separate kinetic parameters with `FitKinetics` or it is used as covariate for `LFC` or `LikelihoodRatioTest`). For that reason there are special functions to set and get this column.

Usage

```
Condition(data, value = NULL)

Condition(data) <- value
```

Arguments

`data`	A grandR object
`value`	Either a vector of column names from the column annotation table, or the condition names themselves

Details

If the conditions column does not exist (or has been set to NULL), all analysis functions will work without stratifying samples or cells. The condition can also be set up directly when loading data, by using *Condition* as one of the design vector entries (see below).

The condition can be set either by `data<-Condition(data,names)` or by `Condition(data)<-names`.

Value

Either the values of the condition column for Condition(data) or the grandR data object having the new condition column

See Also

Coldata

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))

Condition(sars)
Condition(sars) <- c("Cell","duration.4sU.original")
Condition(sars)

sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition",Design$dur.4sU,Design$Replicate))
Condition(sars)
```

`correctdropout` *Correct for 4sU dropout*

Description

For several potential reasons, a sample specific percentage of reads from labelled RNA might be lost. This can be corrected for by increasing the amount of labelled RNA (see details).

Usage

```
Correct4sUDropoutHLFactor(
  data,
  pairs = Findno4sUPairs(data),
  factors = Estimate4sUDropoutPercentage(data, pairs = pairs, ...),
  ...
)

Correct4sUDropoutHLSpline(data, pairs = Findno4sUPairs(data), spline.df = 15)
```

Arguments

`data`	a grandR object
`pairs`	a no4sU pairs list as generated by Findno4sUPairs
`factors`	the 4sU dropout percentages
`...`	further arguments to be passed to or from other methods.
`spline.df`	the degrees of freedom to be used for smoothing splines

Details

The factor based correction approach requires estimates of the 4sU dropout percentage d. Labelled RNA is multiplied by 1/(1-d), and total count and NTRs are adapted accordingly. alpha and beta are also adapted such that their sum is maintained, but the mean of the corresponding beta function is the new NTR. All other slots are treated to be expression estimates (and are adapted accordingly).

The spline approach uses quantile regression to fit a smoothing spline to the 4sU dropout rank plot, which is then used to correct labelled RNA.

Value

a new grandR object that is corrected for 4sU dropout

See Also

Estimate4sUDropoutPercentage,ComputeSummaryStatistics

`CreateConvolutionTable` *Create Convolution Table from a Seurat object*

Description

Create Convolution Table from a Seurat object

Usage

```
CreateConvolutionTable(data, n.neighbors = 20)
```

Arguments

`data`	a Seurat object
`n.neighbors`	the number of neighbors to be convoluted

Details

This function returns a table which can be used as input for GRAND3. Note that a data set contatining multiple time points should be split before convolution.

Value

a table with two columns "Cell" and "Pseudobulk"

`CreatePdfs` *Convencience methods for creating QC pdfs*

Description

These methods are invoked by GRAND3 to generate pdfs.

Usage

```
CreatePdfs(data, labels = NULL, estimators = NULL)

CreatePdfsParameters(data, labels = NULL, estimators = NULL)

CreatePdfsComparison(data, labels = NULL, estimators = NULL)

CreatePdfsProfiles(data, labels = NULL, estimators = NULL)
```

Arguments

`data`	a grandR object
`labels`	which label to consider (see GetDiagnosticParameters); if NULL, all available estimators are used
`estimators`	which estimator to consider (see GetDiagnosticParameters); if NULL, all available estimators are used

Functions

- `CreatePdfs()`: Create all pdfs
- `CreatePdfsParameters()`: Create pdfs visualizing the estimated parameters
- `CreatePdfsComparison()`: Create pdfs comparing the estimated parameters
- `CreatePdfsProfiles()`: Create pdfs visualizing the profile likelihoods

`CreatePseudobulkTable` *Create Pseudobulk Table from a Seurat object*

Description

Create Pseudobulk Table from a Seurat object

Usage

```
CreatePseudobulkTable(
  data,
  name.column = "Name",
  pseudobulk.column = "Condition"
)
```

Arguments

`data`	a Seurat object
`name.column`	name of the metadata column containing the sample/cell names. Default "Name".
`pseudobulk.column`	name of the metadata column containing the Pseudobulk names. Default "Condition".

Details

This function returns a table which can be used as input for GRAND3

Value

a table with two columns "Cell" and "Pseudobulk"

`data.apply` *Internal function to apply functions to all slots etc.*

Description

Internal function to apply functions to all slots etc.

Usage

```
data.apply(data, fun, fun.gene.info = NULL, fun.coldata = NULL, ...)
```

Arguments

`data`	a grandR object
`fun`	apply this function to each data slot (i.e. it receives each data matrix)
`fun.gene.info`	apply this function to the gene info table
`fun.coldata`	apply this function to the column annotation table
`...`	passed further to fun, fun.gene.info and fun.coldata

Details

The additional parameters are provided to each of the functions.

Value

A new grandR object

`DefaultSlot` *Get or set the default slot for a grandR object.*

Description

The default slot is used by default by many functions including `GetData`,`GetTable` or `FitKinetics`

Usage

```
DefaultSlot(data, value = NULL)

DefaultSlot(data) <- value
```

Arguments

`data`	A grandR object
`value`	the name of the new default slot

Details

The default slot can be set either by `data<-DefaultSlot(data,"norm")` or by `DefaultSlot(data)<-"norm"`.

Value

Either the name of the default slot for DefaultSlot(data) or the grandR data object having the new default slot

See Also

Slots

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))

DefaultSlot(sars)
sars <- Normalize(sars)     # default behavior is to update the default slot
DefaultSlot(sars)
DefaultSlot(sars)="count"
```

`Defer` *Defer calling a function*

Description

This generates a function with one mandatory parameter (and additional optional parameters) that, when called, (i) also receives the parameters given when calling `Defer`, and (ii) after calling it each element of the `add` list is appended by `+`. When no optional parameters are given, the result is cached.

Usage

```
Defer(FUN, ..., add = NULL, cache = TRUE, width.height = NULL)
```

Arguments

`FUN`	the function to be deferred
`...`	additional parameters to be used when the deferred function is called
`add`	list containing additional elements to be added + to the result of the deferred function
`cache`	use caching mechanism
`width.height`	a vector containing the desired width and height (not checked!)

Details

The following expressions are very similar: `f <- function(d) Heavy.function(d)` and `f <- Defer(Heavy.function)`. In both cases, you get a function `f` that you can call for some `d`, which in turn calls `Heavy.function`. The only difference is that in the second case, the result is cached: `Heavy.function` is called only once when first calling `f`, if `f` is called a second time, the previous result is returned. This makes sense if the parameter `d` is constant (like a grandR object) and if `Heavy.function` is deterministic.

If additional parameters are provided to `f`, caching is disabled. If any of these additional parameters has the same name as the parameters given to `Defer()`, the parameters given to `Defer()` are overwritten. Be careful if `Heavy.function` is not deterministic (see examples).

Use case scenario: You want to produce a heatmap from a grandR object to be used as `plot.static` in the shiny web interface. `PlotHeatmap` takes some time, and the resulting object is pretty large in memory. Saving the heatmap object to disk is very inefficient (the Rdata file will be huge, especially with many heatmaps). Deferring the call without caching also is bad, because whenever the user clicks onto the heatmap, it is regenerated.

Value

a function that can be called

Examples

```
Heavy.function <- function(data) rnorm(5,mean=data)
f1=Defer(Heavy.function)
f2=function(d) Heavy.function(d)
f2(4)
f2(4) # these are not equal, as rnorm is called twice
f1(4)
f1(4) # these are equal, as the result of rnorm is cached
```

`density2d` *Density estimation in 2d*

Description

Estimate point densities on a regular grid for.

Usage

```
density2d(x, y, facet = NULL, n = 100, margin = "n")
```

Arguments

`x`	x coordinates
`y`	y coordinates
`facet`	factor: estimate for each unique factor; can be NULL
`n`	size of the grid
`margin`	one of 'n','x' or 'y'; should the density be computed along both axes ('n'), or along 'x' or 'y' axis only

Value

a density value for each point

DESeq2BIC *Compute the Bayesian information criterion (BIC)*

Description

Compute the delta BIC for a list of potential models

Usage

```
DESeq2BIC(
  data,
  name = "BIC",
  mode = "total",
  normalization = mode,
  formulas = list(Condition = ~Condition, Background = ~1),
  no4sU = FALSE,
  columns = NULL,
  verbose = FALSE
)
```

Arguments

`data`	A grandR object
`name`	the user defined analysis name to store the results
`mode`	either "total", "new" or "old"
`normalization`	normalize on "total", "new", or "old" (see details)
`formulas`	list of formulas specifying the models (you can use any column name from the `Coldata(data)`)
`no4sU`	Use no4sU columns (TRUE) or not (FALSE)
`columns`	logical vector of which columns (samples or cells) to use (or NULL: use all)
`verbose`	Print status updates

Details

DESeq2 by default performs size factor normalization. When computing differential expression of new RNA, it might be sensible to normalize w.r.t. to total RNA, i.e. use the size factors computed from total RNA instead of computed from new RNA. This can be accomplished by setting mode to "new", and normalization to "total"!

Value

a new grandR object including a new analysis table. The columns of the new analysis table are named as <name in list>.dBIC

`Design` *A list of predefined names for design vectors*

Description

These predefined names mainly are implemented here to harmonize analyses. It is good practise to use these names if sensible.

Usage

```
Design
```

Format

An object of class `list` of length 11.

`DesignSemantics` *Build the design semantics list*

Description

This is used to add additional columns to the `Coldata` table by giving additional semantics to existing columns.

Usage

```
DesignSemantics(...)
```

Arguments

`...`	named parameter list of functions (see details)

Details

DesignSemantics returns a list of functions that is supposed to be used as `semantics` parameter when calling `MakeColdata`. For each design vector element matching a name of this list the corresponding function is called by MakeColdata to add additional columns.

Each function takes two parameters, the first being the original column in the `Coldata` table column, the second being its name.

Semantics.time is such a predefined function: Contents such as 3h or 30min are converted into a numerical value (in hours), and no4sU is converted into 0.

Semantics.concentration is such a predefined function: Contents such as 200uM or 1mM are converted into a numerical value (in uM), and no4sU is converted into 0.

By default, Semantics.time is used for the names duration.4sU and Experimental.time, and Semantics.concentration is used for concentration.4sU

Value

a named list; the names should correspond to column names in the Coldata table, and the values are functions to add semantics to this table

See Also

MakeColdata

Examples

```
Semantics.time(c("5h","30min","no4sU"),"Test")


myfun <- function(s,name) {
        r<-Semantics.time(s,name)
        cbind(r,data.frame(hpi=paste0(r$duration.4sU+3,"h")))
}
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=function(names)
                    MakeColdata(names,c("Cell",Design$dur.4sU,Design$Replicate),
                  semantics=DesignSemantics(duration.4sU=myfun)),
                  verbose=TRUE)
Coldata(sars)
```

`dropout` *Perform 4sU dropout tests*

Description

Testing for RNA dropout of a 4sU sample is performed by comparing half-lives or NTR ranks against the log2 fold change of the 4sU sample vs equivalent no4sU samples.

Usage

```
Plot4sUDropoutRankAll(data, pairs = Findno4sUPairs(data), ...)

Plot4sUDropoutAll(data, pairs = Findno4sUPairs(data), ...)

Plot4sUDropoutDeferAll(data, pairs = NULL, ...)

Plot4sUDropoutRankDeferAll(data, pairs = NULL, ...)

Plot4sUDropoutRank(
  data,
  w4sU,
  no4sU = Findno4sUPairs(data)[[w4sU]],
```

```
    ntr = w4sU,
    ylim = NULL,
    LFC.fun = lfc::PsiLFC,
    slot = "count",
    correction = 1,
    label.corr = TRUE,
    return.corr = FALSE,
    boxplot.bins = 10,
    title = w4sU,
    size = 1.5,
    invert.ranks = FALSE
  )

  Plot4sUDropout(
    data,
    w4sU,
    no4sU = Findno4sUPairs(data)[[w4sU]],
    ntr = w4sU,
    ylim = NULL,
    LFC.fun = lfc::PsiLFC,
    slot = "count",
    hl.quantile = 0.8,
    hl = NULL,
    correction = 1,
    label.corr = FALSE,
    return.corr = FALSE,
    title = w4sU,
    size = 1.5,
    color.by.ntr = FALSE
  )
```

Arguments

`data`	a grandR object
`pairs`	a no4sU pairs list as generated by Findno4sUPairs
`...`	further arguments to be passed to or from other methods.
`w4sU`	the name of a 4sU sample
`no4sU`	the name(s) of equivalent no4sU sample(s)
`ntr`	the name of a sample to take NTRs from (usually equal to w4sU)
`ylim`	y axis limits
`LFC.fun`	function to compute log fold change (default: PsiLFC, other viable option: NormLFC)
`slot`	the slot of the grandR object to take the data from; for PsiLFC, this really should be "count"!
`correction`	correction factor
`label.corr`	add statistics as subtitle

`return.corr`	instead of only the ggplot object, return a list with slots plot (what is normally returned) and label (the correlation statistics)
`boxplot.bins`	how many boxplots for `Plot4sUDropoutRank`
`title`	the main title for the plot
`size`	the point size
`invert.ranks`	if TRUE, left to right on the plot is largest NTR to smallest NTR
`hl.quantile`	the half-life quantile to cut the plot
`hl`	if NULL, compute half-lives from the ntr column; otherwise, must be a vector containing half-lives
`color.by.ntr`	if true, compute the density colors along the ntr axis instead of globally

Details

The deferred versions are useful to be used in conjunction with ServeGrandR plot.static. Their implementation make sure that they are lightweight, i.e. when saving the returned function to an Rdata file, the grandR object is not stored.

Value

either a ggplot object, a list of ggplot objects, or a list of deferred functions for plotting

See Also

Findno4sUPairs,Defer

`dropoutpercent` *Estimate 4sU dropout percentages*

Description

For several potential reasons, a sample specific percentage of reads from labelled RNA might be lost. This percentage can be estimated from data of this sample and an equivalent 4sU naive control (see details).

Usage

```
Estimate4sUDropoutPercentage(data, pairs = Findno4sUPairs(data), ...)

Estimate4sUDropoutPercentageForSample(
  data,
  w4sU,
  no4sU,
  ntr = w4sU,
  LFC.fun = lfc::PsiLFC,
  type = c("spearman", "quantreg", "linear", "lowess"),
  bootstrap = FALSE
)
```

Arguments

`data`	a grandR object
`pairs`	a no4sU pairs list as generated by Findno4sUPairs
`...`	further arguments to be passed to or from other methods.
`w4sU`	the name of a 4sU sample
`no4sU`	the name(s) of equivalent no4sU sample(s)
`ntr`	the name of a sample to take NTRs from (usually equal to w4sU)
`LFC.fun`	function to compute log fold change (default: PsiLFC, other viable option: NormLFC)
`type`	one of "spearman","quantreg","linear" or "lowess" (see details)
`bootstrap`	if TRUE, perform a single bootstrap sample (by drawing genes with replacement)

Details

The percentage of 4sU dropout is estimated by numerical optimization of the factor f that has to be multiplied with the NTR to mitigate the effect of 4sU dropout. The exact objective function depends on the type parameter:

- spearman: f is estimated such that the spearman correlation coefficient of the log2 fold change 4sU/no4sU vs the ntr rank is 0
- quantreg: f is estimated such that the slope of a median regression with the the ntr rank as independent variable and the log2 fold change 4sU/no4sU as dependent variable is 0
- linear: f is estimated such that the slope of a linear regression with the the ntr rank as independent variable and the log2 fold change 4sU/no4sU as dependent variable is 0
- lowess: f is estimated by minimizing the sum-of-squares of the residuals from a lowess regression with the the ntr rank as independent variable and the log2 fold change 4sU/no4sU as dependent variable is 0

Once f is computed the percentage of 4sU dropout is f/(f+1).

Value

the percentage of 4sU dropout for a single sample (Estimate4sUDropoutPercentageForSample) or all samples (Estimate4sUDropoutPercentage)

See Also

Correct4sUDropoutHLFactor,ComputeSummaryStatistics

`estimate.dispersion` *Estimate dispersion parameters for a count matrix using DESeq2*

Description

Estimate dispersion parameters for a count matrix using DESeq2

Usage

```
estimate.dispersion(ss)
```

Arguments

`ss`	the count matrix

Value

a vector of dispersion parameters (to be used as size=1/dispersion for Xnbinom functions)

`EstimateRegulation` *Estimate regulation from snapshot experiments*

Description

Compute the posterior log2 fold change distributions of RNA synthesis and degradation

Usage

```
EstimateRegulation(
  data,
  name.prefix = "Regulation",
  contrasts,
  reference.columns = NULL,
  slot = DefaultSlot(data),
  time.labeling = Design$dur.4sU,
  time.experiment = NULL,
  ROPE.max.log2FC = 0.25,
  sample.f0.in.ss = TRUE,
  N = 10000,
  N.max = N * 10,
  CI.size = 0.95,
  seed = 1337,
  dispersion = NULL,
  sample.level = 2,
  correct.labeling = FALSE,
  verbose = FALSE
)
```

Arguments

`data`	the grandR object
`name.prefix`	the prefix for the new analysis name; a dot and the column names of the contrast matrix are appended; can be NULL (then only the contrast matrix names are used)
`contrasts`	contrast matrix that defines all pairwise comparisons, generated using GetContrasts
`reference.columns`	a reference matrix usually generated by FindReferences to define reference samples for each sample; can be NULL if all conditions are at steady state (see details)
`slot`	the data slot to take f0 and totals from
`time.labeling`	the column in the Coldata table denoting the labeling duration, or the numeric labeling duration itself
`time.experiment`	the column in the Coldata table denoting the experimental time point (can be NULL, see details)
`ROPE.max.log2FC`	the region of practical equivalence is [-ROPE.max.log2FC,ROPE.max.log2FC] in log2 fold change space
`sample.f0.in.ss`	whether or not to sample f0 under steady state conditions
`N`	the sample size
`N.max`	the maximal number of samples (necessary if old RNA > f0); if more are necessary, a warning is generated
`CI.size`	A number between 0 and 1 representing the size of the credible interval
`seed`	Seed for the random number generator
`dispersion`	overdispersion parameter for each gene; if NULL this is estimated from data
`sample.level`	Define how the NTR is sampled from the hierarchical Bayesian model (must be 0,1, or 2; see details)
`correct.labeling`	Labeling times have to be unique; usually execution is aborted, if this is not the case; if this is set to true, the median labeling time is assumed
`verbose`	Print status messages

Details

The kinetic parameters s and d are computed using TransformSnapshot. For that, the sample either must be in steady state (this is the case if defined in the reference.columns matrix), or if the levels at an earlier time point are known from separate samples, so called temporal reference samples. Thus, if s and d are estimated for a set of samples x_1,...,x_k (that must be from the same time point t), we need to find (i) the corresponding temporal reference samples from time t0, and (ii) the time difference between t and t0.

The temporal reference samples are identified by the reference.columns matrix. This is a square matrix of logicals, rows and columns correspond to all samples and TRUE indicates that the row sample is a temporal reference of the columns sample. This time point is defined by `time.experiment`. If `time.experiment` is NULL, then the labeling time of the A or B samples is used (e.g. useful if labeling was started concomitantly with the perturbation, and the steady state samples are unperturbed samples).

