Package ‘EZbakR’

December 10, 2025

Title Analyze and Integrate Any Type of Nucleotide Recoding RNA-Seq Data

Version 0.1.0

Description A complete rewrite and reimagining of 'bakR' (see 'Vock et al.' (2025) <doi:10.1371/journal.pcbi.1013179>).
Designed to support a wide array of analyses of nucleotide recoding RNA-seq (NR-seq) datasets of any
type, including TimeLapse-seq/SLAM-seq/TUC-seq, Start-TimeLapse-seq (STL-seq),
TT-TimeLapse-seq (TT-TL-seq), and subcellular NR-seq. 'EZbakR' extends
standard NR-seq standard NR-seq mutational modeling to support multi-label analyses
(e.g., 4sU and 6sG dual labeling), and implements an improved hierarchical model to better
account for transcript-to-transcript variance in metabolic label incorporation.
'EZbakR' also generalized dynamical systems modeling of NR-seq data to support analyses
of premature mRNA processing and flow between subcellular compartments. Finally,
'EZbakR' implements flexible and well-powered comparative analyses of all
estimated parameters via design matrix-specified generalized linear modeling.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.2

Suggests knitr, MASS, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

Imports arrow, data.table, dplyr, ggplot2, magrittr, methods, purrr,
rlang, tidyr, tximport

Depends R (>= 3.5)

LazyData true

VignetteBuilder knitr

URL `https://isaacvock.github.io/EZbakR/`,
`https://github.com/isaacvock/EZbakR`

BugReports `https://github.com/isaacvock/EZbakR/issues`

NeedsCompilation no

Author Isaac Vock [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0002-7178-6886>)

Maintainer Isaac Vock <isaac.vock@gmail.com>

Repository CRAN

Date/Publication 2025-12-10 21:20:08 UTC

Contents

`AverageAndRegularize` *Average parameter estimates across replicates, and regularize variance estimates*

Description

`AverageAndRegularize` fits a generalized linear model to your data to effectively average parameter estimates across replicates and get overall uncertainty estimates for those parameters. The linear model to which your data is fit is specified via an R formula object supplied to the `formula_mean` parameter. Uncertainty estimates are regularized via a hierarchical modeling strategy originally introduced with bakR, though slightly improved upon since then.

Usage

```
AverageAndRegularize(
  obj,
  features = NULL,
  parameter = "log_kdeg",
  type = "kinetics",
  kstrat = NULL,
  populations = NULL,
  fraction_design = NULL,
  exactMatch = TRUE,
  repeatID = NULL,
  formula_mean = NULL,
  sd_grouping_factors = NULL,
  include_all_parameters = TRUE,
  sd_reg_factor = 10,
  error_if_singular = TRUE,
  min_reads = 10,
  convert_tl_to_factor = TRUE,
  regress_se_with_abs = TRUE,
  force_lm = FALSE,
  force_optim = force_lm,
  conservative = FALSE,
  character_limit = 20,
  feature_lengths = NULL,
  feature_sample_counts = NULL,
  scale_factor_df = NULL,
  overwrite = TRUE
)
```

Arguments

`obj`	An `EZbakRFractions` or `EZbakRKinetics` object, which is an `EZbakRData` object on which `EstimateFractions()` or `EstimateKinetics()` has been run.
`features`	Character vector of the set of features you want to stratify reads by and estimate proportions of each RNA population. The default of "all" will use all feature columns in the `obj`'s cB.
`parameter`	Parameter to average across replicates of a given condition.
`type`	What type of table is the parameter found in? Default is "kinetics", but can also set to "fractions".
`kstrat`	If `type` == `"kinetics"`, then `kstrat` specifies the kinetic parameter inference strategy.
`populations`	Character vector of the set of mutational populations that you want to infer the fractions of. Only relevant if type == "fractions".
`fraction_design`	"Design matrix" specifying which RNA populations exist in your samples. Only relevant if type == "fractions".
`exactMatch`	If TRUE, then `features` and `populations` have to exactly match those for a given fractions table for that table to be used. Means that you can't specify a subset of features or populations by default, since this is TRUE by default.
`repeatID`	If multiple `kinetics` or `fractions` tables exist with the same metadata, then this is the numerical index by which they are distinguished.
`formula_mean`	An R formula object specifying how the `parameter` of interest depends on the sample characteristics specified in `obj`'s metadf. The most common formula will be ~ `treatment` or ~ `treatment:duration`, where `treatment` and `duration` would be replaced with whatever you called the relevant sample characteristics in your metadf. ~ `treatment` means that an average value of `parameter` should be estimated for each set of samples with the same value for `treatment` in the metadf. ~ `treatment:duration` specifies that an average value of `parameter` should be estimated for each set of samples with the same combination of `treatment` and `duration` values in the metadf. An example of the latter case is a situation where you have two or more treatments (e.g., drug treated and untreated control) which were applied for different durations of time (e.g., 4 and 8 hours). NOTE: EZbakR automatically removes any intercept terms from the model. That way, there is no ambiguity about what parameter is defined as the reference.
`sd_grouping_factors`	What metadf columns should data be grouped by when estimating standard deviations across replicates? If this is NULL, then EZbakR will check to see if

the `formula_mean` specifies a formula that cleanly stratifies samples into disjoint groups. For example, the formula `~ treatment` will assign each sample to a single factor (its value for the metadf's `treatment` column). In this case, standard deviations can be calculated for sets of replicates in each `treatment` group. If such a stratification does not exist, a single standard deviation will be estimated for each feature (i.e., homoskedasticity will be assumed as in standard linear modeling).

`include_all_parameters`
If TRUE, an additional table will be saved with the prefix `fullfit_`, which includes all of the parameters estimated throughout the course of linear modeling and regularization. This can be nice for visualizing the regularized mean-variance trend.

`sd_reg_factor` Determines how strongly variance estimates are shrunk towards trend. Higher numbers lead to more regularization. Eventually, this will be replaced with estimation of how much variance there seems to be in the population of variances.

`error_if_singular`
If TRUE, linear model will throw an error if parameters cannot be uniquely identified. This is most often caused by parameters that cannot be estimated from the data, e.g., due to limited replicate numbers or correlated sample characteristics (i.e., all treatment As also correspond to batch As, and all treatment Bs correspond to batch Bs).

`min_reads` Minimum number of reads in all samples for a feature to be kept.

`convert_tl_to_factor`
If a label time variable is included in the `formula_mean`, convert its values to factors so as to avoid performing continuous regression on label times. Defaults to TRUE as including label time in the regression is often meant to stratify samples by their label time if, for example, you are averaging logit(fractions).

`regress_se_with_abs`
If TRUE, and if `type == "fractions"`, then standard error will be regressed against logit fraction rather than magnitude of logit fraction. Makes sense to set this to FALSE if analyzing certain site-specific mutational probing methods when high mutation content things are likely low variance SNPs.

`force_lm` Certain formula lend them selves to efficient approximations of the full call to `lm()`. Namely, formulas that stratify samples into disjoint groups where a single parameter of the model is effectively estimated from each group can be tackled via simple averaging of data from each from group. If you would like to force EZbakR to fit the fully rigorous linear model though, set `force_lm` to `TRUE`.

`force_optim` Old parameter that is now passed the value `force_lm` and will be deprecated in later releases

`conservative` If TRUE, conservative variance regularation will be performed. In this case, variances below the trend will be regularized up to the trend, and variances above the trend will be left unregularized. This avoids underestimation of variances.

`character_limit`
Limit on the number of characters of the name given to the output table. Will attempt to concatenate the parameter name with the names of all of the features. If this is too long, only the parameter name will be used.

`feature_lengths`
: Table of effective lengths for each feature combination in your data. For example, if your analysis includes features named GF and XF, this should be a data frame with columns GF, XF, and length.

`feature_sample_counts`
: Data frame with columns `<feature names>` and `nsamps`, where `<feature names>` are all of the feature columns in the input to `AverageAndRegularize()`, and `nsamps` is the number of samples that samples from that feature combination needs to have over the read count threshold.

`scale_factor_df`
: Data frame with columns "sample" and a second column of whatever name you please. The second column should denote scale factors by which read counts in that sample should be multiplied by in order to normalize these read counts.

`overwrite`
: If TRUE, identical, existing output will be overwritten.

Details

The EZbakR website has an extensive vignette walking through various use cases and model types that you can fit with `AverageAndRegularize()`: vignette link. EZbakR improves upon bakR by balancing extra conservativeness in several steps with a more highly powered statistical testing scheme in its `CompareParameters()` function. In particular, the following changes to the variance regularization scheme were made:

- Sample-specific parameter uncertainties are used to generate conservative estimates of feature-specific replicate variabilties. In addition, a small floor is set to ensure that replicate variance estimates are never below a certain level, for the same reason.
- Condition-wide replicate variabilities are regressed against both read coverage and either a) |logit(estimate)| when modeling average fraction labeled. This captures the fact thta estimates are best around a logit(fraction labeled) of 0 and get worse for more extreme fraction labeled's.; b) log(kdeg) when modeling log degradation rate constants. At first, I considered a strategy similar to the fraction labeled modeling, but found that agreement between a fully rigorous MCMC sampling approach and EZbakR was significantly improved by just regressing hee value of the log kientic parameter, likely due to the non-linear transformation of fraction labeled to log(kdeg); and c) only coverage in all other cases.
- Features with replicate variabilities below the inferred trend have their replicate variabilites set equal to that predicted by the trend. This helps limit underestimation of parameter variance. Features with above-trend replicate variabilties have their replicate variabilities regularized with a Normal prior Normal likelihood Bayesian model, as in bakR (so the log(variance) is the inferred mean of this distribution, and the known variance is inferred from the amount of variance about the linear dataset-wide trend).

All of this allows `CompareParameters()` to use a less conservative statistical test when calculating p-values, while still controlling false discovery rates.

Value

`EZbakRData` object with an additional "averages" table, as well as a fullfit table under the same list heading, which includes extra information about the priors used for regularization purposes.

Examples

```
# Simulate data to analyze
simdata <- EZSimulate(30)

# Create EZbakR input
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)

# Estimate Fractions
ezbdo <- EstimateFractions(ezbdo)

# Estimate Kinetics
ezbdo <- EstimateKinetics(ezbdo)

# Average estimates across replicate
ezbdo <- AverageAndRegularize(ezbdo)
```

`CompareParameters` *Get contrasts of estimated parameters*

Description

`CompareParameters()` calculates differences in parameters estimated by `AverageAndRegularize()` or `EZDynamics()` and performs null hypothesis statistical testing, comparing their values to a null hypothesis of 0.