By default, the hierarchical Bayesian model is estimated. If sample.level = 0, the NTRs are sampled from a beta distribution that approximates the mixture of betas from the replicate samples. If sample.level = 1, only the first level from the hierarchical model is sampled (corresponding to the uncertainty of estimating the biological variability). If sample.level = 2, the first and second levels are estimated (corresponding to the full hierarchical model).

if N is set to 0, then no sampling from the posterior is performed, but the transformed MAP estimates are returned

Value

a new grandR object including a new analysis table. The columns of the new analysis table are

`"s.A"`	the posterior mean synthesis rate for sample A from the comparison
`"s.B"`	the posterior mean synthesis rate for sample B from the comparison
`"HL.A"`	the posterior mean RNA half-life for sample A from the comparison
`"HL.B"`	the posterior mean RNA half-life for sample B from the comparison
`"s.log2FC"`	the posterior mean synthesis rate log2 fold change
`"s.cred.lower"`	the lower CI boundary of the synthesis rate log2 fold change
`"s.cred.upper"`	the upper CI boundary of the synthesis rate log2 fold change
`"s.ROPE"`	the signed ROPE probability (negative means downregulation) for the synthesis rate fold change
`"HL.log2FC"`	the posterior mean half-life log2 fold change
`"HL.cred.lower"`	the lower CI boundary of the half-life log2 fold change
`"HL.cred.upper"`	the upper CI boundary of the half-life log2 fold change
`"HL.ROPE"`	the signed ROPE probability (negative means downregulation) for the half-life fold change

See Also

FitKineticsGeneSnapshot,FitKineticsSnapshot

Examples

```
banp <- ReadGRAND(system.file("extdata", "BANP.tsv.gz", package = "grandR"),
          design=c("Cell","Experimental.time","Genotype",
                          Design$dur.4sU,Design$has.4sU,Design$Replicate))
contrasts <- GetContrasts(banp,contrast=c("Experimental.time.original","0h"),name.format="$A")
reference.columns <- FindReferences(banp,reference= Experimental.time==0)
```

```
banp <- EstimateRegulation(banp,"Regulation",
                            contrasts=contrasts,
                            reference.columns=reference.columns,
                            verbose=TRUE,
                            time.experiment = "Experimental.time",
                            N=0,              # don't sample in the example
                          dispersion=0.1)    # don't estimate dispersion in the example
head(GetAnalysisTable(banp))
```

`f.nonconst` *Function to compute the abundance of new or old RNA at time t for non-constant rates.*

Description

The standard mass action kinetics model of gene expression arises from the differential equation $df/dt = s(t) - d(t)f(t)$, with s(t) being the synthesis rate at time t, d(t) the degradation rate at time t and $f0 = f(0)$ (the abundance at time 0). Here, both s and d have the following form $s(t) = so + sf \cdot t^{se}$.

Usage

```
f.nonconst(t, f0, s, d)
```

Arguments

`t`	time in h (can be a vector)
`f0`	the abundance at time t=0
`s`	the synthesis rate (see details)
`d`	the degradation rate (see details)

Details

Both rates can be either (i) a single number (constant rate), (ii) a data frame with names "offset", "factor" and "exponent" (for linear functions, see ComputeNonConstantParam; only one row allowed) or (iii) a unary function time->rate. Functions

Value

the RNA abundance at time t

See Also

f.nonconst.linear

`f.nonconst.linear` *Function to compute the abundance of new or old RNA at time t for non-constant rates.*

Description

The standard mass action kinetics model of gene expression arises from the differential equation $df/dt = s(t) - d(t)f(t)$, with s(t) being the synthesis rate at time t, d(t) the degradation rate at time t and $f0 = f(0)$ (the abundance at time 0). Here, both s and d have the following form $s(t) = so + sf \cdot t^{se}$.

Usage

```
f.nonconst.linear(t, f0, so, sf, se, do, df, de)
```

Arguments

`t`	time in h (can be a vector)
`f0`	the abundance at time t=0
`so`	synthesis date offset
`sf`	synthesis date factor
`se`	synthesis date exponent
`do`	degradation rate offset
`df`	degradation rate factor
`de`	degradation rate exponent

Value

the RNA abundance at time t

See Also

f.nonconst

`f.old.equi` *Functions to compute the abundance of new or old RNA at time t.*

Description

The standard mass action kinetics model of gene expression arises from the differential equation $df/dt = s - df(t)$, with s being the constant synthesis rate, d the constant degradation rate and $f0 = f(0)$ (the abundance at time 0).

Usage

```
f.old.equi(t, s, d)

f.old.nonequi(t, f0, s, d)

f.new(t, s, d)
```

Arguments

`t`	time in h
`s`	synthesis date in U/h (arbitrary unit U)
`d`	degradation rate in 1/h
`f0`	the abundance at time t=0

Value

the RNA abundance at time t

Functions

- `f.old.equi()`: abundance of old RNA assuming steady state (i.e. f0=s/d)
- `f.old.nonequi()`: abundance of old RNA without assuming steady state
- `f.new()`: abundance of new RNA (steady state does not matter)

Examples

```
d=log(2)/2
s=10

f.new(2,s,d)  # Half-life 2, so after 2h the abundance should be half the steady state
f.old.equi(2,s,d)
s/d

t<-seq(0,10,length.out=100)
plot(t,f.new(t,s,d),type='l',col='blue',ylim=c(0,s/d))
lines(t,f.old.equi(t,s,d),col='red')
abline(h=s/d,lty=2)
```

```
abline(v=2,lty=2)
# so old and new RNA are equal at t=HL (if it is at steady state at t=0)

plot(t,f.new(t,s,d),type='l',col='blue')
lines(t,f.old.nonequi(t,f0=15,s,d),col='red')
abline(h=s/d,lty=2)
abline(v=2,lty=2)
# so old and new RNA are not equal at t=HL (if it is not at steady state at t=0)
```

`FilterGenes` *Filter genes*

Description

Return a grandR object with fewer genes than the given grandR object (usually to filter out weakly expressed genes).

Usage

```
FilterGenes(
  data,
  mode.slot = "count",
  minval = 100,
  mincol = ncol(data)/2,
  min.cond = NULL,
  use = NULL,
  keep = NULL,
  return.genes = FALSE
)
```

Arguments

`data`	the grandR object
`mode.slot`	the mode.slot that is used for filtering (see details)
`minval`	the minimal value for retaining a gene
`mincol`	the minimal number of columns (i.e. samples or cells) a gene has to have a value >= minval
`min.cond`	if not NULL, do not compare values per column, but per condition (see details)
`use`	if not NULL, defines the genes directly that are supposed to be retained (see details)
`keep`	if not NULL, defines genes directly, that should be kept even though they do not adhere to the filtering criteria (see details)
`return.genes`	if TRUE, return the gene names instead of a new grandR object

Details

By default genes are retained, if they have 100 read counts in at least half of the columns (i.e. samples or cells).

The `use` parameter can be used to define genes to be retained directly. The `keep` parameter, in contrast, defines *additional* genes to be retained. For both, genes can be referred to by their names, symbols, row numbers in the gene table, or a logical vector referring to the gene table rows.

To refer to data slots, the mode.slot syntax can be used: Each name is either a data slot, or one of (new,old,total) followed by a dot followed by a slot. For new or old, the data slot value is multiplied by ntr or 1-ntr. This can be used e.g. to filter by *new counts*.

if the `min.cond` parameter is given, first all columns belonging to the same `Condition` are summed up, and then the usual filtering is performed by conditions instead of by columns.

Value

either a new grandR object (if return.genes=FALSE), or a vector containing the gene names that would be retained

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition",Design$dur.4sU,Design$Replicate))

nrow(sars)
# This is already filtered and has 1045 genes
nrow(FilterGenes(sars,minval=1000))
# There are 966 genes with at least 1000 read counts in half of the samples
nrow(FilterGenes(sars,minval=10000,min.cond=1))
# There are 944 genes with at least 10000 read counts in the Mock or SARS condition
nrow(FilterGenes(sars,use=GeneInfo(sars,"Type")!="Cellular"))
# These are the 11 viral genes.
```

`Findno4sUPairs` *Find equivalent no4sU samples for 4sU samples*

Description

Identify all no4sU samples in the same condition, and return everything as a list to be used in Plot4sUDropout, Plot4sUDropoutRank, Plot4sUDropoutAll, Plot4sUDropoutRankAll

Usage

```
Findno4sUPairs(data, paired.replicates = FALSE, discard.no4sU = TRUE)
```

Arguments

`data`	a grandR object
`paired.replicates`	pair replicates, i.e. only no4sU.A is found for 4sU.A
`discard.no4sU`	do not report references for no4sU samples

Value

a named list containing, for each 4sU sample, a vector of equivalent no4sU samples

See Also

Plot4sUDropout, Plot4sUDropoutRank, Plot4sUDropoutAll, Plot4sUDropoutRankAll

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition",Design$dur.4sU,Design$Replicate))
Findno4sUPairs(sars)
```

`FindReferences` *Obtain reference columns (samples or cells) for all columns (samples or cells) in the data set*

Description

In some situations (see examples) it is required to find a reference sample of some kind for each sample in a data set. This is a convenience method to find such reference samples, and provide them as a lookup table.

Usage

```
FindReferences(
  data,
  reference = NULL,
  reference.function = NULL,
  group = NULL,
  as.list = FALSE,
  columns = NULL
)
```

Arguments

`data`	A grandR object
`reference`	Expression evaluating to a logical vector to indicate which columns are reference columns; evaluated in an environment having the columns of Coldata(data)
`reference.function`	Function evaluating to a logical vector to indicate which columns are reference columns; called with the data frame row corresponding to the sample, and evaluated in an environment having the columns of Coldata(data)
`group`	a vector of colnames in Coldata(data)
`as.list`	return it as a list (names correspond to each sample, elements are the reference samples)
`columns`	find references only for a subset of the columns (samples or cells; can be NULL)

Details

Without any group, the list simply contains all references for each sample/cell. With groups defined, each list entry consists of all references from the same group.

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment havin the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

Value

A logical matrix that contains for each sample or cell (in columns) a TRUE for the corresponding corresponding reference samples or cells in rows

See Also

Coldata,Findno4sUPairs, Condition

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition",Design$dur.4sU,Design$Replicate))
FindReferences(sars,reference=no4sU)
# obtain the corresponding no4sU sample for each sample; use the Condition column
FindReferences(sars,Condition=="Mock",group="duration.4sU.original")
# obtain for each sample the corresponding sample in the Mock condition
FindReferences(sars,Condition=="Mock",group=c("duration.4sU.original","Replicate"))
# obtain for each sample the corresponding Mock sample, paying attention to replicates
```

`FitKinetics` *Fit kinetic models to all genes.*

Description

Fit the standard mass action kinetics model of gene expression by different methods. Some methods require steady state assumptions, for others data must be properly normalized. The parameters are fit per Condition.

Usage

```
FitKinetics(
  data,
  name.prefix = "kinetics",
  type = c("nlls", "ntr", "lm", "chase"),
  slot = DefaultSlot(data),
  time = Design$dur.4sU,
  CI.size = 0.95,
  return.fields = c("Synthesis", "Half-life"),
  return.extra = NULL,
  ...
)
```

Arguments

`data`	A grandR object
`name.prefix`	the prefix of the analysis name to be stored in the grandR object
`type`	Which method to use (either one of "full","ntr","lm", "chase")
`slot`	The data slot to take expression values from
`time`	The column in the column annotation table representing the labeling duration
`CI.size`	A number between 0 and 1 representing the size of the confidence interval
`return.fields`	which statistics to return (see details)
`return.extra`	additional statistics to return (see details)
`...`	forwarded to `FitKineticsGeneNtr`, `FitKineticsGeneLeastSquares` or `FitKineticsGeneLogSpaceLi`

Details

The start of labeling for all samples should be the same experimental time point. The fit gets more precise with multiple samples from multiple labeling durations.

The standard mass action kinetics model of gene expression arises from the following differential equation:

$$df/dt = s - df(t)$$

This model assumes constant synthesis and degradation rates. Based on this, there are different ways for fitting the parameters:

- FitKineticsGeneLeastSquares: non-linear least squares fit on the full model; depends on proper normalization; can work without steady state; assumption of homoscedastic gaussian errors is theoretically not justified
- FitKineticsGeneLogSpaceLinear: linear model fit on the old RNA; depends on proper normalization; assumes steady state for estimating the synthesis rate; assumption of homoscedastic gaussian errors in log space is problematic and theoretically not justified
- FitKineticsGeneNtr: maximum a posteriori fit on the NTR posterior transformed to the degradation rate; as it is based on the NTR only, it is independent on proper normalization; assumes steady state; theoretically well justified

Pulse-chase designs are fit using FitKineticsGeneLeastSquares while only considering the drop of labeled RNA. Note that in this case the notion "new" / "old" RNA is misleading, since labeled RNA corresponds to pre-existing RNA!

This function is flexible in what to put in the analysis table. You can specify the statistics using return.fields and return.extra (see `kinetics2vector`)

Value

A new grandR object with the fitted parameters as an analysis table

See Also

FitKineticsGeneNtr, FitKineticsGeneLeastSquares, FitKineticsGeneLogSpaceLinear

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))
sars <- FilterGenes(sars,use=1:10)
sars<-FitKinetics(sars,name="kinetics.ntr",type='ntr')
sars<-Normalize(sars)
sars<-FitKinetics(sars,name="kinetics.nlls",type='nlls')
sars<-FitKinetics(sars,name="kinetics.lm",type='lm')
head(GetAnalysisTable(sars,columns="Half-life"))
```

`FitKineticsGeneLeastSquares`

Fit a kinetic model according to non-linear least squares.

Description

Fit the standard mass action kinetics model of gene expression using least squares (i.e. assuming gaussian homoscedastic errors) for the given gene. The fit takes both old and new RNA into account and requires proper normalization, but can be performed without assuming steady state. The parameters are fit per Condition.

Usage

```
FitKineticsGeneLeastSquares(
  data,
  gene,
  slot = DefaultSlot(data),
  time = Design$dur.4sU,
  chase = FALSE,
  CI.size = 0.95,
  steady.state = NULL,
  use.old = TRUE,
  use.new = TRUE,
  maxiter = 250,
  compute.residuals = TRUE
)
```

Arguments

`data`	A grandR object
`gene`	The gene for which to fit the model
`slot`	The data slot to take expression values from
`time`	The column in the column annotation table representing the labeling duration
`chase`	is this a pulse-chase experiment? (see details)
`CI.size`	A number between 0 and 1 representing the size of the confidence interval
`steady.state`	either a named list of logical values representing conditions in steady state or not, or a single logical value for all conditions
`use.old`	a logical vector to exclude old RNA from specific time points
`use.new`	a logical vector to exclude new RNA from specific time points
`maxiter`	the maximal number of iterations for the Levenberg-Marquardt algorithm used to minimize the least squares
`compute.residuals`	set this to TRUE to compute the residual matrix

Details

The start of labeling for all samples should be the same experimental time point. The fit gets more precise with multiple samples from multiple labeling durations. In particular (but not only) without assuming steady state, also a sample without 4sU (representing time 0) is useful.

The standard mass action kinetics model of gene expression arises from the following differential equation:

$$df/dt = s - df(t)$$

This model assumes constant synthesis and degradation rates (but not necessarily that the system is in steady state at time 0). From the solution of this differential equation, it is straight forward to derive the expected abundance of old and new RNA at time t for given parameters s (synthesis rate), d (degradation rate) and f0=f(0) (the abundance at time 0). These equations are implemented in `f.old.equi` (old RNA assuming steady state gene expression, i.e. f0=s/d), `f.old.nonequi` (old RNA without assuming steady state gene expression) and `f.new` (new RNA; whether or not it is steady state does not matter).

This function finds s and d such that the squared error between the observed values of old and new RNA and their corresponding functions is minimized. For that to work, data has to be properly normalized.

For pulse-chase designs, only the drop of the labeled RNA is considered. Note that in this case the notion "new" / "old" RNA is misleading, since labeled RNA corresponds to pre-existing RNA!

Value

A named list containing the model fit:

- data: a data frame containing the observed value used for fitting
- residuals: the computed residuals if compute.residuals=TRUE, otherwise NA
- Synthesis: the synthesis rate (in U/h, where U is the unit of the slot)
- Degradation: the degradation rate (in 1/h)
- Half-life: the RNA half-life (in h, always equal to log(2)/degradation-rate
- conf.lower: a vector containing the lower confidence bounds for Synthesis, Degradation and Half-life
- conf.upper: a vector containing the lower confidence bounds for Synthesis, Degradation and Half-life
- f0: The abundance at time 0 (in U)
- logLik: the log likelihood of the model
- rmse: the total root mean square error
- rmse.new: the total root mean square error for all new RNA values used for fitting
- rmse.old: the total root mean square error for all old RNA values used for fitting
- total: the total sum of all new and old RNA values used for fitting
- type: non-equi or equi

If `Condition(data)` is not NULL, the return value is a named list (named according to the levels of `Condition(data)`), each element containing such a structure.

See Also

FitKinetics, FitKineticsGeneLogSpaceLinear, FitKineticsGeneNtr

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition",Design$dur.4sU,Design$Replicate))
sars <- Normalize(sars)
FitKineticsGeneLeastSquares(sars,"SRSF6",steady.state=list(Mock=TRUE,SARS=FALSE))
```

`FitKineticsGeneLogSpaceLinear` *Fit a kinetic model using a linear model.*

Description

Fit the standard mass action kinetics model of gene expression using a linear model after log-transforming the observed values (i.e. assuming gaussian homoscedastic errors of the logarithmized values) for the given gene. The fit takes only old RNA into account and requires proper normalization, but can be performed without assuming steady state for the degradation rate. The parameters are fit per Condition.

Usage

```
FitKineticsGeneLogSpaceLinear(
  data,
  gene,
  slot = DefaultSlot(data),
  time = Design$dur.4sU,
  CI.size = 0.95
)
```

Arguments

`data`	A grandR object
`gene`	The gene for which to fit the model
`slot`	The data slot to take expression values from
`time`	The column in the column annotation table representing the labeling duration
`CI.size`	A number between 0 and 1 representing the size of the confidence interval

Details

The start of labeling for all samples should be the same experimental time point. The fit gets more precise with multiple samples from multiple labeling durations. Also a sample without 4sU (representing time 0) is useful.

The standard mass action kinetics model of gene expression arises from the following differential equation:

$$df/dt = s - df(t)$$

This model assumes constant synthesis and degradation rates (but not necessarily that the system is in steady state at time 0). From the solution of this differential equation, it is straight forward to derive the expected abundance of old and new RNA at time t for given parameters s (synthesis rate), d (degradation rate) and f0=f(0) (the abundance at time 0). These equations are implemented in `f.old.equi` (old RNA assuming steady state gene expression, i.e. f0=s/d), `f.old.nonequi` (old

RNA without assuming steady state gene expression) and `f.new` (new RNA; whether or not it is steady state does not matter).

This function primarily finds d such that the squared error between the observed values of old and new RNA and their corresponding functions is minimized in log space. For that to work, data has to be properly normalized, but this is independent on any steady state assumptions. The synthesis rate is computed (under the assumption of steady state) as $s = f0 \cdot d$

Value

A named list containing the model fit:

- data: a data frame containing the observed value used for fitting
- Synthesis: the synthesis rate (in U/h, where U is the unit of the slot)
- Degradation: the degradation rate (in 1/h)
- Half-life: the RNA half-life (in h, always equal to log(2)/degradation-rate
- conf.lower: a vector containing the lower confidence bounds for Synthesis, Degradation and Half-life
- conf.upper: a vector containing the lower confidence bounds for Synthesis, Degradation and Half-life
- f0: The abundance at time 0 (in U)
- logLik: the log likelihood of the model
- rmse: the total root mean square error
- adj.r.squared: adjusted R^2 of the linear model fit
- total: the total sum of all new and old RNA values used for fitting
- type: always "lm"

If `Condition(data)` is not NULL, the return value is a named list (named according to the levels of `Condition(data)`), each element containing such a structure.

See Also

FitKinetics, FitKineticsGeneLeastSquares, FitKineticsGeneNtr

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition",Design$dur.4sU,Design$Replicate))
sars <- Normalize(sars)
FitKineticsGeneLogSpaceLinear(sars,"SRSF6")   # fit per condition
```

`FitKineticsGeneNtr` *Fit a kinetic model using the degradation rate transformed NTR posterior distribution.*

Description

Fit the standard mass action kinetics model of gene expression by maximum a posteriori on a model based on the NTR posterior. The fit takes only the NTRs into account and is completely independent on normalization, but it cannot be performed without assuming steady state. The parameters are fit per Condition.

Usage

```
FitKineticsGeneNtr(
  data,
  gene,
  slot = DefaultSlot(data),
  time = Design$dur.4sU,
  CI.size = 0.95,
  transformed.NTR.MAP = TRUE,
  exact.ci = FALSE,
  total.fun = median
)
```

Arguments

`data`	A grandR object
`gene`	The gene for which to fit the model
`slot`	The data slot to take expression values from
`time`	The column in the column annotation table representing the labeling duration
`CI.size`	A number between 0 and 1 representing the size of the credible interval
`transformed.NTR.MAP`	Use the transformed NTR MAP estimator instead of the MAP of the transformed posterior
`exact.ci`	compute exact credible intervals (see details)
`total.fun`	use this function to summarize the expression values (only relevant for computing the synthesis rate s)

Details

The start of labeling for all samples should be the same experimental time point. The fit gets more precise with multiple samples from multiple labeling durations.

The standard mass action kinetics model of gene expression arises from the following differential equation:

$$df/dt = s - df(t)$$

This model assumes constant synthesis and degradation rates. Further assuming steady state allows to derive the function transforming from the NTR to the degradation rate d as $d(ntr) = -1/tlog(1 - ntr)$. Furthermore, if the ntr is (approximately) beta distributed, it is possible to derive the distribution of the transformed random variable for the degradation rate (see Juerges et al., Bioinformatics 2018).

This function primarily finds d by maximizing the degradation rate posterior distribution. For that, data does not have to be normalized, but this only works under steady-state conditions. The synthesis rate is then computed (under the assumption of steady state) as $s = f0 \cdot d$

The maximum-a-posteriori estimator is biased. Bias can be removed by a correction factor (which is done by default).

By default the chi-squared approximation of the log-posterior function is used to compute credible intervals. If exact.ci is used, the posterior is integrated numerically.

Value

A named list containing the model fit:

- data: a data frame containing the observed value used for fitting
- Synthesis: the synthesis rate (in U/h, where U is the unit of the slot)
- Degradation: the degradation rate (in 1/h)
- Half-life: the RNA half-life (in h, always equal to log(2)/degradation-rate
- conf.lower: a vector containing the lower confidence bounds for Synthesis, Degradation and Half-life
- conf.upper: a vector containing the lower confidence bounds for Synthesis, Degradation and Half-life
- f0: The abundance at time 0 (in U)
- logLik: the log likelihood of the model
- rmse: the total root mean square error
- total: the total sum of all new and old RNA values used for fitting
- type: always "ntr"

If `Condition(data)` is not NULL, the return value is a named list (named according to the levels of `Condition(data)`), each element containing such a structure.

See Also

FitKinetics, FitKineticsGeneLeastSquares, FitKineticsGeneLogSpaceLinear

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition",Design$dur.4sU,Design$Replicate))
sars <- Normalize(sars)
sars <- subset(sars,columns=Condition=="Mock")
FitKineticsGeneNtr(sars,"SRSF6")
```

FitKineticsGeneSnapshot

Compute the posterior distributions of RNA synthesis and degradation for a particular gene

Description

Compute the posterior distributions of RNA synthesis and degradation for a particular gene

Usage

```
FitKineticsGeneSnapshot(
  data,
  gene,
  columns = NULL,
  reference.columns = NULL,
  dispersion = NULL,
  slot = DefaultSlot(data),
  time.labeling = Design$dur.4sU,
  time.experiment = NULL,
  sample.f0.in.ss = TRUE,
  sample.level = 2,
  beta.prior = NULL,
  return.samples = FALSE,
  return.points = FALSE,
  N = 10000,
  N.max = N * 10,
  CI.size = 0.95,
  correct.labeling = FALSE
)
```

Arguments

`data`	the grandR object
`gene`	a gene name or symbol or index
`columns`	samples or cell representing the same experimental condition (must refer to a unique labeling duration)

`reference.columns`	a reference matrix usually generated by FindReferences to define reference samples for each sample (see details)
`dispersion`	dispersion parameter for the given columns (if NULL, this is estimated from the data, takes a lot of time!)
`slot`	the data slot to take f0 and totals from
`time.labeling`	the column in the column annotation table denoting the labeling duration or the labeling duration itself
`time.experiment`	the column in the column annotation table denoting the experimental time point (can be NULL, see details)
`sample.f0.in.ss`	whether or not to sample f0 under steady state conditions
`sample.level`	Define how the NTR is sampled from the hierarchical Bayesian model (must be 0,1, or 2; see details)
`beta.prior`	The beta prior for the negative binomial used to sample counts, if NULL, a beta distribution is fit to all expression values and given dispersions
`return.samples`	return the posterior samples of the parameters?
`return.points`	return the point estimates per replicate as well?
`N`	the posterior sample size
`N.max`	the maximal number of posterior samples (necessary if old RNA > f0); if more are necessary, a warning is generated
`CI.size`	A number between 0 and 1 representing the size of the credible interval
`correct.labeling`	whether to correct labeling times

Details

The kinetic parameters s and d are computed using TransformSnapshot. For that, the sample either must be in steady state (this is the case if defined in the reference.columns matrix), or if the levels of reference samples from a specific prior time point are known. This time point is defined by `time.experiment` (i.e. the difference between the reference samples and samples themselves). If `time.experiment` is NULL, then the labeling time of the samples is used (e.g. useful if labeling was started concomitantly with the perturbation, and the reference samples are unperturbed samples).

By default, the hierarchical Bayesian model is estimated. If sample.level = 0, the NTRs are sampled from a beta distribution that approximates the mixture of betas from the replicate samples. If sample.level = 1, only the first level from the hierarchical model is sampled (corresponding to the uncertainty of estimating the biological variability). If sample.level = 2, the first and second levels are estimated (corresponding to the full hierarchical model).

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

Value

a list containing the posterior mean of s and s, its credible intervals and, if return.samples=TRUE a data frame containing all posterior samples

---

`FitKineticsPulseR` *Fit kinetics using pulseR*

Description

Fit kinetics using pulseR

Usage

```
FitKineticsPulseR(data, name = "pulseR", time = Design$dur.4sU)
```

Arguments

`data`	A grandR object
`name`	the user defined analysis name to store the results
`time`	The column in the column annotation table representing the labeling duration

Details

This is adapted code from https://github.com/dieterich-lab/ComparisonOfMetabolicLabeling

Value

a new grandR object containing the pulseR analyses in a new analysis table

---

`FitKineticsSnapshot` *Fits RNA kinetics from snapshot experiments*

Description

Compute the posterior distributions of RNA synthesis and degradation from snapshot experiments for each condition

Usage

```
FitKineticsSnapshot(
  data,
  name.prefix = "Kinetics",
  reference.columns = NULL,
  slot = DefaultSlot(data),
  conditions = NULL,
  time.labeling = Design$dur.4sU,
  time.experiment = NULL,
  sample.f0.in.ss = TRUE,
  N = 10000,
  N.max = N * 10,
  CI.size = 0.95,
  seed = 1337,
  dispersion = NULL,
  sample.level = 2,
  correct.labeling = FALSE,
  verbose = FALSE
)
```

Arguments

`data`	the grandR object
`name.prefix`	the prefix for the new analysis name; a dot and the column names of the contrast matrix are appended; can be NULL (then only the contrast matrix names are used)
`reference.columns`	a reference matrix usually generated by FindReferences to define reference samples for each sample (see details), can be NULL if all conditions are at steady state
`slot`	the data slot to take f0 and totals from
`conditions`	character vector of all condition names to estimate kinetics for; can be NULL (i.e. all conditions)
`time.labeling`	the column in the column annotation table denoting the labeling duration or the labeling duration itself
`time.experiment`	the column in the column annotation table denoting the experimental time point (can be NULL, see details)
`sample.f0.in.ss`	whether or not to sample f0 under steady state conditions
`N`	the sample size
`N.max`	the maximal number of samples (necessary if old RNA > f0); if more are necessary, a warning is generated
`CI.size`	A number between 0 and 1 representing the size of the credible interval
`seed`	Seed for the random number generator

`dispersion`	overdispersion parameter for each gene; if NULL this is estimated from data
`sample.level`	Define how the NTR is sampled from the hierarchical Bayesian model (must be 0,1, or 2; see details)
`correct.labeling`	Labeling times have to be unique; usually execution is aborted, if this is not the case; if this is set to true, the median labeling time is assumed
`verbose`	Vebose output

Details

The kinetic parameters s and d are computed using TransformSnapshot. For that, the sample either must be in steady state (this is the case if defined in the reference.columns matrix), or if the levels at an earlier time point are known from separate samples, so called temporal reference samples. Thus, if s and d are estimated for a set of samples x_1,...,x_k (that must be from the same time point t), we need to find (i) the corresponding temporal reference samples from time t0, and (ii) the time difference between t and t0.

The temporal reference samples are identified by the reference.columns matrix. This is a square matrix of logicals, rows and columns correspond to all samples and TRUE indicates that the row sample is a temporal reference of the columns sample. This time point is defined by `time.experiment`. If `time.experiment` is NULL, then the labeling time of the A or B samples is used (e.g. useful if labeling was started concomitantly with the perturbation, and the steady state samples are unperturbed samples).

By default, the hierarchical Bayesian model is estimated. If sample.level = 0, the NTRs are sampled from a beta distribution that approximates the mixture of betas from the replicate samples. If sample.level = 1, only the first level from the hierarchical model is sampled (corresponding to the uncertainty of estimating the biological variability). If sample.level = 2, the first and second levels are estimated (corresponding to the full hierarchical model).

if N is set to 0, then no sampling from the posterior is performed, but the transformed MAP estimates are returned

Value

a new grandR object including new analysis tables (one per condition). The columns of the new analysis table are

`"s"`	the posterior mean synthesis rate
`"HL"`	the posterior mean RNA half-life
`"s.cred.lower"`	the lower CI boundary of the synthesis rate
`"s.cred.upper"`	the upper CI boundary of the synthesis rate
`"HL.cred.lower"`	the lower CI boundary of the half-life
`"HL.cred.upper"`	the upper CI boundary of the half-life

`FormatCorrelation` *Formatting function for correlations*

Description

Returns a function that takes x and y and returns a formatted output to describe the correlation of x and y

Usage

```
FormatCorrelation(
  method = "pearson",
  n.format = NULL,
  coeff.format = "%.2f",
  p.format = "%.2g",
  slope.format = NULL,
  rmsd.format = NULL
)
```

Arguments

`method`	how to compute correlation coefficients (can be pearson, spearman or kendall)
`n.format`	format string for the number of data points (see sprintf); can be NULL (don't output the number of data points)
`coeff.format`	format string for the correlation coefficient (see sprintf); can be NULL (don't output the correlation coefficient)
`p.format`	format string for the P value (see sprintf); can be NULL (don't output the P value)
`slope.format`	format string for the slope (see sprintf); can be NULL (don't output the slope)
`rmsd.format`	format string for the root mean square deviation (see sprintf); can be NULL (don't output the rmsd)

Details

Use this for the `correlation` parameter of PlotScatter

The slope is computed via a principal component analysis and *not* by linear regression

Value

a function

Examples

```
set.seed(42)
data <- data.frame(u=runif(500))  # generate some correlated data
data$x <- rnorm(500,mean=data$u)
data$y <- rnorm(500,mean=data$u)

fun <- FormatCorrelation()
fun(data$x,data$y)

fun <- FormatCorrelation(method="spearman",p.format="%.4g")
fun(data$x,data$y)
```

`GeneInfo` *Get the gene annotation table or add additional columns to it*

Description

The gene annotation table contains meta information for the rows of a grandR object. When loaded from the GRAND-SLAM output, this this contains gene ids, gene symbols, the transcript length and the type.

Usage

```
GeneInfo(data, column = NULL, value = NULL)

GeneInfo(data, column) <- value
```

Arguments

`data`	A grandR object
`column`	The name of the additional annotation column
`value`	The additional annotation per gene

Details

New columns can be added either by `data<-GeneInfo(data,name,values)` or by `GeneInfo(data,name)<-values`.

Value

Either the gene annotation table or a new grandR object having an updated gene annotation table

See Also

Genes, Coldata, ReadGRAND

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))

head(GeneInfo(sars))
GeneInfo(sars,"LengthCategory")<-cut(GeneInfo(sars)$Length,c(0,1500,2500,Inf),
                                      labels=c("Short","Medium","Long"))
table(GeneInfo(sars)$LengthCategory)
```

`Genes` *Gene and sample (or cell) names*

Description

Get the genes and sample (or cell) names for a grandR object, or add an additional gene annotation column

Usage

```
Genes(data, genes = NULL, use.symbols = TRUE, regex = FALSE)

Columns(data, columns = NULL, reorder = FALSE)
```

Arguments

`data`	A grandR object
`genes`	which genes to use
`use.symbols`	obtain the gene symbols instead of gene names
`regex`	treat genes as a regex, and return all that match
`columns`	which columns (i.e. samples or cells) to return (see details)
`reorder`	if TRUE, do not enforce the current order of columns

Details

The genes are either the (often unreadable) gene ids (e.g. Ensembl ids), or the symbols.

`Genes(data,use.symbols=FALSE)` it the same as `rownames(data)`, and `Columns(data)` is the same as `colnames(data)`

If both column and value are specified for `GeneInfo`, a new column is added to the gene annotation table

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

Value

Either the gene or column names of the grandR data object, or the columns of an analysis table in the grandR object

See Also

Coldata, GeneInfo, Analyses

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))

all(Genes(sars,use.symbols = FALSE)==rownames(sars))
all(Columns(sars)==colnames(sars))
```

`get.mode.slot` *Internal functions to parse mode.slot strings*

Description

Internal functions to parse mode.slot strings

Usage

```
get.mode.slot(data, mode.slot, allow.ntr = TRUE)
```

Arguments

`data`	a grandR object
`mode.slot`	a mode.slot
`allow.ntr`	whether to allow for the value "ntr" (and throw an error in case)

Details

A mode.slot is a mode followed by a dot followed by a slot name, or just a slot name. A mode is either *total*, *new* or *old*

Value

a named list with elements mode and slot (or only slot in case of *ntr*,*alpha* or *beta*)

`GetAnalysisTable` *Obtain a table of analysis results values*

Description

This is the main function to access analysis results. For slot data, use `GetTable` (as a large matrix) or `GetData` (as tidy table).

Usage

```
GetAnalysisTable(
  data,
  analyses = NULL,
  regex = TRUE,
  columns = NULL,
  genes = Genes(data),
  by.rows = FALSE,
  gene.info = TRUE,
  name.by = "Symbol",
  prefix.by.analysis = TRUE
)
```

Arguments

`data`	A grandR object
`analyses`	One or several regex to be matched against analysis names (Analyses); all analysis tables if NULL
`regex`	Use regex for analyses (TRUE) or don't (FALSE, i.e. must specify the exact name)
`columns`	Regular expressions to select columns from the analysis table (all have to match!); all columns if NULL
`genes`	Restrict the output table to the given genes
`by.rows`	if TRUE, add rows if there are multiple analyses; otherwise, additional columns are appended; TRUE also sets prefix.by.analysis to FALSE!
`gene.info`	Should the table contain the GeneInfo values as well (at the beginning)?
`name.by`	A column name of Coldata(data). This is used as the rownames of the output table
`prefix.by.analysis`	Should the column names in the output prefixed by the analysis name?

Details

The names for the output table are <Analysis name>.<columns name>

Value

A data frame containing the analysis results

See Also

GetTable,GetData,Genes

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition",Design$dur.4sU,Design$Replicate))
sars<-LFC(sars,contrasts=GetContrasts(sars,group = "duration.4sU"))
head(GetAnalysisTable(sars,columns="LFC"))
```

`GetContrasts` *Create a contrast matrix*

Description

Each column of a contrast matrix represents a pairwise comparison of all samples or cells of a grandR object (or a column annotation table). Elements being 1 are contrasted vs. elements being -1 (and all 0 are irrelevant for this comparison).

Usage

```
GetContrasts(x, ...)

## S3 method for class 'grandR'
GetContrasts(
  x,
  contrast = "Condition",
  no4sU = FALSE,
  columns = NULL,
  group = NULL,
  name.format = NULL,
  ...
)

## Default S3 method:
GetContrasts(
  x,
  contrast,
  columns = NULL,
  group = NULL,
  name.format = NULL,
  ...
)
```

Arguments

`x`	A grandR object or a column annotation table
`...`	further arguments to be passed to or from other methods.
`contrast`	A vector describing what should be contrasted
`no4sU`	Use no4sU columns (TRUE) or not (FALSE)
`columns`	logical vector of which columns (samples or cells) to use (or NULL: use all); for grandR objects, see details
`group`	Split the samples or cells according to this column of the column annotation table (and adapt the of the output table)
`name.format`	Format string for generating the column from the contrast vector (see details)

Details

To compare one specific factor level *A* against another level *B* in a particular column *COL* of the column annotation table, specify contrast=c("COL","A","B")

To compare all levels against a specific level *A* in a particular column *COL* of the column annotation table, specify contrast=c("COL","A")

To perform all pairwise comparisons of all levels from a particular column *COL* of the column annotation table, specify contrast=c("COL")

If the column *COL* only has two levels, all three are equivalent.

In all cases, if groups is not NULL, the columns annotation table is first split and contrasts are applied within all samples or cells with the same *group* factor level.

The format string specifies the column name in the generated contrast matrix (which is used as the *Analysis* name when calling `ApplyContrasts`, `LFC`, `PairwiseDESeq2`, etc.). The keywords *$GRP*, *$COL*, *$A* and *$B* are substituted by the respective elements of the contrast vector or the group this comparison refers to. By default, it is "$A vs $B" if group is NULL, and "$A vs $B.$GRP" otherwise.

The method for grandR objects simply calls the general method

For grandR objects, columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition="x").