Usage

```
CompareParameters(
  obj,
  design_factor,
  reference,
  experimental,
  param_name,
  parameter = "log_kdeg",
  type = "averages",
  param_function,
  condition = NULL,
  features = NULL,
  exactMatch = TRUE,
  repeatID = NULL,
  formula_mean = NULL,
  sd_grouping_factors = NULL,
  fit_params = NULL,
  normalize_by_median = FALSE,
  reference_levels = NULL,
  experimental_levels = NULL,
  overwrite = TRUE
)
```

Arguments

`obj`	An `EZbakRFit` object, which is an `EZbakRFractions` object on which `AverageAndRegularize()` has been run.
`design_factor`	Name of factor from `metadf` whose parameter estimates at different factor values you would like to compare. If you specify this, you need to also specify `reference` and `experimental`. If `type` == "dynamics", this can have multiple values, being the names of all of the factors you would like to stratify a group by.
`reference`	Name of reference `design_factor` factor level value. Difference will be calculated as `experimental - reference`. If type == "dynamics", then this should specify the levels of all of the `design_factor`(s) reference group. For example, if you have multiple `design_factor`'s, then `reference` must be a named character vector with one element per `design_factor`, with elements named the corresponding `design_factor`. For example, if `design_factor` is c("genotype", "treatment"), and you would like to compare genotype = "WT" and treatment = "untreated" (reference) to genotype = "KO" and treatment = "treated", then `reference` would need to be a vector with one element named "genotype", equal to "WT" and one element named "treatment" equal to "untreated" (this example could be created with, `c(genotype = "WT", treatment = "untreated")`).
`experimental`	Name of `design_factor` factor level value to compare to reference. Difference will be calculated as `experimental - reference`. If type == "dynamics", then this should specify the levels of all of the `design_factor`(s) reference group. For example, if you have multiple `design_factor`'s, then `experimental` must be a named character vector with one element per `design_factor`, with elements named the corresponding `design_factor`. For example, if `design_factor` is c("genotype", "treatment"), and you would like to compare genotype = "WT" and treatment = "untreated" (reference) to genotype = "KO" and treatment = "treated", then `experimental` would need to be a vector with one element named "genotype", equal to "KO" and one element named "treatment" equal to "treated" (this example could be created with, `c(genotype = "KO", treatment = "treated")`).
`param_name`	If you want to assess the significance of a single parameter, rather than the comparison of two parameters, specify that one parameter's name here.
`parameter`	Parameter to average across replicates of a given condition.
`type`	Type of table to use. Can either be "averages" or "dynamics".
`param_function`	NOT YET IMPLEMENTED. Will allow you to specify more complicated functions of parameters when hypotheses you need to test are combinations of parameters rather than individual parameters or simple differences in two parameters.
`condition`	Same as `design_factor`, will be deprecated in favor of the former in later release.
`features`	Character vector of the set of features you want to stratify reads by and estimate proportions of each RNA population. The default of "all" will use all feature columns in the `obj`'s cB.

`exactMatch`	If TRUE, then `features` and `populations` have to exactly match those for a given fractions table for that table to be used. Means that you can't specify a subset of features or populations by default, since this is TRUE by default.
`repeatID`	If multiple `averages` tables exist with the same metadata, then this is the numerical index by which they are distinguished.
`formula_mean`	An R formula object specifying how the `parameter` of interest depends on the sample characteristics for the averages object you want to use.
`sd_grouping_factors`	Metadf columns should data was grouped by when estimating standard deviations across replicates for the averages object you want to use.
`fit_params`	Character vector of parameter names in the averages object you would like to use.
`normalize_by_median`	If TRUE, then median difference will be set equal to 0. This can account for global biases in parameter estimates due to things like differences in effective label times. Does risk eliminating real signal though, so user discretion is advised.
`reference_levels`	Same as `reference`, but exclusively parsed in case of `type` == "dynamics, included for backwards compatibility.
`experimental_levels`	Same as `experimental`, but exclusively parsed in case of `type` == "dynamics, included for backwards compatibility.
`overwrite`	If TRUE, then identical output will be overwritten if it exists.

Details

The EZbakR website has an extensive vignette walking through various use cases and parameters you can compare with `CompareParameters()`: vignette link.

There are essentially 3 scenarios that `CompareParameters()` can handle:

- Pairwise comparisons: compare `reference` to `experimental` parameter estimates of a specified `design_factor` from `AverageAndRegularize()`. log(`experimental` / `reference`) is the computed "difference" in this case.
- Assess the value of a single parameter, which itself should represent a difference between other parameters. The name of this parameter can be specified via the `param_name` argument. This is useful for various interaction models where some of the parameters of these models may represent things like "effect of A on condition X".
- Pairwise comparison of dynamical systems model parameter estimate: similar to the first case listed above, but now when `type == "dynamics"`. `design_factor` can now be a vector of all the `metadf` columns you stratified parameter estimates by.

Eventually, a 4th option via the currently non-functional `param_function` argument will be implemented, which will allow you to specify functions of parameters to be assessed, which can be useful for certain interaction models.

`CompareParameters()` calculates p-values using what is essentially an asymptotic Wald test, meaning that a standard normal distribution is integrated. P-values are then multiple-test adjusted using the method of Benjamini and Hochberg, implemented in R's `p.adjust()` function.

Value

EZbakRData object with an additional "comparisons" table, detailing the result of a comparison of a parameter estimate's valules across two different conditions.

Examples

```
# Simulate data to analyze
simdata <- EZSimulate(30)

# Create EZbakR input
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)

# Estimate Fractions
ezbdo <- EstimateFractions(ezbdo)

# Estimate Kinetics
ezbdo <- EstimateKinetics(ezbdo)

# Average estimates across replicate
ezbdo <- AverageAndRegularize(ezbdo)

# Compare parameters across conditions
ezbdo <- CompareParameters(
ezbdo,
design_factor = "treatment",
reference = "treatment1",
experimental = "treatment2"
)
```

CorrectDropout *Correct for experimental/bioinformatic dropout of labeled RNA.*

Description

Uses the strategy described here, and similar to that originally presented in Berg et al. 2024.

Usage

```
CorrectDropout(
  obj,
  strategy = c("grandR", "bakR"),
  grouping_factors = NULL,
  features = NULL,
  populations = NULL,
  fraction_design = NULL,
  repeatID = NULL,
  exactMatch = TRUE,
  read_cutoff = 25,
  dropout_cutoff = 5,
  ...
)
```

Arguments

`obj`	An EZbakRFractions object, which is an EZbakRData object on which you have run `EstimateFractions()`.
`strategy`	Which dropout correction strategy to use. Options are: • grandR: Described here. Cite that work and grandR if using this strategy. Quasi-non-parametric strategy that finds an estimate of the dropout rate that eliminates any linear correlation between the newness of a transcript and the difference in +s4U and -s4U normalized read counts. • bakR: Described here. Uses a simple generative model of dropout to derive a likelihood function, and the dropout rate is estimated via the method of maximum likelihood. The "bakR" strategy has the advantage of being model-derived, making it possible to assess model fit and thus whether the simple assumptions of both the "bakR" and "grandR" dropout models are met. The "grandR" strategy has the advantage of being more robust. Thus, the "grandR" strategy is currently used by default.
`grouping_factors`	Which sample-detail columns in the metadf should be used to group -s4U samples by for calculating the average -s4U RPM? The default value of `NULL` will cause all sample-detail columns to be used.
`features`	Character vector of the set of features you want to stratify reads by and estimate proportions of each RNA population. The default of `NULL` will expect there to be only one fractions table in the EZbakRFractions object.
`populations`	Mutational populations that were analyzed to generate the fractions table to use. For example, this would be "TC" for a standard s4U-based nucleotide recoding experiment.
`fraction_design`	"Design matrix" specifying which RNA populations exist in your samples. By default, this will be created automatically and will assume that all combinations of the `mutrate_populations` you have requested to analyze are present in your data. If this is not the case for your data, then you will have to create one manually. See docs for `EstimateFractions` (run ?EstimateFractions()) for more details.
`repeatID`	If multiple `fractions` tables exist with the same metadata, then this is the numerical index by which they are distinguished.
`exactMatch`	If TRUE, then `features` must exactly match the `features` metadata for a given fractions table for it to be used. Means that you cannot specify a subset of features by default. Set this to FALSE if you would like to specify a feature subset.
`read_cutoff`	Minimum number of reads for a feature to be used to fit the dropout model.

`dropout_cutoff`	Maximum ratio of -s4U:+s4U RPMs for a feature to be used to fit the dropout model (i.e., simple outlier filtering cutoff).
`...`	Parameters passed to internal `calculate_dropout()` function; namely `dropout_cutoff_min`, which sets the minimum dropout value used for fitting the dropout model.

Details

Dropout is the disproportionate loss of labeled RNA/reads from said RNA described independently here and here. It can originate from a combination of bioinformatic (loss of high mutation content reads due to alignment problems), technical (loss of labeled RNA during RNA extraction), and biological (transcriptional shutoff in rare cases caused by metabolic label toxicity) sources. `CorrectDropout()` compares label-fed and label-free controls from the same experimental conditions to estimate and correct for this dropout. It assumes that there is a single number (referred to as the dropout rate, or pdo) which describes the rate at which labeled RNA is lost (relative to unlabeled RNA). pdo ranges from 0 (no dropout) to 1 (complete loss of all labeled RNA), and is thus interpreted as the percentage of labeled RNA/reads from labeled RNA disproportionately lost, relative to the equivalent unlabeled species.

Value

An `EZbakRData` object with the specified "fractions" table replaced with a dropout corrected table.

Examples

```
# Simulate data to analyze
simdata <- EZSimulate(30)

# Create EZbakR input
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)

# Estimate Fractions
ezbdo <- EstimateFractions(ezbdo)

# Correct for dropout
ezbdo <- CorrectDropout(ezbdo)
```

`create_fraction_design` *Generate a* `fraction_design` *table for* `EstimateFractions`.

Description

A `fraction_design` table denotes what populations of labeled/unlabeled RNA are present in your data. A `fraction_design` table as one column for each mutation type (e.g., TC) present in your cB file, and one column named "present". Each entry is either `TRUE` or `FALSE`. The rows include all possible combinations of `TRUE` and `FALSE` for all mutation types columns. A value of `TRUE` in a mutation type column represents a population of reads that have high amounts (on average) of that mutation type. For example, if your `fraction_design` table has mutation type columns "TC" and "GA", the row with TC == `TRUE` and GA == `FALSE` represents a population of reads with high T-to-C mutation content and low G-to-A mutation content. In other words, these are reads from RNA synthesized in the presence of s4U but not s6G. If such a population exists in your data, the "present" column for that row should have a value of `TRUE`.

Usage

```
create_fraction_design(mutrate_populations)
```

Arguments

`mutrate_populations`	Character vector of the set of mutational populations present in your data. For example, s4U fed data with standard nucleotide recoding chemistry (e.g., Time-Lapse, SLAM, TUC, AMUC, etc.) would have a `mutrate_populations` of c("TC"). Dual labeling experiments with s4U and s6G feeds would have a `mutrate_populations` of c("TC", "GA").

Value

A `fraction_design` table that assumes that every possible combination of mutational populations listed in `mutrate_populations` are present in your data. The `present` column can be modified if this assumption is incorrect. This default is chosen as it will in theory work for all analyses, it may just be unnecessarily inefficient and estimate the abundance of populations that don't exist.

Examples

```
# Standard, single-label NR-seq
fd <- create_fraction_design(c("TC"))

# Dual-label NR-seq
fd2 <- create_fraction_design(c("TC", "GA"))

# Adjust dual-label output for TILAC
fd2$present <- ifelse(fd2$TC & fd2$GA, FALSE, fd2$present)
```

`DeconvolveFractions` *Deconvolve multi-feature fractions.*

Description

Combines the output of `EstimateFractions` with feature quantification performed by an outside tool (e.g., RSEM, kallisto, salmon, etc.) to infer fraction news for features that reads cannot always be assigned to unambiguously. This is a generalization of `EstimateIsoformFractions` which performs this deconvolution for transcript isoforms.

Usage

```
DeconvolveFractions(
  obj,
  feature_type = c("gene", "isoform"),
  features = NULL,
  populations = NULL,
  fraction_design = NULL,
  repeatID = NULL,
  exactMatch = TRUE,
  fraction_name = NULL,
  quant_name = NULL,
  gene_to_transcript = NULL,
  overwrite = TRUE,
  TPM_min = 1,
  count_min = 10
)
```

Arguments

`obj`	An `EZbakRData` object
`feature_type`	Either `"gene"` (if deconvolving gene-level fraction news, i.e., for resolving fusion gene and component gene fraction news) or `"isoform"` (if deconvolving transcript isoform fraction news).
`features`	Character vector of the set of features you want to stratify reads by and estimate proportions of each RNA population. The default of "all" will use all feature columns in the `obj`'s cB.
`populations`	Mutational populations that were analyzed to generate the fractions table to use. For example, this would be "TC" for a standard s4U-based nucleotide recoding experiment.
`fraction_design`	"Design matrix" specifying which RNA populations exist in your samples. By default, this will be created automatically and will assume that all combinations of the `mutrate_populations` you have requested to analyze are present in your data. If this is not the case for your data, then you will have to create one manually. See docs for `EstimateFractions` (run ?EstimateFractions()) for more details.
`repeatID`	If multiple `fractions` tables exist with the same metadata, then this is the numerical index by which they are distinguished.
`exactMatch`	If TRUE, then `features` and `populations` have to exactly match those for a given fractions table for that table to be used. Means that you can't specify a subset of features or populations by default, since this is TRUE by default.
`fraction_name`	Name of fraction estimate table to use. Should be stored in the `obj$fractions` list under this name. Can also rely on specifying `features` and/or `populations` and having `EZget()` find it.
`quant_name`	Name of transcript isoform quantification table to use. Should be stored in the obj$readcounts list under this name. Use `ImportIsoformQuant()` to create this table. If `quant_name` is `NULL`, it will search for tables containing the

	string "isoform_quant" in their name, as that is the naming convention used by `ImportIsoformQuant()`. If more than one such table exists, an error will be thrown and you will have to specify the exact name in `quant_name`.
`gene_to_transcript`	Table with columns `transcript_id` and all feature related columns that appear in the relevant fractions table. This is only relevant as a hack to to deal with the case where STAR includes in its transcriptome alignment transcripts on the opposite strand from where the RNA actually originated. This table will be used to filter out such transcript-feature combinations that should not exist.
`overwrite`	If TRUE and a fractions estimate output already exists that would possess the same metadata (features analyzed, populations analyzed, and fraction_design), then it will get overwritten with the new output. Else, it will be saved as a separate output with the same name + "_#" where "#" is a numerical ID to distinguish the similar outputs.
`TPM_min`	Minimum TPM for a transcript to be kept in analysis.
`count_min`	Minimum expected_count for a transcript to be kept in analysis.

Details

`DeconvolveFractions` expects as input a "fractions" table with estimates for fraction news of at least one convolved feature set. A convolved feature set is one where some reads cannot be unambiguously assigned to one instance of that feature type. For example, it is often impossible to assign short reads to a single transcript isoform. Thus, something like the "TEC" assignment provided by fastq2EZbakR is an instance of a convolved feature set, as it is an assignment of reads to transcript isoforms with which they are compatible. Another example is assignment to the exonic regions of genes, for fusion genes (where a read may be consistent with both the putative fusion gene as well as one of the fusion components).

`DeconvolveFractions` deconvolves fraction news by fitting a linear mixing model to the convolved fraction new estimates + feature abundance estimates. In other words, each convolved fraction new (data) is modeled as a weighted average of single feature fraction news (parameters to estimate), with the weights determined by the relative abundances of the features in the convolved set (data). The convolved fraction new is modeled as a beta distribution with mean equal to the weighted feature fraction new average and variance related to the number of reads in the convolved feature set.

Features with estimated TPMs less than `TPM_min` (1 by default) or an estimated number of read counts less than `count_min` (10 by default) are removed from convolved feature sets and will not have their fraction news estimated.

Value

An `EZbakRData` object with an additional table under the "fractions" list. Has the same form as the output of `EstimateFractions()`, and will have the feature column "transcript_id".

Examples

```
# Load dependencies
library(dplyr)
```

```
# Simulates a single sample worth of data
simdata_iso <- SimulateIsoforms(nfeatures = 30)

# We have to manually create the metadf in this case
metadf <- tibble(sample = 'sampleA',
                     tl = 4,
                     condition = 'A')

ezbdo <- EZbakRData(simdata_iso$cB,
                    metadf)

ezbdo <- EstimateFractions(ezbdo)

### Hack in the true, simulated isoform levels
reads <- simdata_iso$ground_truth %>%
  dplyr::select(transcript_id, true_count, true_TPM) %>%
  dplyr::mutate(sample = 'sampleA',
                effective_length = 10000) %>%
  dplyr::rename(expected_count = true_count,
                TPM = true_TPM)

# Name of table needs to have "isoform_quant" in it
ezbdo[['readcounts']][['simulated_isoform_quant']] <- reads

### Perform deconvolution
ezbdo <- DeconvolveFractions(ezbdo, feature_type = "isoform")
```

`EstimateFractions` *Estimate fractions of each RNA population.*

Description

The first step of any NR-seq analysis is to figure out the fraction of reads from each mutational population in your data. For example, if you are performing a standard SLAM-seq or TimeLapse-seq experiment, this means estimating the fraction of reads with high T-to-C mutation content, and the fraction with low T-to-C mutation content. This is what `EstimateFractions` is for.

Usage

```
EstimateFractions(
  obj,
  features = "all",
  mutrate_populations = "all",
  fraction_design = NULL,
  Poisson = TRUE,
  strategy = c("standard", "hierarchical"),
  filter_cols = "all",
  filter_condition = `&`,
  remove_features = c("NA", "__no_feature"),
  split_multi_features = FALSE,
  multi_feature_cols = NULL,
  multi_feature_sep = "+",
  pnew_prior_mean = -2.94,
  pnew_prior_sd = 0.3,
  pold_prior_mean = -6.5,
  pold_prior_sd = 0.5,
  hier_readcutoff = 300,
  init_pnew_prior_sd = 0.8,
  pnew_prior_sd_min = 0.01,
  pnew_prior_sd_max = 0.15,
  pold_est = NULL,
  pold_from_nolabel = FALSE,
  grouping_factors = NULL,
  character_limit = 20,
  overwrite = TRUE
)

## S3 method for class 'EZbakRData'
EstimateFractions(
  obj,
  features = "all",
  mutrate_populations = "all",
  fraction_design = NULL,
  Poisson = TRUE,
  strategy = c("standard", "hierarchical"),
  filter_cols = "all",
  filter_condition = `&`,
  remove_features = c("NA", "__no_feature"),
  split_multi_features = FALSE,
  multi_feature_cols = NULL,
  multi_feature_sep = "+",
  pnew_prior_mean = -2.94,
  pnew_prior_sd = 0.3,
  pold_prior_mean = -6.5,
  pold_prior_sd = 0.5,
  hier_readcutoff = 300,
  init_pnew_prior_sd = 0.3,
  pnew_prior_sd_min = 0.01,
  pnew_prior_sd_max = 0.15,
  pold_est = NULL,
  pold_from_nolabel = FALSE,
  grouping_factors = NULL,
  character_limit = 20,
  overwrite = TRUE
)
```

```
## S3 method for class 'EZbakRArrowData'
EstimateFractions(
  obj,
  features = "all",
  mutrate_populations = "all",
  fraction_design = NULL,
  Poisson = TRUE,
  strategy = c("standard", "hierarchical"),
  filter_cols = "all",
  filter_condition = `&`,
  remove_features = c("NA", "__no_feature"),
  split_multi_features = FALSE,
  multi_feature_cols = NULL,
  multi_feature_sep = "+",
  pnew_prior_mean = -2.94,
  pnew_prior_sd = 0.3,
  pold_prior_mean = -6.5,
  pold_prior_sd = 0.5,
  hier_readcutoff = 300,
  init_pnew_prior_sd = 0.8,
  pnew_prior_sd_min = 0.01,
  pnew_prior_sd_max = 0.15,
  pold_est = NULL,
  pold_from_nolabel = FALSE,
  grouping_factors = NULL,
  character_limit = 20,
  overwrite = TRUE
)
```

Arguments

`obj` — `EZbakRData` or `EZbakRArrowData` object

`features` — Character vector of the set of features you want to stratify reads by and estimate proportions of each RNA population. The default of "all" will use all feature columns in the `obj`'s cB file.

`mutrate_populations`
Character vector of the set of mutational populations that you want to infer the rates of mutations for. By default, all mutation rates are estimated for all populations present in cB.

`fraction_design`
"Design matrix" specifying which RNA populations exist in your samples. By default, this will be created automatically and will assume that all combinations of the `mutrate_populations` you have requested to analyze are present in your data. If this is not the case for your data, then you will have to create one manually.