Value

A data frame representig a contrast matrix to be used in `ApplyContrasts`, `LFC`, `PairwiseDESeq2`

See Also

`ApplyContrasts`, `LFC`, `PairwiseDESeq2`

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition","Time",Design$Replicate))

GetContrasts(sars,contrast="Condition")
# Compare all Mock vs. all SARS
GetContrasts(sars,contrast=c("Condition","SARS","Mock"))
# This direction of the comparison is more reasonable
GetContrasts(sars,contrast=c("Condition","SARS","Mock"),group="Time")
# Compare SARS vs Mock per time point
GetContrasts(sars,contrast=c("Time.original","no4sU"), group="Condition",no4sU=TRUE,
                                                 name.format="$A vs $B ($GRP)")
# Compare each sample against the respective no4sU sample

# See the differential-expression vignette for more examples!
```

`GetData` *Obtain a tidy table of values for a gene or a small set of genes*

Description

This is the main function to access slot data data from a particular gene (or a small set of genes) as a tidy table. If data for all genes must be retrieved (as a large matrix), use the `GetTable` function. For analysis results, use the `GetAnalysisTable` function.

Usage

```
GetData(
  data,
  mode.slot = DefaultSlot(data),
  columns = NULL,
  genes = Genes(data),
  by.rows = FALSE,
  coldata = TRUE,
  ntr.na = TRUE,
  name.by = "Symbol"
)
```

Arguments

`data`	A grandR object
`mode.slot`	Which kind of data to access (see details)
`columns`	A vector of columns (see details); all condition/cell names if NULL
`genes`	Restrict the output table to the given genes (this typically is a single gene, or very few genes)
`by.rows`	if TRUE, add rows if there are multiple genes / mode.slots; otherwise, additional columns are appended

`coldata`	Should the table contain the Coldata values as well (at the beginning)?
`ntr.na`	For columns representing a 4sU naive sample, should mode.slot *ntr*,*new.count* and *old.count* be 0,0 and count (ntr.na=FALSE; can be any other slot than count) or NA,NA and NA (ntr.na=TRUE)
`name.by`	A column name of Coldata(data). This is used as the colnames of the output table

Details

To refer to data slots, the mode.slot syntax can be used: Each name is either a data slot, or one of (new,old,total) followed by a dot followed by a slot. For new or old, the data slot value is multiplied by ntr or 1-ntr. This can be used e.g. to obtain the *new counts*.

If only one mode.slot and one gene is given, the output table contains one column (and potentially columns from Coldata) named *Value*. If one gene and multiple mode.slots are given, the columns are named according to the mode.slots. If one mode.slot and multiple genes are given, the columns are named according to the genes. If multiple genes and mode.slots are given, columns are named gene.mode.slot.

If by.rows=TRUE, the table is molten such that each row contains only one value (for one of the genes and for one of the mode.slots). If only one gene and one mode.slot is given, melting does not have an effect.

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment havin the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

Value

A data frame containing the desired values

See Also

GetTable,GetAnalysisTable,DefaultSlot,Genes

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))
GetData(sars,mode.slot="ntr",gene="MYC")
# one gene, one mode.slot
GetData(sars,mode.slot=c("count","ntr"),gene="MYC",coldata = FALSE)
# one gene, multiple mode.slots
GetData(sars,mode.slot=c("count","ntr"),gene=c("SRSF6","MYC"),by.rows=TRUE)
# multiple genes, multiple mode.slots, by rows
```

`GetDiagnosticParameters` *Describe parameters relevant to diagnostics*

Description

Many of the diagnostics functions expect (optional or mandatory) parameters that are described by this function

Usage

```
GetDiagnosticParameters(data)
```

Arguments

`data`	a grandR object

Value

a list with

- orientation: Sense or Antisense, only relevant to mismatches for strand unspecific data
- category: all available categories (Exonic/Intronic, genomes). Note that this might differ from what is available from GeneInfo(data,"Category"), since Grand3 might not have estimated NTRs for all categories!
- label: which nucleoside analogs have been used
- model: which model (binom or tbbinom) to inspect
- estimator: which estimator (joint or separate NTRs were estimated for subreads)

`GetMatrix` *Obtain a genes x values table as a large matrix*

Description

This is the main function to access slot data for all genes as a (potentially sparse) matrix.

Usage

```
GetMatrix(
  data,
  mode.slot = DefaultSlot(data),
  columns = NULL,
  genes = Genes(data),
  name.by = "Symbol",
  summarize = NULL
)
```

Arguments

`data`	A grandR object
`mode.slot`	Which kind of data to access (see details)
`columns`	which columns (i.e. samples or cells) to return (see details)
`genes`	Restrict the output table to the given genes
`name.by`	A column name of Coldata(data). This is used as the rownames of the output table
`summarize`	Should replicates by summarized? see details

Details

To refer to data slots, the mode.slot syntax can be used: It is either a data slot, or one of (new,old,total) followed by a dot followed by a slot. For new or old, the data slot value is multiplied by ntr or 1-ntr. This can be used e.g. to obtain the *new counts*.

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment havin the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

The summarization parameter can only be specified if columns is NULL. It is either a summarization matrix (GetSummarizeMatrix) or TRUE (in which case GetSummarizeMatrix(data) is called). If there a NA values, they are imputed as the mean per group!

Value

A (potentially) sparse matrix containing the desired values

See Also

GetData,GetAnalysisTable,DefaultSlot,Genes,GetSummarizeMatrix

---

`GetPairContrasts` *Create a contrast matrix for two given conditions*

Description

Each column of a contrast matrix represents a pairwise comparison of all samples or cells of a grandR object (or a column annotation table). Elements being 1 are contrasted vs. elements being -1 (and all 0 are irrelevant for this comparison).

Usage

```
GetPairContrasts(d, name, A, B)
```

Arguments

`d`	A grandR object or a column annotation table
`name`	the name of the contrast
`A`	definition of the condition of interest, see details
`B`	definition of the reference condition , see details

Details

This creates a contrast A vs B (i.e. the fold change would be A/B)

Columns that belong to A or B can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition="x").

Value

A data frame with a single column representig a contrast matrix to be used in `ApplyContrasts`, `LFC`, `PairwiseDESeq2`

See Also

`ApplyContrasts`, `LFC`, `PairwiseDESeq2`, `GetContrasts`

`GetSignificantGenes` *Significant genes*

Description

Return significant genes for this grandR object

Usage

```
GetSignificantGenes(
  data,
  analysis = NULL,
  regex = TRUE,
  criteria = NULL,
  as.table = FALSE,
  use.symbols = TRUE,
  gene.info = TRUE
)
```

Arguments

`data`	the grandR object
`analysis`	the analysis to use, can be more than one and can be regexes (see details)
`regex`	interpret analyses as regex?
`criteria`	the criteria used to define what significant means; if NULL, Q<0.05 & abs(LFC)>=1 is used; can use the column names of the analysis table as variables, should be a logical or numerical value per gene (see Details)
`as.table`	return a table
`use.symbols`	return them as symbols (gene ids otherwise)
`gene.info`	add gene infos to the output table

Details

The analysis parameter (just like for GetAnalysisTable can be a regex (that will be matched against all available analysis names). It can also be a vector (of regexes). Be careful with this, if more than one table e.g. with column LFC ends up in here, only the first is used (if criteria=LFC).

The criteria parameter can be used to define how analyses are performed. If criteria is a logical, it obtains significant genes defined by cut-offs (e.g. on q value and LFC). If it is a numerical, all genes are returned sorted (descendingly) by this value. The columns of the given analysis table(s) can be used to build this expression.

Value

a vector of gene names (or symbols), or a table

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c(Design$Condition,Design$dur.4sU,Design$Replicate))
sars <- subset(sars,Coldata(sars,Design$dur.4sU)==2)
sars<-LFC(sars,mode="total",contrasts=GetContrasts(sars,contrast=c("Condition","Mock")))
GetSignificantGenes(sars,criteria=LFC>1)
```

`GetSummarizeMatrix` *Create a summarize matrix*

Description

If this matrix is multiplied with a count table (e.g. obtained by `GetTable`), either the average (average=TRUE) or the sum (average=FALSE) of all columns (samples or cells) belonging to the same `Condition` is computed.

Usage

```
GetSummarizeMatrix(x, ...)

## S3 method for class 'grandR'
GetSummarizeMatrix(x, no4sU = FALSE, columns = NULL, average = TRUE, ...)

## Default S3 method:
GetSummarizeMatrix(x, subset = NULL, average = TRUE, ...)
```

Arguments

`x`	A grandR object or a named vector (the names indicate the sample names, the value the conditions to be summarized)
`...`	further arguments to be passed to or from other methods.
`no4sU`	Use no4sU columns (TRUE) or not (FALSE)
`columns`	which columns (i.e. samples or cells) to return (see details)
`average`	matrix to compute the average (TRUE) or the sum (FALSE)
`subset`	logical vector of which elements of the vector v to use (or NULL: use all)

Details

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition="x").

The method for grandR object simply calls the general method

Value

A matrix to be multiplied with a count table

See Also

GetTable

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition",Design$dur.4sU,Design$Replicate))

GetSummarizeMatrix(sars)
head(as.matrix(GetTable(sars)) %*% GetSummarizeMatrix(sars))  # average by matrix multiplication
head(GetTable(sars,summarize = TRUE))                          # shortcut, does the same

# See the data-matrices-and-analysis-results vignette for more examples!
```

`GetTable` *Obtain a genes x values table*

Description

This is the main function to access slot data for all genes as a large matrix. If data from a particular gene (or a small set of genes) must be retrieved, use the `GetData` function. For analysis results, use the `GetAnalysisTable` function.

Usage

```
GetTable(
  data,
  type = DefaultSlot(data),
  columns = NULL,
  genes = Genes(data),
  ntr.na = TRUE,
  gene.info = FALSE,
  summarize = NULL,
  prefix = NULL,
  name.by = "Symbol"
)
```

Arguments

`data`	A grandR object
`type`	Either a mode.slot (see details) or a regex to be matched against analysis names. Can also be a vector
`columns`	A vector of columns (either condition/cell names if the type is a mode.slot, or names in the output table from an analysis; use Columns(data,<analysis>) to learn which columns are available); all condition/cell names if NULL
`genes`	Restrict the output table to the given genes
`ntr.na`	For columns representing a 4sU naive sample, should types *ntr*,*new.count* and *old.count* be 0,0 and count (ntr.na=FALSE; can be any other slot than count) or NA,NA and NA (ntr.na=TRUE)
`gene.info`	Should the table contain the GeneInfo values as well (at the beginning)?
`summarize`	Should replicates by summarized? see details
`prefix`	Prepend each column in the output table (except for the gene.info columns) by the given prefix
`name.by`	A column name of Coldata(data). This is used as the rownames of the output table

Details

This is a convenience wrapper for GetData (values from data slots) and GetAnalysisTable (values from analyses). Types can refer to any of the two (and can be mixed). If there are types from both data and analyses, columns must be NULL. Otherwise columns must either be condition/cell names (if type refers to one or several data slots), or regular expressions to match against the names in the analysis tables.

Columns definitions for data slots can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

To refer to data slots via `type`, the mode.slot syntax can be used: Each name is either a data slot, or one of (new,old,total) followed by a dot followed by a slot. For new or old, the data slot value is multiplied by ntr or 1-ntr. This can be used e.g. to obtain the *new counts*.

The summarization parameter can only be specified if columns is NULL. It is either a summarization matrix (GetSummarizeMatrix) or TRUE (in which case GetSummarizeMatrix(data) is called). If there a NA values, they are imputed as the mean per group!

Value

A data frame containing the desired values

See Also

GetData,GetAnalysisTable,DefaultSlot,Genes,GetSummarizeMatrix

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition",Design$dur.4sU,Design$Replicate))
sars <- Normalize(FilterGenes(sars))

head(GetTable(sars))
# DefaultSlot values, i.e. size factor normalized read counts for all samples
head(GetTable(sars,summarize=TRUE))
# DefaultSlot values averaged over the two conditions
head(GetTable(sars,type="new.count",columns=!no4sU))
# Estimated counts for new RNA for all samples with 4sU

sars<-LFC(sars,contrasts=GetContrasts(sars,group = "duration.4sU"))
head(GetAnalysisTable(sars,columns="LFC"))
# Estimated fold changes SARS vs Mock for each time point
```

`grandR` *Create a grandR object and retrieve basic information*

Description

The grandR object contains

- metadata about the origin (file/url) of the GRAND-SLAM output
- the current state (e.g., what is the current default slot) of the grandR object
- a gene info table (i.e. metadata for the rows of the data matrices)
- a column annotation table (i.e. metadata for the columns of the data matrices)
- several data matrices for read counts, normalized expression values, NTRs, etc. (genes x samples or genes x cells; stored in so-called *slots*)
- potentially several analysis output tables (for kinetic modeling, differential gene expression testing)

Usually, this constructor is not invoked directly (but by `ReadGRAND` or `SimulateTimeCourse`).

Usage

```
grandR(
  prefix = parent$prefix,
  gene.info = parent$gene.info,
  slots = parent$data,
  coldata = parent$coldata,
  metadata = parent$metadata,
  analyses = NULL,
  plots = NULL,
  parent = NULL
)

Title(data)

IsSparse(data)

## S3 method for class 'grandR'
dim(x)

is.grandR(x)

## S3 method for class 'grandR'
dimnames(x)

## S3 method for class 'grandR'
print(x, ...)
```

```
Metadata(x, ...)

## S3 method for class 'grandR'
subset(x, columns, reorder = TRUE, ...)

## S3 method for class 'grandR'
split(x, f = Design$Condition, drop = FALSE, ...)

RenameColumns(data, map = NULL, fun = NULL)

SwapColumns(data, s1, s2)

## S3 method for class 'grandR'
merge(..., list = NULL, column.name = Design$Origin)
```

Arguments

`prefix`	Can either be the prefix used to call GRAND-SLAM with, or the main output file ($prefix.tsv.gz); if the RCurl package is installed, this can also be a URL
`gene.info`	a data frame with metadata for all genes
`slots`	A list of matrices representing the slots
`coldata`	a data frame with metadata for all samples (or cells)
`metadata`	a metadata list
`analyses`	the analyses list
`plots`	the plots list
`parent`	A parent object containing default values for all other parameters (i.e. all parameters not specified are obtained from this object)
`data, x`	a grandR object
`...`	further arguments to be passed to or from other methods.
`columns`	which columns (i.e. samples or cells) to return (see details)
`reorder`	reorder all factors in coldata (if columns for subset define a different order)
`f`	The name of the annotation table according to which the object is split or the new annotation table column name denoting the origin after merging
`drop`	unused
`map`	named list or vector representing a lookup table (names are current column names)
`fun`	a function that maps a vector of names to a new vector of names
`s1, s2`	column names
`list`	a list of grandR objects
`column.name`	a new name for the Coldata table to annotate the merged objects

Details

The dimensions (nrow, ncol) of the grandR object are considered to be the dimensions of the data tables, i.e. `nrow(data)` provides the number of genes and `ncol(data)` the number of samples (or cells).

Currently, the object is implemented as a list of the above mentioned items. This implementation is subject to change. Make sure to use accessor functions to obtain the information you want.

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment havin the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

Value

A grandR object containing the read counts, NTRs, information on the NTR posterior distribution (alpha,beta) and potentially additional information of all genes detected by GRAND-SLAM

Functions

Title Obtain a useful title for the project (from the prefix parameter)

dim Obtain the dimensions (genes x samples or genes x cells)

is Check whether it is a grandR object

dimnames Obtain the row and column names of this object (genes x samples or genes x cells)

print Print information on this grandR object

subset Create a new grandR object with a subset of the columns (use `FilterGenes` to subset on genes)

split Split the grandR object into a list of multiple grandR objects (according to the levels of an annotation table column)

RenameColumns Rename the column names according to a lookup table (map) or a function (invoked on the current names)

SwapColumns Swap two columns (samples or cells); this is what you do if samples were mislabeled!

Metadata Obtain global metadata

merge Merge several grandR objects into one

See Also

Slots, DefaultSlot, Genes, GeneInfo, Coldata, GetTable, GetData, Analyses, GetAnalysisTable

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))
# this is part of the corona data from Finkel et al.
dim(sars)
head(rownames(sars))
```

`IsParallel` *Checks for parallel execution*

Description

Checks for parallel execution

Usage

```
IsParallel()
```

Value

whether or not parallelism is activated

`LFC` *Estimation of log2 fold changes*

Description

Estimate the log fold changes based on a contrast matrix, requires the LFC package.

Usage

```
LFC(
  data,
  name.prefix = mode,
  contrasts,
  slot = "count",
  LFC.fun = lfc::PsiLFC,
  mode = "total",
  normalization = NULL,
  compute.M = TRUE,
  genes = NULL,
  verbose = FALSE,
  ...
)
```

Arguments

`data`	the grandR object
`name.prefix`	the prefix for the new analysis name; a dot and the column names of the contrast matrix are appended; can be NULL (then only the contrast matrix names are used)

`contrasts`	contrast matrix that defines all pairwise comparisons, generated using GetContrasts
`slot`	the slot of the grandR object to take the data from; for PsiLFC, this really should be "count"!
`LFC.fun`	function to compute log fold changes (default: PsiLFC, other viable option: NormLFC)
`mode`	compute LFCs for "total", "new", or "old" RNA
`normalization`	normalize on "total", "new", or "old" (see details)
`compute.M`	also compute the mean expression (in log10 space)
`genes`	restrict analysis to these genes; NULL means all genes
`verbose`	print status messages?
`...`	further arguments forwarded to LFC.fun

Details

Both PsiLFC and NormLFC) by default perform normalization by subtracting the median log2 fold change from all log2 fold changes. When computing LFCs of new RNA, it might be sensible to normalize w.r.t. to total RNA, i.e. subtract the median log2 fold change of total RNA from all the log2 fold change of new RNA. This can be accomplished by setting mode to "new", and normalization to "total"!

Normalization can also be a mode.slot! Importantly, do not specify a slot containing normalized values, but specify a slot of unnormalized values (which are used to compute the size factors for normalization!) Can also be a numeric vector of size factors with the same length as the data as columns. Then each value is divided by the corresponding size factor entry.