If you call the function `create_fraction_design(...)`, providing a vector of mutational population names as input, it will create a `fraction_design` table for you, with the assumption that every single possible combination of mutational populations is present in your data. You can then edit the `present` column as necessary to get an appropriate `fraction_design` for your use case. See below for details on the required contents of `fraction_design` and its interpretation.

`fraction_design` must have one column per element of `mutrate_populations`, with these columns sharing the name of the `mutrate_populations`. It must also have one additional column named `present`. All elements of fraction_design should be booleans (`TRUE` or `FALSE`). It should include all possible combinations of `TRUE` and `FALSE` for the `mutrate_populations` columns. A `TRUE` in one of these columns represents a population of RNA that is expected to have above background mutation rates of that type. `present` will denote whether or not that population of RNA is expected to exist in your data.

For example, assume you are doing a typical TimeLapse-seq/SLAM-seq/TUC-seq/etc. experiment where you have fed cells with s^4U and recoded any incorporated s^4U to a nucleotide that reverse transcriptase will read as a cytosine. That means that `mutrate_populations` will be "TC", since you want to estimate the fraction of RNA that was s^4U labeled, i.e., the fraction with high T-to-C mutation content. `fraction_design` will thus have two columns: `TC` and `present`. It will also have two rows. One of these rows must have a value of `TRUE` for `TC`, and the other must have a value of `FALSE`. The row with a value of `TRUE` for `TC` represents the population of reads with high T-to-C mutation content, i.e., the reads from RNA that were synthesized while s^4U was present. The row with a value of `FALSE` for `TC` reprsents the population of reads with low T-to-C mutation content, i.e., the reads from RNA that existed prior to s^4U labeling. Both of these populations exist in your data, so the value of the `present` column should be `TRUE` for both of these. See the lazily loaded `standard_fraction_design` object for an example of what this tibble could look like. ("lazily loaded `standard_fraction_design` object" means that if you run `print(standard_fraction_design)` after loading `EZbakR` with `library(EZbakR)`, then you can see its contents. More specifically, lazily loaded means that this table is not loaded into memory until you ask for it, via something like a `print()` call.)

As another example, consider TILAC, a NR-seq extension developed by the Simon lab. TILAC was originally described in Courvan et al., 2022. In this method, two populations of RNA, one from s^4U fed cells and one from s^6G fed cells, are pooled and prepped for sequencing together. This allows for internally controlled comparisons of RNA abundance without spike-ins. s^4U is recoded to a cytosine analog by TimeLapse chemistry (or similar chemistry) and s^6G is recoded to an adenine analog. Thus, `fraction_design` includes columns called `TC` and `GA`. A unique aspect of the TILAC `fraction_design` table is that one of the possible populations, `TC` and `GA` both `TRUE`, is denoted as not present (`present = FALSE`). This is because there is no RNA that was exposed to both s^4U and s^6G, thus a population of reads with both high T-to-C and G-to-A mutational content should not exist. To see an example of what a TILAC `fraction_design` table could look like, see the lazily loaded `tilac_fraction_design` object.

`Poisson` If `TRUE`, use U-content adjusted Poisson mixture modeling strategy. Often provides significant speed gain without sacrificing accuracy.

`strategy` String denoting which new read mutation rate estimation strategy to use. Options include:

- standard: Estimate a single new read and old read mutation rate for each sample. This is done via a binomial mixture model aggregating over
- hierarchical: Estimate feature-specific new read mutation rate, regularizing the feature-specific estimate with a sample-wide prior. Currently only compatible with single mutation type mixture modeling.

`filter_cols` Which feature columns should be used to filter out feature-less reads. The default value of "all" checks all feature columns for whether or not a read failed to get assigned to said feature.

`filter_condition`
Only two possible values for this make sense: `` `&` `` and `` `|` ``. If set to `` `&` ``, then all features in `filter_cols` must have a "null" value (i.e., a value included in `remove_features`) for the row to get filtered out. If set to `` `|` ``, then only a single feature in `filter_cols` needs to have one of these "null" values to get filtered out.

`remove_features`
All of the feature names that could indicate failed assignment of a read to a given feature. the fastq2EZbakR pipeline uses a value of '__no_feature'.

`split_multi_features`
If a set of reads maps ambiguously to multiple features, should data for such reads be copied for each feature in the ambiguous set? If this is `TRUE`, then `multi_feature_cols` also must be set. Examples where this should be set to `TRUE` includes when analyzing exonic bins (concept defined in original DEXSeq paper), exon-exon junctions, etc.

`multi_feature_cols`
Character vector of columns that have the potential to include assignment to multiple features. Only these columns will have their features split if `split_multi_features` is `TRUE`.

`multi_feature_sep`
String representing how ambiguous feature assignments are distinguished in the feature names. For example, the default value of "+" denotes that if a read maps to multiple features (call them featureA and featureB, for example), then the feature column will have a value of "featureA+featureB" for that read.

`pnew_prior_mean`
Mean for logit(pnew) prior.

`pnew_prior_sd` Standard deviation for logit(pnew) prior.

`pold_prior_mean`
Mean for logit(pold) prior.

`pold_prior_sd` Standard deviation for logit(pold) prior.

`hier_readcutoff`
If `strategy` == `hierarchical`, only features with this many reads are used to infer the distribution of feature-specific labeled read mutation rates.

`init_pnew_prior_sd`
If `strategy` == `hierarchical`, this is the initial logit(pnew) prior standard deviation to regularize feature-specific labeled read mutation rate estimates.

`pnew_prior_sd_min`
: The minimum logit(pnew) prior standard deviation when `strategy` is set to "hierarchcial". EZbakR will try to estimate this empirically as the standard deviation of initial feature-specific logit(pnew) estimates using high coverage features, minus the average uncertainty in the logit(pnew) estimates. As this difference can sometimes be negative, a value of `pnew_prior_sd_min` will be imputed in that case.

`pnew_prior_sd_max`
: Similar to `pnew_prior_sd_min`, but now representing the maximum allowed logit(pnew) prior sd.

`pold_est`
: Background mutation rate estimates if you have them. Can either be a single number applied to all samples or a named vector of values, where the names should be sample names.

`pold_from_nolabel`
: Fix background mutation rate estimate to mutation rates seen in -label samples. By default, a single background rate is used for all samples, inferred from the average mutation rate across all -label samples. The `grouping_factors` argument can be specified to use certain -label samples to infer background mutation rates for certain sets of +label samples.

`grouping_factors`
: If `pold_from_nolabel` is TRUE, then `grouping_factors` will specify the sample-detail columns in the metadf that should be used to group -label samples by. Average mutation rates in each group of -label samples will be used as the background mutation rate estimate in +label samples with the same values for the relevant metadf columns.

`character_limit`
: Maximum number of characters for naming out fractions output. EZbakR will try to name this as a "_" separated character vector of all of the features analyzed. If this name is greater than `character_limit`, then it will default to "fraction#", where "#" represents a simple numerical ID for the table.

`overwrite`
: If TRUE and a fractions estimate output already exists that would possess the same metadata (features analyzed, populations analyzed, and fraction_design), then it will get overwritten with the new output. Else, it will be saved as a separate output with the same name + "_#" where "#" is a numerical ID to distinguish the similar outputs.

Details

`EstimateFractions` uses mixture modeling to estimate the fraction of reads from each mutational population in your data, and this is done for each feature in your data (i.e., combination of columns that specify genomic features from which reads were derived). The set of mutational populations in your data can be specified by providing a `fraction_design` object, described in more depth above (also see `?create_fraction_design`). There are several flavors of mixture modeling that can be performed by `EstimateFractions`. These are as follows:

1. The default: global mutation rate parameters (global = same for all reads from all features) are estimated for each sample by fitting a single two-component mixture model to all of the reads in that sample. These are used to estimate the fraction of reads from each feature that are from each mutational population, also using a two-component mixture model.

    - With `Poisson` set to `TRUE`, this is a nucleotide-content adjusted Poisson mixture model, which is a more efficient alternative to binomial mixture modeling.
    - With `Poisson` set to `FALSE`, this is a binomial mixture model.

2. Low mutation rate from -label: if `pold_from_nolabel` is `TRUE`, then the background, no label mutation rates are estimated from -label samples. By default, a single set of background mutation rates are estimated for all samples, but you can change this behavior by setting `grouping_factors` to specify columns in your `metadf` by which samples should be stratified.
    - This is a great strategy to use if you have low label incorporation rates or if you used a fairly short label time.
3. Hierarchical model: if `strategy == "hierarchical"`, which is currently only compatible for single-mutation type modeling (e.g., standard T-to-C mutation modeling), then high T-to-C content mutation rates are estimated for each feature. The global sample-wide estimates are used as an informative prior to increase the accuracy of this process by avoiding extreme estimates.

Value

An `EZbakRFractions` object, which is just an `EZbakRData` object with a "fractions" list element. This will include tables of fraction estimates (e.g., fraction of reads from the high T-to-C mutation rate population in a standard single-label s4U experiment; termed fraction_highTC in the table) for all features in all samples.

Methods (by class)

- `EstimateFractions(EZbakRData)`: Method for class **EZbakRData** Estimates fractions using an entirely in memory object.
- `EstimateFractions(EZbakRArrowData)`: Mehthod for class **EZbakRArrowData** This is an alternative to the fully in memory **EZbakRData** `EstimateFractions` method that can help with analyses of larger than RAM datasets. The provided "cB" is expected to be an on-disk Arrow Dataset. Furthermore, it is expected to be partitioned by the sample name, which will allow this method to read only a single-sample worth of data into memory at a time. This can significantly reduce RAM usage. Input object should be created with `EZbakRArrowData()`.

Examples

```
# Simulate data to analyze
simdata <- SimulateOneRep(30)

# Create EZbakR input
metadf <- data.frame(sample = "sampleA", tl = 2)
ezbdo <- EZbakRData(simdata$cB, metadf)

# Estimate fractions
ezbdo <- EstimateFractions(ezbdo)
```

EstimateIsoformFractions

Estimate isoform-specific fraction news (or more generally "fractions").

Description

Combines the output of `EstimateFractions` with transcript isoform quantification performed by an outside tool (e.g., RSEM, kallisto, salmon, etc.) to infer transcript isoform-specific fraction news (or more generally fraction of reads coming from a particular mutation population).

Usage

```
EstimateIsoformFractions(
  obj,
  features = NULL,
  populations = NULL,
  fraction_design = NULL,
  repeatID = NULL,
  exactMatch = TRUE,
  fraction_name = NULL,
  quant_name = NULL,
  gene_to_transcript = NULL,
  overwrite = TRUE,
  TPM_min = 1,
  count_min = 10
)
```

Arguments

`obj`	An `EZbakRData` object
`features`	Character vector of the set of features you want to stratify reads by and estimate proportions of each RNA population. The default of "all" will use all feature columns in the `obj`'s cB.
`populations`	Mutational populations that were analyzed to generate the fractions table to use. For example, this would be "TC" for a standard s4U-based nucleotide recoding experiment.
`fraction_design`	"Design matrix" specifying which RNA populations exist in your samples. By default, this will be created automatically and will assume that all combinations of the `mutrate_populations` you have requested to analyze are present in your data. If this is not the case for your data, then you will have to create one manually. See docs for `EstimateFractions` (run ?EstimateFractions()) for more details.
`repeatID`	If multiple `fractions` tables exist with the same metadata, then this is the numerical index by which they are distinguished.

`exactMatch`	If TRUE, then `features` and `populations` have to exactly match those for a given fractions table for that table to be used. Means that you can't specify a subset of features or populations by default, since this is TRUE by default.
`fraction_name`	Name of fraction estimate table to use. Should be stored in the `obj$fractions` list under this name. Can also rely on specifying `features` and/or `populations` and having `EZget()` find it.
`quant_name`	Name of transcript isoform quantification table to use. Should be stored in the obj$readcounts list under this name. Use `ImportIsoformQuant()` to create this table. If `quant_name` is `NULL`, it will search for tables containing the string "isoform_quant" in their name, as that is the naming convention used by `ImportIsoformQuant()`. If more than one such table exists, an error will be thrown and you will have to specify the exact name in `quant_name`.
`gene_to_transcript`	Table with columns `transcript_id` and all feature related columns that appear in the relevant fractions table. This is only relevant as a hack to to deal with the case where STAR includes in its transcriptome alignment transcripts on the opposite strand from where the RNA actually originated. This table will be used to filter out such transcript-feature combinations that should not exist.
`overwrite`	If TRUE and a fractions estimate output already exists that would possess the same metadata (features analyzed, populations analyzed, and fraction_design), then it will get overwritten with the new output. Else, it will be saved as a separate output with the same name + "_#" where "#" is a numerical ID to distinguish the similar outputs.
`TPM_min`	Minimum TPM for a transcript to be kept in analysis.
`count_min`	Minimum expected_count for a transcript to be kept in analysis.

Details

`EstimateIsoformFractions` expects as input a "fractions" table with estimates for transcript equivalence class (TEC) fraction news. A transcript equivalence class is the set of transcript isoforms with which a sequencing read is compatible. fastq2EZbakR is able to assign reads to these equivalence classes so that EZbakR can estimate the fraction of reads in each TEC that are from labeled RNA.

`EstimateIsoformFractions` estimates transcript isoform fraction news by fitting a linear mixing model to the TEC fraction new estimates + transcript isoform abundance estimates. In other words, each TEC fraction new (data) is modeled as a weighted average of transcript isoform fraction news (parameters to estimate), with the weights determined by the relative abundances of the transcript isoforms in the TEC (data). The TEC fraction new is modeled as a Beta distribution with mean equal to the weighted transcript isoform fraction new average and variance related to the number of reads in the TEC.

Transcript isoforms with estimated TPMs less than with an estimated TPM greater than `TPM_min` (1 by default) or an estimated number of read counts less than `count_min` (10 by default) are removed from TECs and will not have their fraction news estimated.

Value

An `EZbakRData` object with an additional table under the "fractions" list. Has the same form as the output of `EstimateFractions()`, and will have the feature column "transcript_id".

Examples

```
# Load dependencies
library(dplyr)

# Simulates a single sample worth of data
simdata_iso <- SimulateIsoforms(nfeatures = 100)

# We have to manually create the metadf in this case
metadf <- tibble(sample = 'sampleA',
                     tl = 4,
                     condition = 'A')

ezbdo <- EZbakRData(simdata_iso$cB,
                    metadf)

ezbdo <- EstimateFractions(ezbdo)

### Hack in the true, simulated isoform levels
reads <- simdata_iso$ground_truth %>%
  dplyr::select(transcript_id, true_count, true_TPM) %>%
  dplyr::mutate(sample = 'sampleA',
                effective_length = 10000) %>%
  dplyr::rename(expected_count = true_count,
                TPM = true_TPM)

# Name of table needs to have "isoform_quant" in it
ezbdo[['readcounts']][['simulated_isoform_quant']] <- reads

### Perform deconvolution
ezbdo <- EstimateIsoformFractions(ezbdo)
```

`EstimateKinetics` *Estimate kinetic parameters*

Description

Simple kinetic parameter (kdeg and ksyn) estimation using fraction estimates from `EstimateFractions()`. Several slightly different kinetic parameter inference strategies are implemented.

Usage

```
EstimateKinetics(
  obj,
  strategy = c("standard", "tilac", "NSS", "shortfeed", "pulse-chase"),
  features = NULL,
  populations = NULL,
  fraction_design = NULL,
  repeatID = NULL,
  exactMatch = TRUE,
  grouping_factor = NULL,
  reference_factor = NULL,
  character_limit = 20,
  feature_lengths = NULL,
  exclude_pulse_estimates = TRUE,
  scale_factors = NULL,
  overwrite = TRUE
)
```

Arguments

`obj`	An `EZbakRFractions` object, which is an `EZbakRData` object on which `EstimateFractions()` has been run.
`strategy`	Kinetic parameter estimation strategy. Options include: • standard: Estimate a single new read and old read mutation rate for each sample. This is done via a binomial mixture model aggregating over • NSS: Use strategy similar to that presented in Narain et al., 2021 that assumes you have data which provides a reference for how much RNA was present at the start of each labeling. In this case, `grouping_factor` and `reference_factor` must also be set. • shortfeed: Estimate kinetic parameters assuming no degradation of labeled RNA, most appropriate if the metabolic label feed time is much shorter than the average half-life of an RNA in your system. • tilac: Estimate TILAC-ratio as described in Courvan et al., 2022. • pulse-chase: Estimate kdeg for a pulse-chase experiment. By default kdeg will be estimated for each time point at which label was present. This includes any pulse-only samples, as well as all samples including a chase after the pulse. If you don't want to include the estimates from the pulse-only samples in the final output, set `exclude_pulse_estimates` to `TRUE`. Pulse-chases are suboptimal for a number of experimental reasons, so we urge users to avoid performing this kind of experiment whenever possible (favoring instead a pulse-label design). One of the challenges of analyzing pulse-chase data is that the fraction labeled after the pulse must be compared to that after each chase. Due to a number of technical reasons, it is possible for the estimated labeling after the chase to be **higher* than that after the pulse. It is impossible to estimate kinetic parameters in this case. Thus, when this arises, a conservative kdeg estimate is imputed, equal to -log(1 - 1/(n+2))/tchase, which is the kdeg estimate you would get if you had no detected reads from labeled RNA and an uninformative prior on the fraction new (i.e., the estimated fraction new is (number of reads from labeled RNA + 1) / (number of reads + 2)).
`features`	Character vector of the set of features you want to stratify reads by and estimate proportions of each RNA population. The default of "all" will use all feature columns in the `obj`'s cB.
`populations`	Mutational populations that were analyzed to generate the fractions table to use. For example, this would be "TC" for a standard s4U-based nucleotide recoding experiment.

`fraction_design`
: "Design matrix" specifying which RNA populations exist in your samples. By default, this will be created automatically and will assume that all combinations of the `mutrate_populations` you have requested to analyze are present in your data. If this is not the case for your data, then you will have to create one manually. See docs for `EstimateFractions` (run ?EstimateFractions()) for more details.

`repeatID`
: If multiple `fractions` tables exist with the same metadata, then this is the numerical index by which they are distinguished.

`exactMatch`
: If TRUE, then `features` and `populations` have to exactly match those for a given fractions table for that table to be used. Means that you can't specify a subset of features or populations by default, since this is TRUE by default.

`grouping_factor`
: Which sample-detail columns in the metadf should be used to group samples by for calculating the average RPM (`strategy = "NSS"`) or pulse fraction labeled (`strategy = "pulse-chase"`) at a particular time point? Only relevant if `strategy = "NSS"` or `strategy = "pulse-chase"`.

`reference_factor`
: Which sample-detail column in the metafd should be used to determine which group of samples provide information about the starting levels of RNA for each sample? This should have values that match those in `grouping_factor`. #' Only relevant if `strategy = "NSS"`.

`character_limit`
: Maximum number of characters for naming out fractions output. EZbakR will try to name this as a "_" separated character vector of all of the features analyzed. If this name is greater than `character_limit`, then it will default to "fraction#", where "#" represents a simple numerical ID for the table.

`feature_lengths`
: Table of effective lengths for each feature combination in your data. For example, if your analysis includes features named GF and XF, this should be a data frame with columns GF, XF, and length.

`exclude_pulse_estimates`
: If `strategy = "pulse-chase"`, would you like to exclude the pulse-only kdeg and ksyn estimates from the final output? This is a good idea if you used a very long pulse with the goal of labeling close to 100% of all RNA.

`scale_factors`
: Data frame with two columns, "sample" and "scale_factor". sample should be the sample name, and scale_factor is the factor which read counts in that sample should be **divided** to get the normalized read counts. This should match the convention from DESeq2 and thus you can provide, for example, scale factors derived from DESeq2 for this. By default, EZbakR uses DESeq2's median of ratios normalization method, but specifying `scale_factors` is useful when you have spike-ins.

`overwrite`
: If TRUE and a fractions estimate output already exists that would possess the same metadata (features analyzed, populations analyzed, and fraction_design), then it will get overwritten with the new output. Else, it will be saved as a separate output with the same name + "_#" where "#" is a numerical ID to distinguish the similar outputs.

Details

`EstimateKinetics()` estimates the kinetics and RNA synthesis and degradation from standard single-label NR-seq data. It also technically supports analyses of the dual-label TILAC experiment, but that functionality is not as actively tested or maintained as the more standard analyses.

`EstimateKinetics()` assumes a simple linear ODE model of RNA dynamics:

$$\frac{\mathrm{dR}}{\mathrm{dt}} = k_{\mathrm{syn}} - k_{\mathrm{deg}} * \mathrm{R}$$

where:

- R = Amount of RNA
- k_{syn} = Synthesis rate constant
- k_{deg} = Degradation rate constant and for which the general solution is:

  $$\mathrm{R(t)} = \mathrm{R(0)}e^{-k_{\mathrm{deg}}\mathrm{t}} + (1 - \frac{k_{\mathrm{syn}}}{k_{\mathrm{deg}}})e^{-k_{\mathrm{deg}}\mathrm{t}}$$

  where R(0) is the initial RNA level.