Value

a new grandR object including a new analysis table. The columns of the new analysis table are

`"LFC"`	the log2 fold change

See Also

PairwiseDESeq2,GetContrasts

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c(Design$Condition,Design$dur.4sU,Design$Replicate))
sars <- subset(sars,Coldata(sars,Design$dur.4sU)==2)
sars<-LFC(sars,mode="total",contrasts=GetContrasts(sars,contrast=c("Condition","Mock")))
sars<-LFC(sars,mode="new",normalization="total",
                            contrasts=GetContrasts(sars,contrast=c("Condition","Mock")))
head(GetAnalysisTable(sars))
```

`LikelihoodRatioTest` *Compute a likelihood ratio test.*

Description

The test is computed on any of total/old/new counts using DESeq2 based on two nested models specified using formulas.

Usage

```
LikelihoodRatioTest(
  data,
  name = "LRT",
  mode = "total",
  slot = "count",
  normalization = mode,
  target = ~Condition,
  background = ~1,
  columns = NULL,
  logFC = FALSE,
  verbose = FALSE
)
```

Arguments

`data`	A grandR object
`name`	the user defined analysis name to store the results
`mode`	either "total", "new" or "old"
`slot`	which slot to use (should be a count slot, not normalized values)
`normalization`	normalize on "total", "new", or "old" (see details)
`target`	formula specifying the target model (you can use any column name from the `Coldata(data)`)
`background`	formula specifying the background model (you can use any column name from the `Coldata(data)`)
`columns`	logical vector of which columns (samples or cells) to use (or NULL: use all)
`logFC`	compute and add the log2 fold change as well
`verbose`	Print status updates

Details

This is a convenience wrapper around the likelihood ratio test implemented in DESeq2.

DESeq2 by default performs size factor normalization. When computing differential expression of new RNA, it might be sensible to normalize w.r.t. to total RNA, i.e. use the size factors computed from total RNA instead of computed from new RNA. This can be accomplished by setting mode to "new", and normalization to "total"!

Value

a new grandR object including a new analysis table. The columns of the new analysis table are

`"M"`	the base mean
`"S"`	the difference in deviance between the reduced model and the full model
`"P"`	the likelihood ratio test P value
`"Q"`	same as P but Benjamini-Hochberg multiple testing corrected
`"LFC"`	the log2 fold change for the target model (only with the logFC parameter set to TRUE)

`ListGeneSets` *List available gene sets*

Description

Helper function to return a table with all available gene sets for AnalyzeGeneSets.

Usage

```
ListGeneSets()
```

Details

This is a convenience wrapper for msigdbr_collections.

Value

the gene set table; use the values in the category and subcategory columns for the corresponding parameters of AnalyzeGeneSets

See Also

AnalyzeGeneSets

MakeColdata *Extract an annotation table from a formatted names vector*

Description

If columns (i.e. sample or cell) follow a specific naming pattern, this can be used to conveniently set up an annotation table.

Usage

```
MakeColdata(
  names,
  design,
  semantics = DesignSemantics(),
  rownames = TRUE,
  keep.originals = TRUE
)
```

Arguments

`names`	Formatted names vector (see details)
`design`	Titles for the columns of the annotation table
`semantics`	Additional semantics to apply to given annotations (see details)
`rownames`	Add rownames to the annotation table
`keep.originals`	To not discard the original values for all annotations where semantics were applied

Details

The names have to contain dots (.) to separate the fields for the column annotation table. E.g. the name *Mock.4h.A* will be split into the fields *Mock*, *4h* and *A*. For such names, a design vector of length 3 has to be given, that describes the meaning of each field. A reasonable design vector for the example would be `c("Treatment","Time","Replicate")`. Some names are predefined in the list Design.

The names given in the design vector might even have additional semantics: E.g. for the name *duration.4sU* the values are interpreted (e.g. 4h is converted into the number 4, or 30min into 0.5, or no4sU into 0).

Semantics can be user-defined via the *semantics* list: For each name in the design vector matching to a name in this list, the corresponding function in the list is run. Functions must accept 2 parameters, the first is the original column in the annotation table, the second the original name. The function must return a data.frame with the number of rows matching to the annotation table. In most cases it is easier to manipulate the returned data frame instead of changing the semantics. However, the build-in semantics provide a convenient way to reduce this kind of manipulation in most cases.

Value

A data frame representing the annotation table

See Also

ReadGRAND,DesignSemantics,Coldata

Examples

```
coldata <- MakeColdata(c("Mock.0h.A","Mock.0h.B","Mock.2h.A","Mock.2h.B"),
                                   design=c("Cell",Design$dur.4sU,Design$Replicate))
```

`MAPlot` *Make an MA plot*

Description

Plot average expression vs. log2 fold changes

Usage

```
MAPlot(
  data,
  analysis = Analyses(data)[1],
  aest = aes(),
  p.cutoff = 0.05,
  lfc.cutoff = 1,
  label.numbers = TRUE,
  highlight = NULL,
  label = NULL,
  label.repel = 1
)
```

Arguments

`data`	the grandR object that contains the data to be plotted
`analysis`	the analysis to plot (default: first analysis)
`aest`	parameter to set visual attributes of the plot
`p.cutoff`	p-value cutoff (default: 0.05)
`lfc.cutoff`	log fold change cutoff (default: 1)
`label.numbers`	if TRUE, label the number of genes
`highlight`	highlight these genes; can be either numeric indices, gene names, gene symbols or a logical vector (see details)
`label`	label these genes; can be either numeric indices, gene names, gene symbols or a logical vector (see details)
`label.repel`	force to repel labels from points and each other (increase if labels overlap)

Value

a ggplot object

Normalize *Normalization*

Description

Normalizes data in a grandR object and puts the normalized data into a new slot

Usage

```
Normalize(
  data,
  genes = Genes(data),
  name = "norm",
  slot = "count",
  set.to.default = TRUE,
  size.factors = NULL,
  return.sf = FALSE
)

NormalizeFPKM(
  data,
  genes = Genes(data),
  name = "fpkm",
  slot = "count",
  set.to.default = TRUE,
  tlen = GeneInfo(data, "Length")
)

NormalizeRPM(
  data,
  genes = Genes(data),
  name = "rpm",
  slot = "count",
  set.to.default = TRUE,
  factor = 1e+06
)

NormalizeTPM(
  data,
  genes = Genes(data),
  name = "tpm",
  slot = "count",
  set.to.default = TRUE,
  tlen = GeneInfo(data, "Length")
)
```

Arguments

`data`	the grandR object
`genes`	compute the normalization w.r.t. these genes (see details)
`name`	the name of the new slot for the normalized data
`slot`	the name of the slot for the data to normalize
`set.to.default`	set the new slot as the default slot
`size.factors`	numeric vector; if not NULL, use these size factors instead of computing size factors
`return.sf`	return the size factors and not a grandR object
`tlen`	the transcript lengths (for FPKM and TPM)
`factor`	the rpm factor (default: 1 (m)illion)

Details

Normalize will perform DESeq2 normalization, i.e. it will use estimateSizeFactorsForMatrix to estimate size factors, and divide each value by this. If genes are given, size factors will be computed only w.r.t. these genes (but then all genes are normalized).

NormalizeFPKM will compute fragments per kilobase and million mapped reads. If genes are given, the scaling factor will only be computed w.r.t. these genes (but then all genes are normalized).

NormalizeRPM will compute reads per million mapped reads. If genes are given, the scaling factor will only be computed w.r.t. these genes (but then all genes are normalized).

NormalizeTPM will compute transcripts per million mapped reads. If genes are given, the scaling factor will only be computed w.r.t. these genes (but then all genes are normalized).

Genes can be referred to by their names, symbols, row numbers in the gene table, or a logical vector referring to the gene table rows.

Value

a new grandR object with a new data slot

See Also

`NormalizeBaseline`

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))

sars <- Normalize(sars)
DefaultSlot(sars)
```

`NormalizeBaseline` *Normalization to a baseline*

Description

Normalizes data in a grandR object to a baseline and puts the normalized data into a new slot

Usage

```
NormalizeBaseline(
  data,
  baseline = FindReferences(data, reference = Condition == levels(Condition)[1]),
  name = "baseline",
  slot = DefaultSlot(data),
  set.to.default = FALSE,
  LFC.fun = lfc::PsiLFC,
  ...
)
```

Arguments

`data`	the grandR object
`baseline`	matrix defining the corresponding baseline (row) for each column (sample or cell; see details)
`name`	the name of the new slot for the normalized data
`slot`	the name of the slot for the data to normalize
`set.to.default`	set the new slot as the default slot
`LFC.fun`	either NormLFC or PsiLFC from the lfc package
`...`	forwarded to LFC.fun

Details

Baseline normalization computes the log2 fold change for a column (i.e. sample or cell) to a baseline columns (or several baseline columns). This is by default done using the `PsiLFC` function from the lfc package, which, by default, also normalizes log2 fold changes by adding a constant such that the median is zero.

Baselines are defined by a square logical matrix, defining for each sample or cell of the grandR object, represented by the column of the matrix, which samples or cells are indeed the baseline (represented by the rows). Such matrices can conveniently be obtained by `FindReferences`.

Value

a new grandR object with an additional slot

See Also

`Normalize`,`FindReferences`

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))
blmat <- FindReferences(sars,reference = duration.4sU==0, group = "Cell")
# the Mock.no4sU or SARS.no4sU sample are the baselines for each sample
sars <- NormalizeBaseline(sars,baseline=blmat)
head(GetTable(sars,type="baseline"))
```

`Pairwise` *Log2 fold changes and Wald tests for differential expression*

Description

This function is a shortcut for first calling PairwiseDESeq2 and then LFC.

Usage

```
Pairwise(
  data,
  name.prefix = mode,
  contrasts,
  LFC.fun = lfc::PsiLFC,
  slot = "count",
  mode = "total",
  normalization = mode,
  genes = NULL,
  verbose = FALSE
)
```

Arguments

`data`	the grandR object
`name.prefix`	the prefix for the new analysis name; a dot and the column names of the contrast matrix are appended; can be NULL (then only the contrast matrix names are used)
`contrasts`	contrast matrix that defines all pairwise comparisons, generated using GetContrasts
`LFC.fun`	function to compute log fold changes (default: PsiLFC, other viable option: NormLFC)
`slot`	the slot of the grandR object to take the data from; should contain counts!
`mode`	compute LFCs for "total", "new", or "old" RNA

`normalization`	normalize on "total", "new", or "old" (see details)
`genes`	restrict analysis to these genes; NULL means all genes
`verbose`	print status messages?

Details

Both PsiLFC and NormLFC) by default perform normalization by subtracting the median log2 fold change from all log2 fold changes. When computing LFCs of new RNA, it might be sensible to normalize w.r.t. to total RNA, i.e. subtract the median log2 fold change of total RNA from all the log2 fold change of new RNA. This can be accomplished by setting mode to "new", and normalization to "total"!

Normalization can also be a mode.slot! Importantly, do not specify a slot containing normalized values, but specify a slot of unnormalized values (which are used to compute the size factors for normalization!) Can also be a numeric vector of size factors with the same length as the data as columns. Then each value is divided by the corresponding size factor entry.

Value

a new grandR object including a new analysis table. The columns of the new analysis table are

`"M"`	the base mean
`"S"`	the log2FoldChange divided by lfcSE
`"P"`	the Wald test P value
`"Q"`	same as P but Benjamini-Hochberg multiple testing corrected
`"LFC"`	the log2 fold change

See Also

PairwiseDESeq2,GetContrasts

`PairwiseDESeq2` *Perform Wald tests for differential expression*

Description

Apply DESeq2 for comparisons defined in a contrast matrix, requires the DESeq2 package.

Usage

```
PairwiseDESeq2(
  data,
  name.prefix = mode,
  contrasts,
  separate = FALSE,
  mode = "total",
  slot = "count",
```

```
    normalization = NULL,
    logFC = FALSE,
    genes = NULL,
    verbose = FALSE
)
```

Arguments

`data`	the grandR object
`name.prefix`	the prefix for the new analysis name; a dot and the column names of the contrast matrix are appended; can be NULL (then only the contrast matrix names are used)
`contrasts`	contrast matrix that defines all pairwise comparisons, generated using GetContrasts
`separate`	model overdispersion separately for all pairwise comparison (TRUE), or fit a single model per gene, and extract contrasts (FALSE)
`mode`	compute LFCs for "total", "new", or "old" RNA
`slot`	which slot to use (should be a count slot, not normalized values)
`normalization`	normalize on "total", "new", or "old" (see details)
`logFC`	compute and add the log2 fold change as well
`genes`	restrict analysis to these genes; NULL means all genes
`verbose`	print status messages?

Details

DESeq2 by default performs size factor normalization. When computing differential expression of new RNA, it might be sensible to normalize w.r.t. to total RNA, i.e. use the size factors computed from total RNA instead of computed from new RNA. This can be accomplished by setting mode to "new", and normalization to "total"!

Normalization can also be a mode.slot! Importantly, do not specify a slot containing normalized values, but specify a slot of unnormalized values (which are used to compute the size factors for normalization!) Can also be a numeric vector of size factors with the same length as the data as columns. Then each value is divided by the corresponding size factor entry.

Value

a new grandR object including a new analysis table. The columns of the new analysis table are

`"M"`	the base mean
`"S"`	the log2FoldChange divided by lfcSE
`"P"`	the Wald test P value
`"Q"`	same as P but Benjamini-Hochberg multiple testing corrected
`"LFC"`	the log2 fold change (only with the logFC parameter set to TRUE)

See Also

LFC,GetContrasts

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c(Design$Condition,Design$dur.4sU,Design$Replicate))
sars <- subset(sars,Coldata(sars,Design$dur.4sU)==2)
sars<-PairwiseDESeq2(sars,mode="total",
                            contrasts=GetContrasts(sars,contrast=c("Condition","Mock")))
sars<-PairwiseDESeq2(sars,mode="new",normalization="total",
                            contrasts=GetContrasts(sars,contrast=c("Condition","Mock")))
head(GetAnalysisTable(sars,column="Q"))
```

`PlotAnalyses` *Convenience function to make the same type of plot for multple analyses.*

Description

Convenience function to make the same type of plot for multple analyses.

Usage

```
PlotAnalyses(data, plot.fun, analyses = Analyses(data), add = NULL, ...)
```

Arguments

`data`	the grandR object that contains the data to be plotted
`plot.fun`	the plottinf function to apply
`analyses`	the analyses to plot (default: all)
`add`	additional ggplot (e.g., geoms) objects to add
`...`	passed further to plot.fun

Value

ggplot objects

`PlotConversionFreq` *Diagnostic plot for conversion frequencies*

Description

This is the second diagnostic plot (estimated conversions) generated by GRAND3.

Usage

```
PlotConversionFreq(data, category, sample = NULL, max.columns = 120)
```

Arguments

`data`	the grandR object
`category`	show a specific category (see GetDiagnosticParameters); cannot be NULL
`sample`	compare subreads for a specific sample; can be NULL, then compare all samples per subread
`max.columns`	if there are more columns (samples for bulk, cells for single cell) than this, show boxplots instead of points

Details

Show the percentage of all conversion types for all samples. In contrast to mismatches (see PlotMismatchPositionForSample and PlotMismatchPositionForType), the correct strand is already inferred for conversions, i.e. conversions refer to actual conversion events on RNA, whereas mismatches are observed events in mapped reads.

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotGeneGroupsBars` *Plot gene values as bars*

Description

Plot old and new RNA of a gene in a row.

Usage

```
PlotGeneGroupsBars(
  data,
  gene,
  slot = DefaultSlot(data),
  columns = NULL,
  show.CI = FALSE,
  xlab = NULL,
  transform = NULL
)
```

Arguments

`data`	the grandR object to get the data to be plotted from
`gene`	the gene to plot
`slot`	the slot of the grandR object to get the data from
`columns`	which columns (i.e. samples or cells) to show (see details)
`show.CI`	show confidence intervals; one of TRUE/FALSE (default: FALSE)
`xlab`	The names to show at the x axis;
`transform`	function that is called on the data frame directly before plotting (can be NULL)

Details

xlab can be given as a character vector or an expression that evaluates into a character vector. The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently it.

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

Value

a ggplot object.

See Also

GetData, PlotGeneTotalVsNtr,PlotGeneOldVsNew,PlotGeneGroupsBars

---

`PlotGeneGroupsPoints` *Plot gene groups as points*

Description

Plot either old, new or total RNA of a gene in a row, per condition.

Usage

```
PlotGeneGroupsPoints(
  data,
  gene,
  group = "Condition",
  mode.slot = DefaultSlot(data),
  columns = NULL,
  log = TRUE,
  show.CI = FALSE,
  aest = NULL,
  size = 2,
  transform = NULL
)
```

Arguments

`data`	the grandR object to get the data to be plotted from
`gene`	the gene to plot
`group`	how to group the genes (default: Condition)
`mode.slot`	the mode.slot of the grandR object to get the data from
`columns`	which columns (i.e. samples or cells) to show (see details)
`log`	show the y axis in log scale
`show.CI`	show confidence intervals; one of TRUE/FALSE (default: FALSE)
`aest`	parameter to set the visual attributes of the plot
`size`	the point size used for plotting; overridden if size is defined via aest
`transform`	function that is called on the data frame directly before plotting (can be NULL)

Details

The value of the aest parameter must be an *Aesthetic mapping* as generated by aes.

To refer to data slots, the mode.slot syntax can be used: Each name is either a data slot, or one of (new,old,total) followed by a dot followed by a slot. For new or old, the data slot value is multiplied by ntr or 1-ntr. This can be used e.g. to obtain the *new counts*.

The table used for plotting is the table returned by GetData with coldata set to TRUE, i.e. you can use all names from the Coldata table for aest.

By default, aest is set to aes(color=Condition,shape=Replicate) (if both Condition and Replicate are names in the Coldata table).

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

Value

a ggplot object.

See Also

GetData, PlotGeneTotalVsNtr,PlotGeneOldVsNew,PlotGeneGroupsBars

---

`PlotGeneOldVsNew` *Gene plot comparing old vs new RNA*

Description

Plot the old vs new RNA values of a gene

Usage

```
PlotGeneOldVsNew(
  data,
  gene,
  slot = DefaultSlot(data),
  columns = NULL,
  log = TRUE,
  show.CI = FALSE,
  aest = NULL,
  size = 2
)
```

Arguments

`data`	the grandR object to get the data to be plotted from
`gene`	the gene to plot
`slot`	the slot of the grandR object to get the data from
`columns`	which columns (i.e. samples or cells) to show (see details)
`log`	show both axes in log scale
`show.CI`	show confidence intervals; one of TRUE/FALSE (default: FALSE)
`aest`	parameter to set the visual attributes of the plot
`size`	the point size used for plotting; overridden if size is defined via aest

Details

The value of the aest parameter must be an *Aesthetic mapping* as generated by aes.

The table used for plotting is the table returned by GetData with coldata set to TRUE, i.e. you can use all names from the Coldata table for aest.

By default, aest is set to aes(color=Condition,shape=Replicate) (if both Condition and Replicate are names in the Coldata table).

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

Value

a ggplot object.

See Also

GetData, PlotGeneTotalVsNtr,PlotGeneGroupsPoints,PlotGeneGroupsBars

---

`PlotGeneProgressiveTimecourse`

Plot progressive labeling timecourses

Description

Plot the abundance of new and old RNA and the fitted model over time for a single gene.

Usage

```
PlotGeneProgressiveTimecourse(
  data,
  gene,
  slot = DefaultSlot(data),
  time = Design$dur.4sU,
  type = c("nlls", "ntr", "lm"),
  exact.tics = TRUE,
  show.CI = FALSE,
  return.tables = FALSE,
  ...
)
```

Arguments

`data`	a grandR object
`gene`	the gene to be plotted
`slot`	the data slot of the observed abundances

`time`	the labeling duration column in the column annotation table
`type`	how to fit the model (see FitKinetics)
`exact.tics`	use axis labels directly corresponding to the available labeling durations?
`show.CI`	show confidence intervals; one of TRUE/FALSE (default: FALSE)
`return.tables`	also return the tables used for plotting
`...`	given to the fitting procedures

Details

For each `Condition` there will be one panel containing the values and the corresponding model fit.

Value

either a ggplot object, or a list containing all tables used for plotting and the ggplot object.

See Also

FitKineticsGeneNtr, FitKineticsGeneLeastSquares, FitKineticsGeneLogSpaceLinear

---

`PlotGeneSnapshotTimecourse`

Gene plot for snapshot timecourse data

Description

Plot the total RNA expression vs the new-to-total RNA ratio for a gene

Usage

```
PlotGeneSnapshotTimecourse(
  data,
  gene,
  time = Design$dur.4sU,
  mode.slot = DefaultSlot(data),
  columns = NULL,
  average.lines = TRUE,
  exact.tics = TRUE,
  log = TRUE,
  show.CI = FALSE,
  aest = NULL,
  size = 2
)
```

Arguments

`data`	the grandR object to get the data to be plotted from
`gene`	the gene to plot
`time`	the times to show on the x axis (see details)
`mode.slot`	the mode.slot of the grandR object to get the data from
`columns`	which columns (i.e. samples or cells) to show (see details)
`average.lines`	add average lines?
`exact.tics`	use axis labels directly corresponding to the available temporal values?
`log`	show the y axis in log scale
`show.CI`	show confidence intervals; one of TRUE/FALSE (default: FALSE)
`aest`	parameter to set the visual attributes of the plot
`size`	the point size used for plotting; overridden if size is defined via aest

Details

The x axis of this plot will show a temporal dimension. The time parameter defines a name in the Coldata table containing the temporal values for each sample.

The value of the aest parameter must be an *Aesthetic mapping* as generated by aes.

The table used for plotting is the table returned by GetData with coldata set to TRUE, i.e. you can use all names from the Coldata table for aest.

By default, aest is set to aes(color=Condition,shape=Replicate) (if both Condition and Replicate are names in the Coldata table).

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

Value

a ggplot object.

See Also

GetData, PlotGeneOldVsNew,PlotGeneGroupsPoints,PlotGeneGroupsBars

---

`PlotGeneTotalVsNtr` *Gene plot comparing total RNA vs the NTR*

Description

Plot the total RNA expression vs the new-to-total RNA ratio for a gene

Usage

```
PlotGeneTotalVsNtr(
  data,
  gene,
  slot = DefaultSlot(data),
  columns = NULL,
  log = TRUE,
  show.CI = FALSE,
  aest = NULL,
  size = 2
)
```

Arguments

`data`	the grandR object to get the data to be plotted from
`gene`	the gene to plot
`slot`	the slot of the grandR object to get the data from
`columns`	which columns (i.e. samples or cells) to show (see details)
`log`	show the x axis (total RNA) in log scale
`show.CI`	show confidence intervals; one of TRUE/FALSE (default: FALSE)
`aest`	parameter to set the visual attributes of the plot
`size`	the point size used for plotting; overridden if size is defined via aest

Details

The value of the aest parameter must be an *Aesthetic mapping* as generated by aes.

The table used for plotting is the table returned by GetData with coldata set to TRUE, i.e. you can use all names from the Coldata table for aest.

By default, aest is set to aes(color=Condition,shape=Replicate) (if both Condition and Replicate are names in the Coldata table).

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

Value

a ggplot object.

See Also

GetData, PlotGeneOldVsNew,PlotGeneGroupsPoints,PlotGeneGroupsBars

---

PlotHeatmap *Create heatmaps from grandR objects*

Description

Convenience method to compare among more two variables (slot data or analyses results).

Usage

```
PlotHeatmap(
  data,
  type = DefaultSlot(data),
  columns = NULL,
  genes = NULL,
  summarize = NULL,
  transform = "Z",
  cluster.genes = TRUE,
  cluster.columns = FALSE,
  label.genes = NULL,
  xlab = NULL,
  breaks = NULL,
  colors = NULL,
  title = NULL,
  return.matrix = FALSE,
  na.to = NA,
  ...
)
```

Arguments

`data`	the grandR object that contains the data to plot
`type`	Either a mode.slot (see details) or a regex to be matched against analysis names. Can also be a vector
`columns`	a vector of columns (either condition/cell names if the type is a mode.slot, or names in the output table from an analysis; use Columns(data,<analysis>) to learn which columns are available); all condition/cell names if NULL
`genes`	the genes to be included in the plot (default: all genes)
`summarize`	Should replicates by summarized? Can only be specified if columns is NULL; either a summarization matrix (GetSummarizeMatrix) or TRUE (in which case GetSummarizeMatrix(data) is called)
`transform`	apply a transformation to the selected data; can be a function, or a character (see details)

`cluster.genes`	should genes be clustered?
`cluster.columns`	should samples (or cells) be clustered?
`label.genes`	should genes be labeled?
`xlab`	The names to show at the x axis (only works if type is a single slot)
`breaks`	vector of color breaks; can be NULL (see details)
`colors`	an RColorBrewer palette name; can be NULL (see details)
`title`	the title for the plot; can be NULL
`return.matrix`	if TRUE, return a list containing the data matrix and the heatmap instead of the heatmap alone
`na.to`	convert NA values in the matrix to this value immediately before computing the heatmap
`...`	additional parameters forwarded to Heatmap

Details

This is just a convenience function which

1. Calls GetTable with the parameter `type,columns,summarize,genes`
2. Transforms the returned table using the `transform` parameter
3. Determines reasonable colors using `breaks` and `colors`
4. and then calls ComplexHeatmap::Heatmap

`type` and `columns` can refer to values from data slots values from analyses (and can be mixed). If there are types from both data and analyses, columns must be NULL. Otherwise columns must either be condition/cell names (if type refers to one or several data slots), or regular expressions to match against the names in the analysis tables.

Columns definitions for data slots can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

To refer to data slots, the mode.slot syntax can be used: Each name is either a data slot, or one of (new,old,total) followed by a dot followed by a slot. For new or old, the data slot value is multiplied by ntr or 1-ntr. This can be used e.g. to obtain the *new counts*.

The transform parameter either is a function that transforms a matrix (which can conveniently be done using the Transform.XXX functions described next), or a character (which must be the XXX to find such a function). Available data transformations are

- transform=Transform.Z() or transform="Z": compute z scores for each row (see Transform.Z)
- transform=Transform.VST() or transform="VST": do a variance stabilizing transformation (see Transform.VST)
- transform=Transform.logFC() or transform="logFC": compute log2 fold changes to one or several reference columns; which must be defined via parameters (see Transform.logFC)
- transform=Transform.no() or transform="no": do not transform (see Transform.no)

Reasonable coloring is chosen depending on the value distribution in the matrix. If the values are zero centered (e.g. z scores or most often log fold changes), then by default the 50 quantile with the larger value. The breaks are -q90,q50,0,q50,q90, and, by default, the red to blue "RdBu" palette from RColorBrewer is taken. If the values are not zero centered, the 5

xlab can be given as a character vector or an expression that evaluates into a character vector. The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables.

Value

a ComplexHeatmap object

See Also

GetTable,Heatmap

`PlotMismatchPositionForSample`

Diagnostic plot for mismatch position for columns (by sample)

Description

This belongs to the first diagnostic plots (raw mismatches) generated by GRAND3.

Usage

```
PlotMismatchPositionForSample(
  data,
  sample,
  orientation = NULL,
  category = NULL
)
```

Arguments

`data`	a grandR object
`sample`	a sample name
`orientation`	restrict to either Sense or Antisense; can be NULL
`category`	restrict to a specific category (see GetDiagnosticParameters); can be NULL

Details

For all positions along the reads (x axis; potentially paired end, shown left and right), show the percentage of all mismatch types. The panel in column T and row C shows T-to-C mismatches. Positions outside of shaded areas are clipped. Uncorrected and Retained means before and after correcting multiply sequenced bases. Sense/Antisense means reads (first read for paired end) that are (based on the annotation) oriented in sense or antisense direction to a gene (i.e. this is only relevant for sequencing protocols that do not preserve strand information).

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotMismatchPositionForType`

Diagnostic plot for mismatch position for columns (by mismatch type)

Description

This belongs to the first diagnostic plots (raw mismatches) generated by GRAND3.

Usage

```
PlotMismatchPositionForType(
  data,
  genomic,
  read,
  orientation = NULL,
  category = NULL
)
```

Arguments

`data`	a grandR object
`genomic`	the nucleotide as it occurs in the genome
`read`	the nucleotide as it occurs in the read
`orientation`	restrict to either Sense or Antisense; can be NULL
`category`	restrict to a specific category (see GetDiagnosticParameters); can be NULL

Details

For all positions along the reads (x axis; potentially paired end, shown left and right), show the percentage of a specific mismatch type for all samples. Positions outside of shaded areas are clipped. Uncorrected and Retained means before and after correcting multiply sequenced bases. Sense/Antisense means reads (first read for paired end) that are (based on the annotation) oriented in sense or antisense direction to a gene (i.e. this is only relevant for sequencing protocols that do not preserve strand information).

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotModelCompareConv` *Diagnostic plot for estimated models (global conversion rate)*

Description

This belongs to the fourth kind (model comparison) of diagnostic plots

Usage

```
PlotModelCompareConv(data, label = "4sU", estimator = "Separate")
```

Arguments

`data`	a grandR object
`label`	which label to consider (see GetDiagnosticParameters); cannot be NULL
`estimator`	which estimator to consider (see GetDiagnosticParameters); cannot be NULL

Details

Compares the estimated conversion rate (i.e., the probability for a conversion on a new RNA molecule) for the binom and tbbinom models (mean conversion rate).

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotModelCompareErr` *Diagnostic plot for estimated models (global error rate)*

Description

This belongs to the fourth kind (model comparison) of diagnostic plots

Usage

```
PlotModelCompareErr(data, label = "4sU", estimator = "Separate")
```

Arguments

`data`	a grandR object
`label`	which label to consider (see GetDiagnosticParameters); cannot be NULL
`estimator`	which estimator to consider (see GetDiagnosticParameters); cannot be NULL

Details

Compares the estimated error rate (i.e., the probability for a conversion on an old RNA molecule) for the binom and tbbinom models.

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotModelCompareErrPrior`

Diagnostic plot for estimated models (global error rate)

Description

This belongs to the fourth kind (model comparison) of diagnostic plots

Usage

```
PlotModelCompareErrPrior(
  data,
  label = "4sU",
  estimator = "Separate",
  model = "Binom"
)
```

Arguments

`data`	a grandR object
`label`	which label to consider (see GetDiagnosticParameters); cannot be NULL
`estimator`	which estimator to consider (see GetDiagnosticParameters); cannot be NULL
`model`	which model to consider (see GetDiagnosticParameters); cannot be NULL

Details

Compares the prior error rate (estimated from no4sU samples or from all other mismatch types) against the final error rate estimate.

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotModelCompareLL` *Diagnostic plot for estimated models (log likelihoods)*

Description

This belongs to the fourth kind (model comparison) of diagnostic plots

Usage

```
PlotModelCompareLL(data, label = "4sU", estimator = "Separate")
```

Arguments

`data`	a grandR object
`label`	which label to consider (see GetDiagnosticParameters); cannot be NULL
`estimator`	which estimator to consider (see GetDiagnosticParameters); cannot be NULL

Details

Shows the difference in log likelihoods between the binom and tbbinom models.

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotModelCompareNtr` *Diagnostic plot for estimated models (global NTR)*

Description

This belongs to the fourth kind (model comparison) of diagnostic plots

Usage

```
PlotModelCompareNtr(data, label = "4sU", estimator = "Separate")
```

Arguments

`data`	a grandR object
`label`	which label to consider (see GetDiagnosticParameters); cannot be NULL
`estimator`	which estimator to consider (see GetDiagnosticParameters); cannot be NULL

Details

Compares the global NTR (i.e. for all reads used for estimation of global parameters, what is the percentage of new RNA) for the binom and tbbinom models.

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotModelConv` *Diagnostic plot for estimated models (global conversion rate)*

Description

This belongs to the third kind (model) of diagnostic plots

Usage

```
PlotModelConv(data, label = "4sU", estimator = "Separate", model = "Binom")
```

Arguments

`data`	a grandR object
`label`	which label to consider (see GetDiagnosticParameters); cannot be NULL
`estimator`	which estimator to consider (see GetDiagnosticParameters); cannot be NULL
`model`	which model to consider (see GetDiagnosticParameters); cannot be NULL

Details

Shows the estimated conversion rate (i.e., the probability for a conversion on a new RNA molecule) for each sample.

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotModelErr` *Diagnostic plot for estimated models (global error rate)*

Description

This belongs to the third kind (model) of diagnostic plots

Usage

```
PlotModelErr(data, label = "4sU", estimator = "Separate", model = "Binom")
```

Arguments

`data`	a grandR object
`label`	which label to consider (see GetDiagnosticParameters); cannot be NULL
`estimator`	which estimator to consider (see GetDiagnosticParameters); cannot be NULL
`model`	which model to consider (see GetDiagnosticParameters); cannot be NULL

Details

Shows the estimated error rate (i.e., the probability for a conversion on an old RNA molecule) for each sample.

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotModelLabelTimeCourse`

Diagnostic plot for estimated models (4sU increase)

Description

This belongs to the third kind (model) of diagnostic plots

Usage

```
PlotModelLabelTimeCourse(data, label = "4sU", estimator = "Separate")
```

Arguments

`data`	a grandR object
`label`	which label to consider (see GetDiagnosticParameters); cannot be NULL
`estimator`	which estimator to consider (see GetDiagnosticParameters); cannot be NULL

Details

Shows the estimated time evolution of 4sU increase in the tbbinom model for each sample.

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotModelNtr` *Diagnostic plot for estimated models (global NTR)*

Description

This belongs to the third kind (model) of diagnostic plots

Usage

```
PlotModelNtr(data, label = "4sU", estimator = "Separate", model = "Binom")
```

Arguments

`data`	a grandR object
`label`	which label to consider (see GetDiagnosticParameters); cannot be NULL
`estimator`	which estimator to consider (see GetDiagnosticParameters); cannot be NULL
`model`	which model to consider (see GetDiagnosticParameters); cannot be NULL

Details

Shows the estimated global NTR (i.e. for all reads used for estimation of global paramters, what is the percentage of new RNA) for each sample.

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotModelShape` *Diagnostic plot for estimated models (global shape parameter)*

Description

This belongs to the third kind (model) of diagnostic plots

Usage

```
PlotModelShape(data, label = "4sU", estimator = "Separate")
```

Arguments

`data`	a grandR object
`label`	which label to consider (see GetDiagnosticParameters); cannot be NULL
`estimator`	which estimator to consider (see GetDiagnosticParameters); cannot be NULL

Details

Shows the estimated shape parameter (describing the increase of 4sU over time) in the tbbinom model for each sample.

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`PlotPCA` *Make a PCA plot*

Description

Make a PCA plot

Usage

```
PlotPCA(
  data,
  mode.slot = DefaultSlot(data),
  ntop = 500,
  aest = NULL,
  x = 1,
  y = 2,
  columns = NULL,
  do.vst = TRUE
)
```

Arguments

`data`	the grandR object that contains the data to plot
`mode.slot`	the mode and slot of data to plot; slot in the grandr object (eg "count")
`ntop`	how many genes to use
`aest`	parameter to set the visual attributes
`x`	number of principal component to show on the x axis (numeric)
`y`	number of principal component to show on the y axis (numeric)
`columns`	which columns (i.e. samples or cells) to perform PCA on (see details)
`do.vst`	perform a variance stabilizing transformation for count data?

Details

Columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells). The expression is evaluated in an environment having the `Coldata`, i.e. you can use names of `Coldata` as variables to conveniently build a logical vector (e.g., columns=Condition=="x").

Value

a PCA plot

`PlotProfileLikelihood` *Diagnostic plot for estimated models (global error rate)*

Description

This belongs to the fifth kind (profile likelihoods) of diagnostic plots

Usage

```
PlotProfileLikelihood(
  data,
  label = "4sU",
  estimator = NULL,
  sample = NULL,
  subread = NULL
)
```

Arguments

`data`	a grandR object
`label`	which label to consider (see GetDiagnosticParameters); cannot be NULL
`estimator`	which estimator to consider (see GetDiagnosticParameters); cannot be NULL
`sample`	which sample to consider (see GetDiagnosticParameters); cannot be NULL
`subread`	which subread to consider (see GetDiagnosticParameters); cannot be NULL

Details

Shows the profile likelihoods for all parameters of the tbbinom model.

Value

a list with a ggplot object, a description, and the desired size for the plot

---

`Plots` *Stored plot functions*

Description

Get plot names and add or remove plots

Usage

```
Plots(data)

AddGenePlot(data, name, FUN)

AddGlobalPlot(data, name, FUN, floating = FALSE)

PlotGene(data, name, gene)

PlotGlobal(data, name)

DropPlots(data, pattern = NULL)
```

Arguments

`data`	A grandR object
`name`	The user-defined plot name
`FUN`	The plotting function to add
`floating`	whether or not the plot should be shown as a floating window
`gene`	The gene to plot
`pattern`	A regular expression that is matched to plot names

Details

FUN has to be a function with a single parameter for global plots (i.e., the grandR object) or two parameters for gene plots (i.e., the grandR object and the gene name). Usually, it is either the name of a plotting function, such as PlotGeneOldVsNew, or, if it is necessary to parametrize it, a call to Defer (which takes care of caching plots without storing an additional copy of the grandR object).

For floating window plots, if names are given in the format <title>.<name>, a plot is created for each <title> with several subplots.

Value

Either the plot names or a grandR data with added/removed plots

Functions

- `Plots()`: Obtain the plot names
- `AddGenePlot()`: Add a gene plot to the grandR object
- `AddGlobalPlot()`: Add a global plot to the grandR object
- `PlotGene()`: Create a gene plot
- `PlotGlobal()`: Create a global plot
- `DropPlots()`: Remove plots from the grandR object

`PlotScatter` *Make a scatter plot*

Description

Convenience method to compare two variables (slot data or analyses results).