The default kinetic parameter estimation strategy (`strategy == "standard"`) assumes that for labeled RNA (or more precisely RNA synthesized in the presence of label) $\mathrm{R(0)} = 0$. Thus, it assumes a pulse-label rather than pulse-chase experimental design (I've written several places, like here and here for example, about why pulse-label designs are superior to pulse-chases in almost all settings).

For unlabeled RNA, it assumes that $\mathrm{R(0)} = \frac{k_{\mathrm{syn}}}{k_{\mathrm{deg}}}$. This is known as the steady-state assumption and is the key assumption of this method. More broadly, assuming steady-state means that this method assumes that RNA levels from a given feature are constant for the entire metabolic labeling duration. As EZbakR is designed for analyses of bulk NR-seq data, "constant" means that the average RNA levels across all profiled cells is constant (thus asynchronous populations of dividing cells still count as steady-state, even if the RNA expression landscapes in individual cells are quite dynamic). This assumption may be violated in cases where labeling coincides with or closely follows a perturbation (e.g., drug treatment). When the steady-state assumption hold, there is a simple relationship between the fraction of reads from labeled RNA (θ) and the turnover kinetics of the RNA:

$$\theta = 1 - e^{-k_{\mathrm{deg}}t}$$

The power in this approach is thus that turnover kinetics are estimated from an "internally normalized" quantity: θ (termed the "fraction new", "fraction labeled", "fraction high mutation content", or new-to-total ratio (NTR), depending on where you look or who you ask). "Internally normalized" means that a normalization scale factor does not need to be estimated in order to accurately infer turnover kinetics. θ represents a ratio of read counts from the same feature in the same library, thus any scale factor would appear in both the numerator and denominator of this estimate and cancel out.

When this assumption is valid, the `strategy = "NSS"` approach may be preferable. This approach relies on the same model of RNA dynamics, but now assumes that the initial RNA levels (R(0) for the unlabeled RNA) are not at the steady-state levels dictated by the current synthesis and turnover kinetics. The idea for this strategy was first presented in Narain et al. 2021. To run this approach, you need to be able to estimate the initial RNA levels of each label-fed sample, as the fraction of reads from labeled RNA will no longer. only reflect the turnover kinetics (as the old RNA is assumed to potentially not have reached the new steady-state levels yet). This means that for each label-fed sample, you need to have a sample whose assayed RNA population represents the starting RNA population for the label-fed sample. EZbakR will use these reference samples to infer R(0) for unlabeled RNAs and estimate turnover kinetics from the ratio of this quantity to the remaining unlabeled RNA levels after labeling:

$$\theta_{\text{NSS}} = \frac{\text{R(tl)}}{\text{R(0)}}$$

While I really like the idea of this strategy, it is not without some severe limitations. For one, the major benefit of NR-seq, internally normalized estimation of turnover kinetics, is out the window. Thus, read counts need to be normalized between the relevant reference and label-fed sample pairs. In addition, the variance patterns of this ratio are somewhat unfortunate. Its variance is incredibly high for more stable RNAs, severely limiting the effective dynamic range of this approach relative to steady-state analyses. I continue to work to refine EZbakR's implementation of this approach, but for now I urge caution in its use and interpretation. See my discussion of an alternative approach for navigating non-steady-state data here.

As an aside, you may wonder how this strategy deals with dynamic regulation of synthesis and degradation rate constants *during labeling*. To answer this, you have to realize that the duration of metabolic labeling represents an integration time over which the best we can do is assess average kinetics. Thus, if rate constants are changing during the labeling, this strategy can still be thought of as providing a an estimate of the time averaged synthesis and turnover kinetics during the label time.

Eventually, I will get around to implementing pulse-chase support in `EstimateKinetics()`. I haven't yet because I don't like pulse-chase experiments and think they are way more popular than they should be for purely historical reasons. But lots of people keep doing pulse-chase NR-seq so c'est la vie.

Value

`EZbakRKinetics` object, which is just an `EZbakRData` object with a "kinetics" slot. This includes tables of kinetic parameter estimates for each feature in each sample for which kinetic parameters can be estimated.

Examples

```
# Simulate data to analyze
simdata <- SimulateOneRep(30)

# Create EZbakR input
metadf <- data.frame(sample = "sampleA", tl = 2)
ezbdo <- EZbakRData(simdata$cB, metadf)

# Estimate Fractions
ezbdo <- EstimateFractions(ezbdo)

# Estimate Kinetics
ezbdo <- EstimateKinetics(ezbdo)
```

EstimateMutRates *Estimate mutation rates*

Description

Two component mixture models are fit to all data to estimate global high and low mutation rates for all samples. Estimation of these mutation rates are regularized through the use of weakly informative priors whose parameters can be altered using the arguments defined below.

Usage

```
EstimateMutRates(
  obj,
  populations = "all",
  pnew_prior_mean = -2.94,
  pnew_prior_sd = 0.3,
  pold_prior_mean = -6.5,
  pold_prior_sd = 0.5,
  pold_est = NULL,
  pold_from_nolabel = FALSE,
  grouping_factors = NULL
)

## S3 method for class 'EZbakRData'
EstimateMutRates(
  obj,
  populations = "all",
  pnew_prior_mean = -2.94,
  pnew_prior_sd = 0.3,
  pold_prior_mean = -6.5,
  pold_prior_sd = 0.5,
  pold_est = NULL,
  pold_from_nolabel = FALSE,
  grouping_factors = NULL
)

## S3 method for class 'EZbakRArrowData'
EstimateMutRates(
  obj,
  populations = "all",
  pnew_prior_mean = -2.94,
  pnew_prior_sd = 0.3,
  pold_prior_mean = -6.5,
  pold_prior_sd = 0.5,
  pold_est = NULL,
  pold_from_nolabel = FALSE,
  grouping_factors = NULL
)
```

Arguments

`obj`	An `EZbakRData` or `EZbakRArrowData` object
`populations`	Character vector of the set of mutational populations that you want to infer the fractions of. For example, say your cB file contains columns tracking T-to-C and G-to-A
`pnew_prior_mean`	logit-Normal mean for logit(pnew) prior.
`pnew_prior_sd`	logit-Normal sd for logit(pnew) prior.
`pold_prior_mean`	logit-Normal mean for logit(pold) prior.
`pold_prior_sd`	logit-Normal sd for logit(pold) prior.
`pold_est`	Background mutation rate estimates if you have them. Can either be a single number applied to all samples or a named vector of values, where the names should be sample names.
`pold_from_nolabel`	Fix background mutation rate estimate to mutation rates seen in -label samples. By default, a single background rate is used for all samples, inferred from the average mutation rate across all -label samples. The `grouping_factors` argument can be specified to use certain -label samples to infer background mutation rates for certain sets of +label samples.
`grouping_factors`	If `pold_from_nolabel` is TRUE, then `grouping_factors` will specify the sample-detail columns in the metadf that should be used to group -label samples by. Average mutation rates in each group of -label samples will be used as the background mutation rate estimate in +label samples with the same values for the relevant metadf columns.

Value

`EZbakRData` object with an added `mutrates` slot containing estimated high and low mutation rates for each mutation type modeled.

Methods (by class)

- `EstimateMutRates(EZbakRData)`: Method for class **EZbakRData** Estimates mutation rates using a fully in memory object.
- `EstimateMutRates(EZbakRArrowData)`: Method for class **EZbakRArrowData** Estimate mutation rates using a partially on-disk object.

Examples

```
# Simulate data to analyze
simdata <- SimulateOneRep(30)
```

```
# Create EZbakR input
metadf <- data.frame(sample = "sampleA", tl = 2)
ezbdo <- EZbakRData(simdata$cB, metadf)

# Estimate mutation rates
mutrates <- EstimateMutRates(ezbdo)
```

`example_cB` *Example cB table*

Description

An example `cB` table used to create an `EZbakRData` object. This cB table is a subset of a cB file from the DCP2 KO dataset published in Luo et al., 2020. The original file is large (69 MB), so the example cB file has been downsampled and contains only a subset of reads from chromosome 21.

Usage

```
example_cB
```

Format

`example_cB`**:**

A tibble with 10,000 rows and 7 columns:

sample Sample name

rname Chromosome name

GF Gene name for reads aligning to any region of a gene

XF Gene name for reads aligning to exclusively exonic regions of a gene

TC Number of T-to-C mutations

nT Number of Ts

n Number of reads with same value for the first 6 columns

References

Luo et al. (2020) Biochemistry. 59(42), 4121-4142

`example_metadf` *Example metadf*

Description

An example `metadf` table used to create an `EZbakRData` object. This metadf describes the DCP2 KO dataset published in Luo et al., 2020.

Usage

```
example_metadf
```

Format

`example_metadf`:

A tibble with 6 rows and 3 columns

sample Sample name

tl Metabolic label feed time

genotype Whether sample was collected from WT or DCP2 KO cells

References

Luo et al. (2020) Biochemistry. 59(42), 4121-4142

`EZbakRArrowData` `EZbakRArrowData` *object helper function for users*

Description

`EZbakRArrowData` creates an object of class `EZbakRArrowData` and checks the validity of the provided input.

Usage

```
EZbakRArrowData(cBds, metadf)
```

Arguments

`cBds` ArrowDataset with the following fields:

- sample: Name given to particular sample from which data was collected.
- mutational counts: Integers corresponding to the number of a particular mutation seen in a sequencing read. The following column names are allowed:
  - TC: Number of Thymine-to-Cytosine mutations
  - TA: Number of Thymine-to-Adenine mutations

  - TG: Number of Thymine-to-Guanine mutations
  - CT: Number of Cytosine-to-Thymine mutations
  - CA: Number of Cytosine-to-Adenine mutations
  - CG: Number of Cytosine-to-Guanine mutations
  - CU: Number of Cytosine-to-Uridine mutations
  - AT: Number of Adenine-to-Thymine mutations
  - AC: Number of Adenine-to-Cytosine mutations
  - AG: Number of Adenine-to-Guanine mutations
  - AU: Number of Adenine-to-Uridine mutations
  - GT: Number of Guanine-to-Thymine mutations
  - GC: Number of Guanine-to-Cytosine mutations
  - GA: Number of Guanine-to-Adenine mutations
  - GU: Number of Guanine-to-Uridine mutations
  - TN: Number of Thymine-to-Adenine/Cytosine/Guanine mutations
  - CN: Number of Cytosine-to-Adenine/Thymine/Guanine/Uridine mutations
  - AN: Number of Adenine-to-Thymine/Cytosine/Guanine/Uridine mutations
  - GN: Number of Guanine-to-Adenine/Cytosine/Thymine/Uridine mutations
  - UN: Number of Uridine-to-Adenine/Cytosine/Guanine mutations
  - NT: Number of Adenine/Cytosine/Guanine-to-Thymine mutations
  - NC: Number of Adenine/Thymine/Guanine/Uridine-to-Cytosine mutations
  - NtoA: Number of Thymine/Cytosine/Guanine/Uridine-to-Adenine mutations. (Naming convention changed because NA taken)
  - NU: Number of Cytosine/Guanine/Adenine-to-Uridine mutations.
  - NN: Number of any kind of mutation
- base nucleotide count: Integers corresponding to the number of instances of a particular type of nucleotide whose mutations are tracked in a corresponding mutation count column. The following column names are allowed:
  - nT: Number of Thymines
  - nG: Number of Guanines
  - nA: Number of Adenines
  - nC: Number of Cytosines
  - nU: Number of Uridines
  - nN: number of any kind of nucleotide
- features: Any columns that cannot be interpreted as a mutation count or base nucleotide count (and that aren't named `sample` or `n`) will be interpreted as an ID for a genomic "feature" from which a read originated. Common examples of features and typical column names for said features include:
  - Genes; common column names: gene, gene_id, gene_name, GF

  - Genes-exonic; common column names: gene_exon, gene_id_exon, gene_name_exon, XF
  - Transcripts; common column names: transcripts, TF
  - Exonic bins; common column names: exonic_bins, EF, EB
  - Exons; common column names: exons, exon_ids

  In some cases, a read will often map to multiple features (e.g., exons). Many functions in bakR expect each of the feature IDs in these cases to be separated by `+`. For example, if a read overlaps with two exons, with IDs exon_1 and exon_2, then the corresponding entry in a column of exonic assignments would be "exon_1+exon_2". The default expectation can be overwritten though and is thus not strictly enforced.
- n: Number of reads with identical values for all other columns.

`metadf` Data frame detailing various aspects of each of the samples included in the cBds. This includes:

- `sample`: The sample ID, which should correspond to a sample ID in the provided cBds.
- `tl`: Metabolic label time. There are several edge cases to be aware of:
  - If more than one metabolic label was used in the set of samples described by the metadf (e.g., s4U and s6G were used), then the `tl` column should be replaced by `tl_<muttype>`, where `<muttype>` represents the corresponding mutation type count column in the cBds that the label whose incubation time will be listed in this column. For example, if feeding with s4U in some samples and s6G in others, then performing standard nucleotide recoding chemistry, you will include `tl_TC` and `tl_GA` columns corresponding to the s4U and s6G label times, respectively.
  - If a pulse-chase experimental design was used (!!this is strongly discouraged unless you have a legitimate reason to prefer this design to a pulse-label design!!), then you should have columns named `tpulse` and `tchase`, corresponding to the pulse and chase times respectively. The same `<muttype>` convention should be used in the case of multi-label pulse-chase designs.
- sample characteristics: The remaining columns can be named whatever you like and should include distinguishing features of groups of samples. Common columns might include:
  - `treatment`: The experimental treatment applied to a set of samples. This could represent things like genetic knockouts or knockdowns, drug treatments, etc.
  - `batch`: An ID for sets of samples that were collected and/or processed together. Useful for regressing out technical batch effects

Value

An EZbakRArrowData object. This is simply a list of the provide `cBds` and `metadf` with class `EZbakRArrowData`

Examples

```
# Load dependency
library(arrow)
simdata <- EZSimulate(30)

# Create directory to write dataset to
outdir <- tempdir()
dataset_dir <- file.path(outdir, "arrow_dataset")

# Create dataset
write_dataset(
  simdata$cB,
  path = dataset_dir,
  format = "parquet",
  partitioning = "sample"
)

# Create EZbakRArrowData object
ds <- open_dataset(dataset_dir)
ezbdo <- EZbakRArrowData(ds,
                         simdata$metadf)
```

EZbakRData — `EZbakRData` *object helper function for users*

Description

`EZbakRData` creates an object of class `EZbakRData` and checks the validity of the provided input.

Usage

```
EZbakRData(cB, metadf)
```

Arguments

`cB` Data frame with the following columns:

- sample: Name given to particular sample from which data was collected.
- mutational counts: Integers corresponding to the number of a particular mutation seen in a sequencing read. The following column names are allowed:
  - TC: Number of Thymine-to-Cytosine mutations
  - TA: Number of Thymine-to-Adenine mutations
  - TG: Number of Thymine-to-Guanine mutations
  - CT: Number of Cytosine-to-Thymine mutations
  - CA: Number of Cytosine-to-Adenine mutations
  - CG: Number of Cytosine-to-Guanine mutations

  - CU: Number of Cytosine-to-Uridine mutations
  - AT: Number of Adenine-to-Thymine mutations
  - AC: Number of Adenine-to-Cytosine mutations
  - AG: Number of Adenine-to-Guanine mutations
  - AU: Number of Adenine-to-Uridine mutations
  - GT: Number of Guanine-to-Thymine mutations
  - GC: Number of Guanine-to-Cytosine mutations
  - GA: Number of Guanine-to-Adenine mutations
  - GU: Number of Guanine-to-Uridine mutations
  - TN: Number of Thymine-to-Adenine/Cytosine/Guanine mutations
  - CN: Number of Cytosine-to-Adenine/Thymine/Guanine/Uridine mutations
  - AN: Number of Adenine-to-Thymine/Cytosine/Guanine/Uridine mutations
  - GN: Number of Guanine-to-Adenine/Cytosine/Thymine/Uridine mutations
  - UN: Number of Uridine-to-Adenine/Cytosine/Guanine mutations
  - NT: Number of Adenine/Cytosine/Guanine-to-Thymine mutations
  - NC: Number of Adenine/Thymine/Guanine/Uridine-to-Cytosine mutations
  - NtoA: Number of Thymine/Cytosine/Guanine/Uridine-to-Adenine mutations. (Naming convention changed because NA taken)
  - NU: Number of Cytosine/Guanine/Adenine-to-Uridine mutations.
  - NN: Number of any kind of mutation
- base nucleotide count: Integers corresponding to the number of instances of a particular type of nucleotide whose mutations are tracked in a corresponding mutation count column. The following column names are allowed:
  - nT: Number of Thymines
  - nG: Number of Guanines
  - nA: Number of Adenines
  - nC: Number of Cytosines
  - nU: Number of Uridines
  - nN: number of any kind of nucleotide
- features: Any columns that cannot be interpreted as a mutation count or base nucleotide count (and that aren't named `sample` or `n`) will be interpreted as an ID for a genomic "feature" from which a read originated. Common examples of features and typical column names for said features include:
  - Genes; common column names: gene, gene_id, gene_name, GF
  - Genes-exonic; common column names: gene_exon, gene_id_exon, gene_name_exon, XF
  - Transcripts; common column names: transcripts, TF
  - Exonic bins; common column names: exonic_bins, EF, EB
  - Exons; common column names: exons, exon_ids

In some cases, a read will often map to multiple features (e.g., exons). Many functions in bakR expect each of the feature IDs in these cases to be separated by `+`. For example, if a read overlaps with two exons, with IDs exon_1 and exon_2, then the corresponding entry in a column of exonic assignments would be "exon_1+exon_2". The default expectation can be overwritten though and is thus not strictly enforced.

- n: Number of reads with identical values for all other columns.

`metadf` Data frame detailing various aspects of each of the samples included in the cB. This includes:

- `sample`: The sample ID, which should correspond to a sample ID in the provided cB.
- `tl`: Metabolic label time. There are several edge cases to be aware of:
  - If more than one metabolic label was used in the set of samples described by the metadf (e.g., s4U and s6G were used), then the `tl` column should be replaced by `tl_<muttype>`, where `<muttype>` represents the corresponding mutation type count column in the cB that the label whose incubation time will be listed in this column. For example, if feeding with s4U in some samples and s6G in others, then performing standard nucleotide recoding chemistry, you will include `tl_TC` and `tl_GA` columns corresponding to the s4U and s6G label times, respectively.
  - If a pulse-chase experimental design was used (!!this is strongly discouraged unless you have a legitimate reason to prefer this design to a pulse-label design!!), then you should have columns named `tpulse` and `tchase`, corresponding to the pulse and chase times respectively. The same `_<muttype>` convention should be used in the case of multi-label pulse-chase designs.
- sample characteristics: The remaining columns can be named whatever you like and should include distinguishing features of groups of samples. Common columns might include:
  - `treatment`: The experimental treatment applied to a set of samples. This could represent things like genetic knockouts or knockdowns, drug treatments, etc.
  - `batch`: An ID for sets of samples that were collected and/or processed together. Useful for regressing out technical batch effects

Value

An EZbakRData object. This is simply a list of the provide `cB` and `metadf` with class `EZbakRData`

Examples

```
# Simulate data
simdata <- EZSimulate(30)

# Create EZbakRData object
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)
```

`EZbakRFractions`   `EZbakRFractions` *helper function for users*

Description

`EZbakRFractions` creates an object of class `EZbakRFractions` and checks the validity of the provided input.

Usage

```
EZbakRFractions(fractions, metadf, name = NULL, character_limit = 20)
```

Arguments

`fractions` Data frame with the following columns:

- sample: Name given to particular sample from which data was collected.
- estimates of population fractions: These columns refer to the estimate for the fraction of reads coming from a particular mutational population. For example, in a standard NR-seq experiment, you should have one column named `fraction_highTC`. This refers to the fraction of RNA inferred to have a high T-to-C mutation rate (e.g., the newly synthesized RNA in a pulse-labeling NR-seq experiment). If you estimated the fractions of more than 2 mutation types (e.g., T-to-C and G-to-A), then you need to explicitly list all fractions of interest estimated. For example, in a TILAC experiment, this would be `fraction_highTC_lowGA`, `fraction_lowTC_highGA`, and `fraction_lowTC_lowGA`.
- n: Number of reads assigned to a given feature in a given sample.
- features: Any columns that cannot be interpreted as an estimate of population fractions (and that aren't named `sample` or `n`) will be interpreted as an ID for a genomic "feature" from which a read originated. Common examples of features and typical column names for said features include:
  - Genes; common column names: gene, gene_id, gene_name, GF
  - Genes-exonic; common column names: gene_exon, gene_id_exon, gene_name_exon, XF
  - Transcripts; common column names: transcripts, TF
  - Exonic bins; common column names: exonic_bins, EF, EB
  - Exons; common column names: exons, exon_ids

  In some cases, a read will often map to multiple features (e.g., exons). Many functions in bakR expect each of the feature IDs in these cases to be separated by `+`. For example, if a read overlaps with two exons, with IDs exon_1 and exon_2, then the corresponding entry in a column of exonic assignments would be "exon_1+exon_2". The default expectation can be overwritten though and is thus not strictly enforced.
- n: Number of reads with identical values for all other columns.