Usage

```
PlotScatter(
  data,
  x = NULL,
  y = NULL,
  analysis = NULL,
  xcol = NULL,
  ycol = NULL,
  xlab = NULL,
  ylab = NULL,
  log = FALSE,
  log.x = log,
  log.y = log,
  axis = TRUE,
  axis.x = axis,
  axis.y = axis,
  remove.outlier = 1.5,
  show.outlier = TRUE,
  lim = NULL,
  xlim = lim,
  ylim = lim,
  size = 0.3,
  cross = NULL,
  diag = NULL,
  filter = NULL,
  genes = NULL,
  highlight = NULL,
  label = NULL,
  label.repel = 1,
```

```
  facet = NULL,
  color = NULL,
  colorpalette = NULL,
  color.label = NULL,
  density.margin = "n",
  density.n = 100,
  rasterize = NULL,
  correlation = NULL,
  correlation.x = -Inf,
  correlation.y = Inf,
  correlation.hjust = 0.5,
  correlation.vjust = 0.5,
  layers.below = NULL
)
```

Arguments

`data`	the grandR object (can also be a plain data frame)
`x`	an expression to compute the x value or a character corresponding to a sample (or cell) name or a fully qualified analysis result name (see details)
`y`	an expression to compute the y value or a character corresponding to a sample (or cell) name or a fully qualified analysis result name (see details)
`analysis`	the name of an analysis table (can be NULL; see details)
`xcol`	a character corresponding to a sample (or cell) name or a fully qualified analysis result name (see details)
`ycol`	a character corresponding to a sample (or cell) name or a fully qualified analysis result name (see details)
`xlab`	the label for x (can be NULL, then the x parameter is used)
`ylab`	the label for y (can be NULL, then the y parameter is used)
`log`	if TRUE, use log scales for x and y axis
`log.x`	if TRUE, use log scale for the x axis
`log.y`	if TRUE, use log scale for the y axis
`axis`	if FALSE, don't show x and y axes
`axis.x`	if FALSE, don't show the x axis
`axis.y`	if FALSE, don't show the y axis
`remove.outlier`	configure how outliers are selected (is the coef parameter to boxplot.stats); can be FALSE, in which case no points are considered outliers (see details)
`show.outlier`	if TRUE, show outlier as gray points at the border of the plotting plane
`lim`	define the both x and y axis limits (vector of length 2 defining the lower and upper bound, respectively)
`xlim`	define the x axis limits (vector of length 2 defining the lower and upper bound, respectively)
`ylim`	define the y axis limits (vector of length 2 defining the lower and upper bound, respectively)

`size`	the point size to use
`cross`	add horizontal and vertical lines through the origin?
`diag`	if TRUE, add main diagonal; if numeric vector, add these diagonals
`filter`	restrict to these rows; is evaluated for the data frame, and should result in a logical vector
`genes`	restrict to these genes; can be either numeric indices, gene names, gene symbols or a logical vector
`highlight`	highlight these genes; can be either numeric indices, gene names, gene symbols or a logical vector (see details)
`label`	label these genes; can be either numeric indices, gene names, gene symbols or a logical vector (see details)
`label.repel`	force to repel labels from points and each other (increase if labels overlap)
`facet`	an expression (evaluated in the same environment as x and y); for each unique value a panel (facet) is created; can be NULL
`color`	either NULL (use point density colors), or a name of the GeneInfo table (use scale_color_xxx to define colors), or a color for all points
`colorpalette`	either NULL (use default colors), or a palette name from color brewer or viridis
`color.label`	the label for the color legend
`density.margin`	for density colors, one of 'n','x' or 'y'; should the density be computed along both axes ('n'), or along 'x' or 'y' axis only
`density.n`	how many bins to use for density calculation (see kde2d)
`rasterize`	use ggrastr to rasterize points? (can be NULL, see details)
`correlation`	a function to format correlation statistics to be annotated (see details)
`correlation.x`	x coordinate to put the correlation annotation in the plot (see details)
`correlation.y`	y coordinate to put the correlation annotation in the plot (see details)
`correlation.hjust`	x adjustment to put the correlation annotation in the plot (see details)
`correlation.vjust`	y adjustment to put the correlation annotation in the plot (see details)
`layers.below`	list of ggplot geoms to add before adding the layer containing the points

Details

Both the x and y parameter are either expressions or names. Names are either sample (or cell, in case of single cell experiments) names or fully qualified analysis results (analysis name followed by a dot and the analysis result table column). If the analysis parameter is given, the analysis name must be omitted from x and y. These names can be used within expressions using non-standard evaluation. Defining by names only works with character literals like "kinetics.Synthesis", but if you give an expression (e.g. a variable name that contains a character), the situation is more complicated, since PlotScatter will try to evaluate this for defining the values, not the name of the column. If the expression evaluates into a single character string that is equal to a name (see above!), PlotScatter knows what to do. For more complicated situations that cannot be resolved by this, you can use the xcol and ycol parameters instead of the x and y parameters!

By default the limits of x and y axis are chosen after removing outliers (using the same algorithm used for boxplot). Thus, larger numbers filter less stringently. remove.outlier can also be set to FALSE (no outlier filtering). If xlim or ylim are set, this overrides outlier filtering. Points outside of the limits (i.e. outliers or points outside of xlim or ylim) are set to infinity (such that they are shown at the border of the plot in gray)

By default, all genes are shown. This can be restricted using the `genes` parameter (see ToIndex). It is also possible to highlight a subset of the genes using `highlight`. This parameter either describes a subset of the genes (either numeric indices, gene names, gene symbols or a logical vector), in which case these genes are plotted in red and with larger points size, or it can be a list of such vectors. The names of this list must be valid colors. Genes can also be labeled (make sure that this is really only a small subset of the genes).

When rendering to vector based devices (such as svg or pds), a genome-wide scatterplot often is painfully big (and rendering therefore slow). The `rasterize` parameter can be used to automatically rasterize the points only (via the ggrastr package). If this parameter is NULL, ggrastr is used if more than 1000 points are plotted!

Often scatter plots show that x and y coordinates are correlated. Correlations can be annotated using the FormatCorrelation function. Most often you will use `PlotScatter(data,x,y,correlation=FormatCorrelation())`. To use a different correlation measure, other formats for correlation coefficient and P values or omit one of these statistics, parametrize `FormatCorrelation`. Use correlation.x and correlation.y to place the annotation in the plot, and correlation.hjust/correlation.vjust to align the annotation at the given x,y coordinates. Infinite values for correlation.x/correlation.y will put the annotation at the border of the plot.

Value

a ggplot object with the data frame used as the df attribute

`PlotSimulation` *Plot simulated data*

Description

The input data is usually created by `SimulateKinetics`

Usage

```
PlotSimulation(
  sim.df,
  ntr = TRUE,
  old = TRUE,
  new = TRUE,
  total = TRUE,
  line.size = 1
)
```

Arguments

`sim.df`	the input data frame
`ntr`	show the ntr?
`old`	show old RNA?
`new`	show new RNA?
`total`	show total RNA?
`line.size`	which line size to use

Value

a ggplot object

See Also

SimulateKinetics for creating the input data frame

Examples

```
PlotSimulation(SimulateKinetics(hl=2))
```

`PlotTypeDistribution` *Plot the distribution of gene types*

Description

Plot the distribution of gene types

Usage

```
PlotTypeDistribution(data, mode.slot = DefaultSlot(data), relative = FALSE)
```

Arguments

`data`	the grandR object to get the data to be plotted from
`mode.slot`	which mode and slot to use
`relative`	show percentage values?

Value

a ggplot object

`PoolColumns` *Pool reads across columns*

Description

Pool read counts, ntrs, and alpha/beta values across columns defined by a pooling matrix

Usage

```
PoolColumns(
  data,
  pooling = GetSummarizeMatrix(data, average = FALSE, no4sU = TRUE)
)
```

Arguments

`data`	grandR object
`pooling`	a pooling matrix (see details)

Details

The pooling matrix must have as many rows as there are columns (i.e., samples or cells) in data, and as many columns as you want to have columns in the resulting object. The matrix should consist of 0 and 1, where 1 indicates a column of the original object that should go into a column of the new object. In essence, to obtain the new count matrix, the old count matrix is matrix-multiplied with the pooling matrix.

The new ntr matrix is computed by componentwise division of the new count and total count matrices derived as just described. alpha and beta are computed using matrix multiplication, i.e. summing up all alpha and beta values of all the columns belonging to a pool.

Value

a new grandR object

`psapply` *Parallel (s/l)apply*

Description

Depending on whether SetParallel has been called, execute in parallel or not.

Usage

```
psapply(..., seed = NULL)

plapply(..., seed = NULL)
```

Arguments

`...`	forwarded to lapply or parallel::mclapply
`seed`	Seed for the random number generator

Details

If the code uses random number specify the seed to make it deterministic

Value

a vector (psapply) or list (plapply)

`ReadCounts` *Read a count table*

Description

grandR can also be used to analyze standard RNA-seq data, and this function is here to read such data.

Usage

```
ReadCounts(
  file,
  design = c(Design$Condition, Design$Replicate),
  classify.genes = ClassifyGenes(),
  rename.sample = NULL,
  filter.table = NULL,
  num.samples = NULL,
  verbose = FALSE,
  sep = "\t"
)
```

Arguments

`file`	a file containing a count matrix
`design`	Either a design vector (see details), or a data.frame providing metadata for all columns (samples/cells), or a function that is called with the condition name vector and is supposed to return this data.frame.
`classify.genes`	A function that is used to add the *type* column to the gene annotation table, always a call to ClassifyGenes
`rename.sample`	function that is applied to each sample name before parsing (or NULL)
`filter.table`	function that is applied to the table directly after read it (or NULL)
`num.samples`	number of sample columns containing read counts (can be NULL, see details)
`verbose`	Print status updates
`sep`	The column separator used in the file

Details

The table is assumed to have read counts in the last n columns, which must be named according to sample names. If num.samples is NULL this n is automatically recognized as the number of numeric columns (so make sure to either specify num.samples, or that the column immediately prior to the first sample column is *not* numeric).

If these columns are named systematically in a particular way, the design vector provides a powerful and easy way to create the column annotations.

The column names have to contain dots (.) to separate the fields for the column annotation table. E.g. the name *Mock.4h.A* will be split into the fields *Mock*, *4h* and *A*. For such names, a design vector of length 3 has to be given, that describes the meaning of each field. A reasonable design vector for the example would be `c("Treatment","Time","Replicate")`. Some names are predefined in the list Design.

The names given in the design vector might even have additional semantics: E.g. for the name *duration.4sU* the values are interpreted (e.g. 4h is converted into the number 4, or 30min into 0.5, or no4sU into 0). Semantics can be user-defined by calling `MakeColdata` and using the return value as the design parameter, or a function that calls MakeColdata. In most cases it is easier to manipulate the `Coldata` table after loading data instead of using this mechanism; the build-in semantics simply provide a convenient way to reduce this kind of manipulation in most cases.

Sometimes you might have forgotten to name all samples consistently (or you simply messed something up). In this case, the rename.sample parameter can be handy (e.g. to rename a particular misnamed sample).

Sometimes the table contains more than you want to read. In this case, use the filter.table parameter to preprocess it. This should be a function that receives a data.frame, and returns a data.frame.

If there are no columns named "Gene" or "Symbol", the first column is used!

Value

a grandR object

`ReadFeatureCounts` *Read featureCounts*

Description

grandR can also be used to analyze standard RNA-seq data, and this function is here to read such data.

Usage

```
ReadFeatureCounts(
  file,
  design = c(Design$Condition, Design$Replicate),
  classify.genes = ClassifyGenes(),
  rename.sample = NULL,
  filter.table = NULL,
```

```
  num.samples = NULL,
  verbose = FALSE,
  sep = "\t"
)
```

Arguments

`file`	a file containing featureCounts
`design`	Either a design vector (see details), or a data.frame providing metadata for all columns (samples/cells), or a function that is called with the condition name vector and is supposed to return this data.frame.
`classify.genes`	A function that is used to add the *type* column to the gene annotation table, always a call to ClassifyGenes
`rename.sample`	function that is applied to each sample name before parsing (or NULL)
`filter.table`	function that is applied to the table directly after read it (or NULL)
`num.samples`	number of sample columns containing read counts (can be NULL, see details)
`verbose`	Print status updates
`sep`	The column separator used in the file

Details

The table is assumed to have read counts in the last n columns, which must be named according to sample names. If num.samples is NULL this n is automatically recognized as the number of columns containing .bam (so make sure to either specify num.samples, or that the count columns are called after the bam files).

If these columns are named systematically in a particular way, the design vector provides a powerful and easy way to create the column annotations.

The column names have to contain dots (.) to separate the fields for the column annotation table. E.g. the name *Mock.4h.A* will be split into the fields *Mock*, *4h* and *A*. For such names, a design vector of length 3 has to be given, that describes the meaning of each field. A reasonable design vector for the example would be `c("Treatment","Time","Replicate")`. Some names are predefined in the list Design.

The names given in the design vector might even have additional semantics: E.g. for the name *duration.4sU* the values are interpreted (e.g. 4h is converted into the number 4, or 30min into 0.5, or no4sU into 0). Semantics can be user-defined by calling `MakeColdata` and using the return value as the design parameter, or a function that calls MakeColdata. In most cases it is easier to manipulate the `Coldata` table after loading data instead of using this mechanism; the build-in semantics simply provide a convenient way to reduce this kind of manipulation in most cases.

Sometimes you might have forgotten to name all samples consistently (or you simply messed something up). In this case, the rename.sample parameter can be handy (e.g. to rename a particular misnamed sample).

Sometimes the table contains more than you want to read. In this case, use the filter.table parameter to preprocess it. This should be a function that receives a data.frame, and returns a data.frame.

If there are no columns named "Geneid", "Gene" or "Symbol", the first column is used!

Value

a grandR object

ReadGRAND *Read the output of GRAND-SLAM 2.0 into a grandR object.*

Description

Metabolic labeling - nucleotide conversion RNA-seq data (such as generated by SLAM-seq,TimeLapse-seq or TUC-seq) must be carefully analyzed to remove bias due to incomplete labeling. GRAND-SLAM is a software package that employs a binomial mixture modeling approach to obtain precise estimates of the new-to-total RNA ratio (NTR) per gene and sample (or cell). This function directly reads the output of GRAND-SLAM 2.0 into a grandR object.

Usage

```
ReadGRAND(
  prefix,
  design = c(Design$Condition, Design$Replicate),
  classify.genes = ClassifyGenes(),
  read.percent.conv = FALSE,
  read.min2 = FALSE,
  rename.sample = NULL,
  verbose = FALSE
)
```

Arguments

`prefix`	Can either be the prefix used to call GRAND-SLAM with, or the main output file ($prefix.tsv.gz); if the RCurl package is installed, this can also be a URL
`design`	Either a design vector (see details), or a data.frame providing metadata for all columns (samples/cells), or a function that is called with the condition name vector and is supposed to return this data.frame.
`classify.genes`	A function that is used to add the *type* column to the gene annotation table, always a call to ClassifyGenes
`read.percent.conv`	Should the percentage of conversions also be read?
`read.min2`	Should the read count with at least 2 mismatches also be read?
`rename.sample`	function that is applied to each sample name before parsing (or NULL)
`verbose`	Print status updates

Details

If columns (samples/cells) are named systematically in a particular way, the design vector provides a powerful and easy way to create the column annotations.

The column names have to contain dots (.) to separate the fields for the column annotation table. E.g. the name *Mock.4h.A* will be split into the fields *Mock*, *4h* and *A*. For such names, a design vector of length 3 has to be given, that describes the meaning of each field. A reasonable design vector for the example would be `c("Treatment","Time","Replicate")`. Some names are predefined in the list Design.

The names given in the design vector might even have additional semantics: E.g. for the name *duration.4sU* the values are interpreted (e.g. 4h is converted into the number 4, or 30min into 0.5, or no4sU into 0). Semantics can be user-defined by calling `MakeColdata` and using the return value as the design parameter, or a function that calls MakeColdata. In most cases it is easier to manipulate the `Coldata` table after loading data instead of using this mechanism; the build-in semantics simply provide a convenient way to reduce this kind of manipulation in most cases.

Sometimes you might have forgotten to name all samples consistently (or you simply messed something up). In this case, the rename.sample parameter can be handy (e.g. to rename a particular misnamed sample).

Value

A grandR object containing the read counts, NTRs, information on the NTR posterior distribution (alpha,beta) and potentially additional information of all genes detected by GRAND-SLAM

See Also

ReadGRAND3,ClassifyGenes,MakeColdata,DesignSemantics

Examples

```
sars <- ReadGRAND("https://zenodo.org/record/5834034/files/sars.tsv.gz",
                      design=c("Cell",Design$dur.4sU,Design$Replicate), verbose=TRUE)
```

`ReadGRAND3` *Read the output of GRAND-SLAM 3.0 into a grandR object.*

Description

Metabolic labeling - nucleotide conversion RNA-seq data (such as generated by SLAM-seq,TimeLapse-seq or TUC-seq) must be carefully analyzed to remove bias due to incomplete labeling. GRAND-SLAM is a software package that employs a binomial mixture modeling approach to obtain precise estimates of the new-to-total RNA ratio (NTR) per gene and sample (or cell). This function directly reads the output of GRAND-SLAM 3.0 into a grandR object.

Usage

```
ReadGRAND3(
  prefix,
  design = NULL,
  label = "4sU",
  estimator = "Binom",
  classify.genes = ClassifyGenes(),
  read.posterior = NULL,
  rename.sample = NULL,
  verbose = FALSE
)
```

Arguments

`prefix`	the prefix used to call GRAND-SLAM
`design`	Either a design vector (see details), or a data.frame providing metadata for all columns (samples/cells), or a function that is called with the condition name vector and is supposed to return this data.frame. if NULL, a library,sample,barcode design is used for sparse data, and a condition,replicate design for dense data
`label`	which nucleoside analog
`estimator`	which estimator to use (one of Binom,TbBinom,TbBinomShape)
`classify.genes`	A function that is used to add the *type* column to the gene annotation table, always a call to ClassifyGenes
`read.posterior`	also read the posterior parameters alpha and beta? if NULL, TRUE for dense data, FALSE for sparse data
`rename.sample`	function that is applied to each sample name before parsing (or NULL)
`verbose`	Print status updates

Details

If columns (samples/cells) are named systematically in a particular way, the design vector provides a powerful and easy way to create the column annotations.

The column names have to contain dots (.) to separate the fields for the column annotation table. E.g. the name *Mock.4h.A* will be split into the fields *Mock*, *4h* and *A*. For such names, a design vector of length 3 has to be given, that describes the meaning of each field. A reasonable design vector for the example would be `c("Treatment","Time","Replicate")`. Some names are predefined in the list Design.

The names given in the design vector might even have additional semantics: E.g. for the name *duration.4sU* the values are interpreted (e.g. 4h is converted into the number 4, or 30min into 0.5, or no4sU into 0). Semantics can be user-defined by calling `MakeColdata` and using the return value as the design parameter, or a function that calls MakeColdata. In most cases it is easier to manipulate the `Coldata` table after loading data instead of using this mechanism; the build-in semantics simply provide a convenient way to reduce this kind of manipulation in most cases.

Sometimes you might have forgotten to name all samples consistently (or you simply messed something up). In this case, the rename.sample parameter can be handy (e.g. to rename a particular misnamed sample).

Value

A grandR object containing the read counts, NTRs, information on the NTR posterior distribution (alpha,beta) and potentially additional information of all genes detected by GRAND-SLAM

See Also

ReadGRAND,ClassifyGenes,MakeColdata,DesignSemantics

`ReadNewTotal` *Read sparse new/total matrices*

Description

This function can be used to load matrix market data in case genes were quantified by (i) counting all reads (for total RNA) and (ii) counting T-to-C mismatch reads (for new RNA)

Usage

```
ReadNewTotal(
  genes,
  cells,
  new.matrix,
  total.matrix,
  detection.rate = 1,
  verbose = FALSE
)
```

Arguments

`genes`	csv file (or URL) containing gene information
`cells`	csv file (or URL) containing cell information
`new.matrix`	Matrix market file of new counts
`total.matrix`	Matrix market file of total counts
`detection.rate`	the detection rate of T-to-C mismatch reads (see details)
`verbose`	verbose output

Details

Metabolic labeling - nucleotide conversion RNA-seq data (such as generated by SLAM-seq,TimeLapse-seq or TUC-seq) must be carefully analyzed to remove bias due to incomplete labeling. We advice against counting read with and without T-to-C mismatches for quantification, and encourage using a statistical method such as GRAND-SLAM that properly deals with incomplete labeling.

To correct for some bias, a detection rate (as suggested by Cao et al., Nature Biotech 2020) should be provided. This detection rate defines, how much new RNA is detected on average using the T-to-C mismatch reads.

Value

a grandR object

`RotatateAxisLabels` *Rotate x axis labels*

Description

Add this to a ggplot object to rotate the x axis labels

Usage

```
RotatateAxisLabels(angle = 90)
```

Arguments

`angle`	the angle by which to rotate

Value

a ggplot theme object

`SaveNtrSlot` *Copy the NTR slot and save under new name*

Description

Copy the NTR slot and save under new name

Usage

```
SaveNtrSlot(data, name)
```

Arguments

`data`	the grandR object
`name`	the name of the new slot

Value

a grandR object

`Scale` *Scale data*

Description

Compute values for all genes standardized (i.e. z scores) across samples.

Usage

```
Scale(
  data,
  name = "scaled",
  slot = DefaultSlot(data),
  set.to.default = FALSE,
  group = NULL,
  center = TRUE,
  scale = TRUE
)
```

Arguments

`data`	a grandR object
`name`	the new slot name
`slot`	the slot from where to take values
`set.to.default`	set the new slot as default slot
`group`	Perform standardization per group of columns (see details)
`center`	Perform centering (forwarded to scale)
`scale`	Perform scaling (forwarded to scale)

Details

Standardization can be done per group. For this, the group parameter has to be a name of the `Coldata` table, to define groups of columns (i.e. samples or cells).

Value

a new grandR object with a new slot

See Also

scale

`Semantics.concentration` *Semantics for concentration columns*

Description

Defines additional semantics for columns representing concentrations

Usage

```
Semantics.concentration(s, name)
```

Arguments

`s`	original column
`name`	the column name

Value

a data frame with a single numeric column, where <x>uM from s is replaced by x, <x>mM is replaced by x*1000, and no4sU is replaced by 0

`Semantics.time` *Semantics for time columns*

Description

Defines additional semantics for columns representing temporal dimensions

Usage

```
Semantics.time(s, name)
```

Arguments

`s`	original column
`name`	the column name

Value

a data frame with a single numeric column, where <x>h from s is replaced by x, <x>min is replaced by x/60, and no4sU is replaced by 0

`ServeGrandR` *Serve a shiny web interface*

Description

Fire up a shiny web server for exploratory analysis of grandR data.

Usage

```
ServeGrandR(
  data,
  table = NULL,
  sizes = NA,
  height = 400,
  plot.gene = NULL,
  plot.global = NULL,
  plot.window = NULL,
  highlight = NULL,
  df.identifier = "Symbol",
  title = Title(data),
  show.sessionInfo = FALSE,
 help = list(".Q: multiple testing corrected p values", ".LFC: log2 fold changes")
)
```

Arguments

`data`	the grandR object (or a file name to an rds file containing a grandR object)
`table`	the table to display (can be NULL or a named list; see details)
`sizes`	the widths for the gene plots to show (12 is full screen with); must be a vector as long as there are gene plots
`height`	the height for the gene plots in pixel
`plot.gene`	a list of gene plots; can be NULL, then the stored gene plots are used (see Plots)
`plot.global`	a list of global plots; can be NULL, then the stored global plots are used (see Plots)
`plot.window`	a list of static plots to show in a floating window; see details
`highlight`	a vector of gene names that are highlighted in the beginning
`df.identifier`	the main identifier (column name) from the table; this is used when calling the gene plot functions;
`title`	the title to show in the header of the website
`show.sessionInfo`	whether to show session info
`help`	a list of characters that is shown as help text at the beginning (when no gene plot is shown); should describe the contents of your table

Details

If the table parameter is NULL, either an analysis table named "ServeGrandR" is used (if it exists), otherwise the columns "Q", "LFC", "Synthesis" and "Half-life" of all analysis tables are used. If it is a list, a menu is created in the navbar

plot.window must be a list of plotting functions that receive the grandR object and return a plot. It can also be a list of list, in which case more than one plotting windows are generated. Each plot will be rendered with a size of 350x350.

The gene plots must be functions that accept two parameters: the grandR object and a gene identifier. You can either use functions directly (e.g. `plot.gene=list(PlotGeneOldVsNew)`), or use Defer in cases you need to specify additional parameters, e.g. `plot.gene=list(Defer(PlotGeneOldVsNew,log=FALSE))`. The global plots are functions accepting a single parameter (the grandR object). Here the use of Defer is encouraged due to its caching mechanism.

Value

a shiny web server

Examples

```
## Not run:
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition",Design$dur.4sU,Design$Replicate))
sars <- Normalize(sars)
sars <- Pairwise(sars,contrasts = GetContrasts(sars,contrast = c("Condition","SARS","Mock")))
sars <- AddGenePlot(sars,"timecourse",
              Defer(PlotGeneProgressiveTimecourse,steady.state=c(Mock=TRUE,SARS=FALSE)))
sars <- AddGlobalPlot(sars,"Vulcano",VulcanoPlot)
ServeGrandR(sars)


## End(Not run)
```

`SetParallel` *Set up parallel execution*

Description

Set the number of cores for parallel execution.

Usage

```
SetParallel(cores = max(1, parallel::detectCores() - 2))
```

Arguments

`cores`	number of cores

Details

Whenever psapply or plapply are used, they are executed in parallel.

Value

No return value, called for side effects

`SimulateKinetics` *Simulate the kinetics of old and new RNA for given parameters.*

Description

The standard mass action kinetics model of gene expression arises from the differential equation $df/dt = s - df(t)$, with s being the constant synthesis rate, d the constant degradation rate and $f0 = f(0)$ (the abundance at time 0). The RNA half-life is directly related to d via $HL = log(2)/d$. This model dictates the time evolution of old and new RNA abundance after metabolic labeling starting at time t=0. This function simulates data according to this model.

Usage

```
SimulateKinetics(
  s = 100 * d,
  d = log(2)/hl,
  hl = 2,
  f0 = NULL,
  min.time = -1,
  max.time = 10,
  N = 1000,
  name = NULL,
  out = c("Old", "New", "Total", "NTR")
)
```

Arguments

`s`	the synthesis rate (see details)
`d`	the degradation rate (see details)
`hl`	the RNA half-life
`f0`	the abundance at time t=0
`min.time`	the start time to simulate
`max.time`	the end time to simulate
`N`	how many time points from min.time to max.time to simuate
`name`	add a Name column to the resulting data frame
`out`	which values to put into the data frame

Details

Both rates can be either (i) a single number (constant rate), (ii) a data frame with names "offset", "factor" and "exponent" (for linear functions, see ComputeNonConstantParam) or (iii) a unary function time->rate. Functions

Value

a data frame containing the simulated values

See Also

PlotSimulation for plotting the simulation

Examples

```
head(SimulateKinetics(hl=2))  # simulate steady state kinetics for an RNA with half-life 2h
```

`SimulateReadsForSample` *Simulate metabolic labeling - nucleotide conversion RNA-seq data.*

Description

This function takes a vector of *true* relative abundances and NTRs, and then simulates (i) read counts per gene and (ii) 4sU incorporation and conversion events. Subsequently, it uses the same approach as implemented in the GRAND-SLAM 2.0 software (Juerges et al., Bioinformatics 2018) to estimate the NTR from these simulated data.

Usage

```
SimulateReadsForSample(
  num.reads = 2e+07,
 rel.abundance = setNames(rlnorm(10000, meanlog = 4.5, sdlog = 1), paste0("Gene",
    1:10000)),
  ntr = setNames(rbeta(10000, 1.5, 3), paste0("Gene", 1:10000)),
  dispersion = 0.05,
  beta.approx = FALSE,
  conversion.reads = FALSE,
  u.content = 0.25,
  u.content.sd = 0.05,
  read.length = 75,
  p.old = 1e-04,
  p.new = 0.04,
  p.new.fit = p.new,
  seed = NULL
)
```

Arguments

`num.reads`	the total amount of reads for simulation
`rel.abundance`	named (according to genes) vector of the true relative abundances. Is divided by its sum.
`ntr`	vector of true NTRs
`dispersion`	vector of dispersion parameters (should best be estimated by DESeq2)
`beta.approx`	should the beta approximation of the NTR posterior be computed?
`conversion.reads`	also output the number of reads with conversion
`u.content`	the relative frequency of uridines in the reads
`u.content.sd`	the standard deviation of the u content
`read.length`	the read length for simulation
`p.old`	the probability for a conversion in reads originating from old RNA
`p.new`	the probability for a conversion in reads originating from new RNA
`p.new.fit`	the probability for a conversion in reads originating from new RNA that is used for fitting (to simulate bias in the estimation of p.new)
`seed`	seed value for the random number generator (set to make it deterministic!)

Details

The simulation proceeds as follows:

1. Draw for each gene the number of reads from a negative binomial distribution parametrized with the relative abundances x read number and the dispersion parameter
2. For each gene: Draw for each read the number of uridines according to a beta binomial distribution for the given read length (the beta prior is parametrized to match the u.content and u.content.sd parameters)
3. For each read: Draw the number of conversions according to the binomial mixture model of GRAND-SLAM (parametrized with p_old, p_new, the gene specific NTR and the read specific number of uridines)
4. Estimate the NTR by using the GRAND-SLAM approach

Value

a matrix containing, per column, the simulated counts, the simulated NTRs, (potentially the shape parameters of the beta distribution approximation,) and the true relative frequencies and ntrs

See Also

SimulateTimeCourse

Examples

```
SimulateReadsForSample(num.reads = 10000,rel.abundance = rep(1,5),ntr=0.9)
SimulateReadsForSample(num.reads = 10000,rel.abundance = rep(1,5),ntr=0.9,seed=1337)
SimulateReadsForSample(num.reads = 10000,rel.abundance = rep(1,5),ntr=0.9,seed=1337)
# the second and third matrix should be equal, the first should be distinct
```

`SimulateTimeCourse` *Simulate a complete time course of metabolic labeling - nucleotide conversion RNA-seq data.*

Description

This function takes a vector of *true* synthesis rates and RNA half-lives, and then simulates data for multiple time points and replicates. Both synthesis rate and RNA half-lives are assumed to be constant, but the system might not be in steady-state.

Usage

```
SimulateTimeCourse(
  condition,
  gene.info,
  s,
  d,
  f0 = s/d,
  s.variation = 1,
  d.variation = 1,
  dispersion,
  num.reads = 1e+07,
  timepoints = c(0, 0, 0, 1, 1, 1, 2, 2, 2, 4, 4, 4),
  beta.approx = FALSE,
  conversion.reads = FALSE,
  verbose = TRUE,
  seed = NULL,
  ...
)
```

Arguments

`condition`	A user-defined condition name (which is placed into the `Coldata` of the final grandR object)
`gene.info`	either a data frame containing gene annotation or a vector of gene names
`s`	a vector of synthesis rates
`d`	a vector of degradation rates (to get a specific half-life HL, use d=log(2)/HL)
`f0`	the abundance at time t=0
`s.variation`	biological variability of s among all samples (see details)

`d.variation`	biological variability of d among all samples (see details)
`dispersion`	a vector of dispersion parameters (estimate from data using DESeq2, e.g. by the estimate.dispersion utility function)
`num.reads`	a vector representing the number of reads for each sample
`timepoints`	a vector representing the labeling duration (in h) for each sample
`beta.approx`	should the beta approximation of the NTR posterior be computed?
`conversion.reads`	also output the number of reads with conversion
`verbose`	Print status updates
`seed`	seed value for the random number generator (set to make it deterministic!)
`...`	provided to `SimulateReadsForSample`

Details

If *s.variation* or *d.variation* are > 1, then for each gene a random gaussian is added to s (or d) such that 90 of the gaussian is log2(s.variation).

Value

a grandR object containing the simulated data in its data slots and the true parameters in the gene annotation table

---

`SimulateTimeCourseNonConstant` *Simulate a complete time course of metabolic labeling - nucleotide conversion RNA-seq data.*

Description

This function takes a vector of *true* synthesis rates and RNA half-lives, and then simulates data for multiple time points and replicates. Both synthesis rate and RNA half-lives are assumed to be constant, but the system might not be in steady-state.

Usage

```
SimulateTimeCourseNonConstant(
  condition,
  gene.info,
  s,
  d,
  dispersion,
  num.reads = 1e+07,
  t = 2,
  replicates = 3,
  beta.approx = FALSE,
```

```
    conversion.reads = FALSE,
    verbose = TRUE,
    seed = NULL,
    ...
  )
```

Arguments

`condition`	A user-defined condition name (which is placed into the `Coldata` of the final grandR object)
`gene.info`	either a data frame containing gene annotation or a vector of gene names
`s`	a vector of synthesis rates (see details)
`d`	a vector of degradation rates (see details)
`dispersion`	a vector of dispersion parameters (estimate from data using DESeq2, e.g. by the estimate.dispersion utility function)
`num.reads`	a vector representing the number of reads for each sample
`t`	a single number denoting the time
`replicates`	a single number denoting the number of replicates
`beta.approx`	should the beta approximation of the NTR posterior be computed?
`conversion.reads`	also output the number of reads with conversion
`verbose`	Print status updates
`seed`	seed value for the random number generator (set to make it deterministic!)
`...`	provided to `SimulateReadsForSample`

Details

Both rates can be either (i) a single number (constant rate), (ii) a data frame with names "offset", "factor" and "exponent" (for linear functions, see ComputeNonConstantParam; only one row allowed) or (iii) a unary function time->rate. Functions

Value

a grandR object containing the simulated data in its data slots and the true parameters in the gene annotation table

See Also

SimulateTimeCourse

`Slots` *Slot functions*

Description

Get slot names and add or remove slots

Usage

```
Slots(data)

DropSlot(data, pattern = NULL)

AddSlot(data, name, matrix, set.to.default = FALSE, warn = TRUE)
```

Arguments

`data`	A grandR object
`pattern`	a regular expression matched against slot names
`name`	the slot name
`matrix`	the data matrix for the new slot
`set.to.default`	set the new slot as the default slot?
`warn`	issue a warning if the slot name already exists and is overwritten

Value

Either the slot names or a grandR data with added/removed slots

Functions

- `Slots()`: Obtain the slot names
- `DropSlot()`: Remove one or several slots from this grandR object
- `AddSlot()`: Add an additional slot to this grandR object

See Also

DefaultSlot

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))

sars <- Normalize(sars)     # default behavior is to update the default slot
sars
sars <- DropSlot(sars,"norm")
sars                        # note that the defauls slot reverted to count
```

`structure2vector` *Convert a structure into a vector*

Description

The structure is supposed to be a list. Flattening is done by extracting the given fields (`return.fields`) and applying the additional function (`return.extra`). This is mainly to be used within `sapply` and similar.

Usage

```
structure2vector(d, return.fields = NULL, return.extra = NULL)

kinetics2vector(
  d,
  condition = NULL,
  return.fields = c("Synthesis", "Half-life"),
  return.extra = NULL
)
```

Arguments

`d`	the data structure
`return.fields`	which fields should be extracted directly (may be NULL)
`return.extra`	apply a function returning a flat list or vector (may be NULL)
`condition`	if the original grandR object had `Condition` set, which condition to extract (NULL otherwise)

Value

the data flattened into a vector

Functions

- `kinetics2vector()`: Convert the output of the FitKinetics methods into a vector

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Condition",Design$dur.4sU,Design$Replicate))
sars <- Normalize(sars)
fit <- FitKineticsGeneLeastSquares(sars,"SRSF6")$Mock
print(fit)
kinetics2vector(fit)
```

`ToIndex` *Obtain the indices of the given genes*

Description

Genes can be referred to by their names, symbols, row numbers in the gene table, or a logical vector referring to the gene table rows. This function accepts all these possibilities and returns the row number in the gene table for the given genes,

Usage

```
ToIndex(data, gene, regex = FALSE, remove.missing = TRUE, warn = TRUE)
```

Arguments

`data`	The grandR object
`gene`	A vector of genes. Can be either numeric indices, gene names, gene symbols or a logical vector
`regex`	Treat gene as a regex and return all that match
`remove.missing`	if TRUE, do not return missing genes (return NA otherwise)
`warn`	if TRUE emit a warning if not all genes are found

Value

Numeric indices corresponding to the given genes

See Also

GeneInfo

Examples

```
sars <- ReadGRAND(system.file("extdata", "sars.tsv.gz", package = "grandR"),
                  design=c("Cell",Design$dur.4sU,Design$Replicate))
ToIndex(sars,c("MYC"))
ToIndex(sars,GeneInfo(sars)$Symbol=="MYC")
```

`Transform.no` *Transformations for PlotHeatmap*

Description

Functions to perform transformations on the matrix used for PlotHeatmap.

Usage

```
Transform.no(label = " ")

Transform.Z(label = "z score", center = TRUE, scale = TRUE)

Transform.VST(label = "VST")

Transform.logFC(label = "log2 FC", LFC.fun = NULL, columns = NULL, ...)
```

Arguments

`label`	label that is used for the heatmap legend
`center`	perform centering when computing Z scores (see scale)
`scale`	perform scaling when computing Z scores (see scale)
`LFC.fun`	function to compute log fold changes (default: PsiLFC, other viable option: NormLFC)
`columns`	which columns (i.e. samples or cells) to use as reference when computing log fold changes (see details)
`...`	further parameters passed down to LFC.fun

Details

These functions should be used as transform parameter to PlotHeatmap. Available data transformations are

- transform=Transform.Z(): compute z scores for each row; you can omit the usual centering or scaling by setting the respective parameters to false; see scale
- transform=Transform.VST(): do a variance stabilizing transformation using vst
- transform=Transform.logFC(): compute log2 fold changes to one or several reference columns; see below how to define them; fold changes are computed using the lfc package)
- transform=Transform.no(): do not transform

The label to be used in the heatmap legend can be changed by specifying the label parameter.

For Transform.logFC, columns can be given as a logical, integer or character vector representing a selection of the columns (samples or cells).

Value

A function that transforms a matrix.

`TransformSnapshot` *Estimate parameters for a one-shot experiment.*

Description

Under steady state conditions it is straight-forward to estimate s and d. Otherwise, the total levels at some other time point are needed.

Usage

```
TransformSnapshot(ntr, total, t, t0 = NULL, f0 = NULL, full.return = FALSE)
```

Arguments

`ntr`	the new to total RNA ratio (measured)
`total`	the total level of RNA (measured)
`t`	the labeling duration
`t0`	time before measurement at which f0 is total level (only necessary under non-steady-state conditions)
`f0`	total level at t0 (only necessary under non-steady-state conditions)
`full.return`	also return the provided parameters

Details

t0 must be given as the total time in between the measurement of f0 and the given ntr and total values!

Value

a named vector for s and d

`UpdateSymbols` *Update symbols using biomaRt*

Description

If your input files only contained ENSEMBL ids, use this to add gene symbols!

Usage

```
UpdateSymbols(data, species = NULL, current.value = "ensembl_gene_id")
```

Arguments

`data`	a grandR object
`species`	the species the genes belong to (eg "Homo sapiens"); can be NULL, then the species is inferred from gene ids (see details)
`current.value`	What it the current value in the symbols field?

Details

If no species is given, a very simple automatic inference is done, which will only work when having human or mouse ENSEMBL identifiers as gene ids. If you need to specify species, it must be one of `biomaRt::listDatasets(biomaRt::useMart("ensembl"))$dataset`!

Current.value must be one of `biomaRt::listAttributes(biomaRt::useMart("ensembl"))$name`!

Value

a grandR object with updated symbol names

`UseNtrSlot` *Copy the NTR slot and save under new name*

Description

Copy the NTR slot and save under new name

Usage

```
UseNtrSlot(data, name)
```

Arguments

`data`	the grandR object
`name`	the name of the new slot

Value

a grandR object

`VulcanoPlot` *Make a Vulcano plot*

Description

Plot log2 fold changes against -log10 multiple testing adjusted P values

Usage

```
VulcanoPlot(
  data,
  analysis = Analyses(data)[1],
  p.cutoff = 0.05,
  lfc.cutoff = 1,
  annotate.numbers = TRUE,
  ...
)
```

Arguments

`data`	the grandR object that contains the data to be plotted
`analysis`	the analysis to plot (default: first analysis)
`p.cutoff`	p-value cutoff (default: 0.05)
`lfc.cutoff`	log fold change cutoff (default: 1)
`annotate.numbers`	if TRUE, label the number of genes
`...`	further parameters passed to PlotScatter

Value

a ggplot object

Index