`metadf` Data frame detailing various aspects of each of the samples included in the fractions data frame. This includes:

- `sample`: The sample ID, which should correspond to a sample ID in the provided fractions data frame.
- `tl`: Metabolic label time. There are several edge cases to be aware of:
  - If more than one metabolic label was used in the set of samples described by the metadf (e.g., s4U and s6G were used), then the `tl` column should be replaced by `tl_<muttype>`, where `<muttype>` represents the corresponding mutation type referenced in the fractions that the label whose incubation time will be listed in this column. For example, if feeding with s4U in some samples and s6G in others, then performing standard nucleotide recoding chemistry, you will include `tl_TC` and `tl_GA` columns corresponding to the s4U and s6G label times, respectively.
  - If a pulse-chase experimental design was used (!!this is strongly discouraged unless you have a legitimate reason to prefer this design to a pulse-label design!!), then you should have columns named `tpulse` and `tchase`, corresponding to the pulse and chase times respectively. The same `_<muttype>` convention should be used in the case of multi-label pulse-chase designs.
- sample characteristics: The remaining columns can be named whatever you like and should include distinguishing features of groups of samples. Common columns might include:
  - `treatment`: The experimental treatment applied to a set of samples. This could represent things like genetic knockouts or knockdowns, drug treatments, etc.
  - `batch`: An ID for sets of samples that were collected and/or processed together. Useful for regressing out technical batch effects

`name` Optional; name to give to fractions table.

`character_limit`

Maximum number of characters for naming out fractions output. EZbakR will try to name this as a "_" separated character vector of all of the features analyzed. If this name is greater than `character_limit`, then it will default to "fraction#", where "#" represents a simple numerical ID for the table.

Value

An EZbakRFractions object. This is simply a list of the provide `fractions` and `metadf` with class `EZbakRFractions`

Examples

```
# Simulate data
simdata <- EZSimulate(30)

# Get fractions table by estimating (for demonstration)
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)
ezbdo <- EstimateFractions(ezbdo)
```

```
fxns <- EZget(ezbdo, type = "fractions")

# Create EZbakRFractions object
ezbfo <- EZbakRFractions(fxns, simdata$metadf)
```

`EZbakRKinetics` `EZbakRKinetics` *helper function for users*

Description

`EZbakRKinetics` creates an object of class `EZbakRKinetics` and checks the validity of the provided input.

Usage

```
EZbakRKinetics(kinetics, metadf, features, name = NULL, character_limit = 20)
```

Arguments

`kinetics` Data frame with the following columns:

- sample: Name given to particular sample from which data was collected.
- features: Any columns that cannot be interpreted as a mutation count or base nucleotide count (and that aren't named `sample` or `n`) will be interpreted as an ID for a genomic "feature" from which a read originated. Common examples of features and typical column names for said features include:
  - Genes; common column names: gene, gene_id, gene_name, GF
  - Genes-exonic; common column names: gene_exon, gene_id_exon, gene_name_exon, XF
  - Transcripts; common column names: transcripts, TF
  - Exonic bins; common column names: exonic_bins, EF, EB
  - Exons; common column names: exons, exon_ids

  In some cases, a read will often map to multiple features (e.g., exons). Many functions in bakR expect each of the feature IDs in these cases to be separated by `+`. For example, if a read overlaps with two exons, with IDs exon_1 and exon_2, then the corresponding entry in a column of exonic assignments would be "exon_1+exon_2". The default expectation can be overwritten though and is thus not strictly enforced.
- n: Number of reads with identical values for all other columns.
- kinetic parameter estimates: These can be named whatever you would like as long as they do not start with the string "se_". This should be reserved for kinetic parameter uncertainties, if provided.
- kinetic parameter uncertainties: Uncertainty in your kinetic parameter estimates. These should be named "se_" followed by the kinetic parameter as its name appears in the relevant column name of the `kinetics` table.

`metadf` Data frame detailing various aspects of each of the samples included in the kinetics data frame. This includes:

- `sample`: The sample ID, which should correspond to a sample ID in the provided kinetics data frame.
- `tl`: Metabolic label time. There are several edge cases to be aware of:
  - If more than one metabolic label was used in the set of samples described by the metadf (e.g., s4U and s6G were used), then the `tl` column should be replaced by `tl_<muttype>`, where `<muttype>` represents the corresponding mutation type referenced in the fractions that the label whose incubation time will be listed in this column. For example, if feeding with s4U in some samples and s6G in others, then performing standard nucleotide recoding chemistry, you will include `tl_TC` and `tl_GA` columns corresponding to the s4U and s6G label times, respectively.
  - If a pulse-chase experimental design was used (!!this is strongly discouraged unless you have a legitimate reason to prefer this design to a pulse-label design!!), then you should have columns named `tpulse` and `tchase`, corresponding to the pulse and chase times respectively. The same `_<muttype>` convention should be used in the case of multi-label pulse-chase designs.
- sample characteristics: The remaining columns can be named whatever you like and should include distinguishing features of groups of samples. Common columns might include:
  - `treatment`: The experimental treatment applied to a set of samples. This could represent things like genetic knockouts or knockdowns, drug treatments, etc.
  - `batch`: An ID for sets of samples that were collected and/or processed together. Useful for regressing out technical batch effects
- `assay`: This optional column should include a string that describes the type of experiment that was done so as to influence how EZbakR analyzes and interprets the data from those samples. Possible values for `assay` currently include:
  - standard: Refers to the "standard" nucleotide recoding RNA-seq methods (e.g., TimeLapse-seq, SLAM-seq, TUC-seq, etc.), in which cells are fed with a single metabolic label, RNA is extracted and sequenced, and mutations of a particular type are counted
  - STL: Refers to Start-TimeLapse-seq, a method combining Start-seq (developed by Karen Adelman's lab) with TimeLapse-seq. Used to infer the kinetics of transcription initiation and promoter-proximal pause site departure.
  - TT: Refers to Transient-Transcriptome NR-seq, a method combining TT-seq (developed by Patrick Cramer's lab) with NR-seq. TT-seq involves biochemically enriching for labeled RNA. By combining this method with nucleotide recoding chemistry (as was first done by the Simon lab with TT-TimeLapse-seq and has since been done with SLAM chemistry, often referred to as TTchem-seq), it is possible to bioinformatically filter out reads coming from unlabeled RNA background.

  - TILAC: Refers to TILAC, a method developed by the Simon lab to achieve spike-in free normalization of RNA-seq data through the use of a dual labeling approach inspired by the proteomic method SILAC.
  - subcellular: Refers to techniques such as subcellular TimeLapse-seq (developed by Stirling Churchman's lab) which combine subcellular fractionation with NR-seq to infer additional kinetic parameters.
  - sc: Refers to single-cell RNA-seq implementations of NR-seq.

`features`	Features tracked in `kinetics` data frame. Needs to be specified explicitly as it cannot be automatically inferred.
`name`	Optional; name to give to fractions table.
`character_limit`	If name is chosen automatically, limit on the number of characters in said name. If default name yields a string longer than this, then kinetics table will be named `kinetics1`

Value

An EZbakRKinetics object. This is simply a list of the provided `kinetics` and `metadf` with class `EZbakRKinetics`

Examples

```
# Simulate data
simdata <- EZSimulate(30)

# Get kinetics table by estimating (for demonstration)
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)
ezbdo <- EstimateFractions(ezbdo)
ezbdo <- EstimateKinetics(ezbdo)
kinetics <- EZget(ezbdo, type = "kinetics")

# Create EZbakRKinetics object
ezbko <- EZbakRKinetics(kinetics, simdata$metadf, features = "feature")
```

`EZDynamics` *Generalized dynamical systems modeling*

Description

`EZDynamics()` estimates parameters of a user-specified dynamical systems model. The dynamical system model is specified through an adjacency matrix, which is an NxN matrix described below (see `graph` documentation). Modeling can either be done for species all assayed in each sample, or species that are assayed across a set of independent samples (e.g., subcellular fractionation involves assaying different species in different samples).

Usage

```
EZDynamics(
  obj,
  graph,
  sub_features,
  grouping_features,
  scale_factors = NULL,
  sample_feature = NULL,
  modeled_to_measured = NULL,
  parameter_names = paste0("logk", 1:max(graph)),
  unassigned_name = "__no_feature",
  type = "averages",
  prior_means = rep(-3, times = max(graph)),
  prior_sds = c(3, rep(1, times = max(graph) - 1)),
  avg_params_tokeep = NULL,
  avg_params_todrop = NULL,
  label_time_name = "tl",
  features = NULL,
  populations = NULL,
  fraction_design = NULL,
  parameter = NULL,
  repeatID = NULL,
  exactMatch = TRUE,
  feature_lengths = NULL,
  use_coverage = TRUE,
  normalization_repeatID = NULL,
  normalization_exactMatch = TRUE,
  species_to_sf = NULL,
  overwrite = TRUE
)
```

Arguments

`obj`	Currently must be an EZbakRData object on which `AverageAndRegularize` has been run. In the future, will also support (in case where all species are assayed in every sample) providing output of just `EstimateFractions()` as input, acting as a generalization of `EstimateKinetics()` in that case.
`graph`	An NxN adjacency matrix, where N represents the number of species being modeled. One of these species must be called "0" and represent the "no RNA" species. This is the species from which some species are synthesized (e.g., 0 -> P, means premature RNA is synthesized from no RNA), and the species to which some species are degraded (e.g., M -> 0 means mature RNA is converted to "no RNA" via degradation). The rows and columns of this matrix must be the names of all modled species, and rownames(graph) == colnames(graph). Entry i,j of the matrix is either 0 if species i cannot be converted into species j under your model, and an integer from 1:npars (where npars = total number of parameters to be estimated) if it can. For example, the model 0 -> P -> M -> 0 would have the `graph`: `matrix(c(0, 1, 0, 0, 0, 2, 3, 0, 0),`

`sub_features` Which feature columns distinguish between the different measured species? Note, the measured species need not have the same name, and may not be directly equivalent to, the modeled species. The relationship between the modeled species in `graph` and `sub_features` needs to be specified in `modeled_to_measured` if the names are not equivalent though.

`grouping_features`
Which features are the overarching feature assignments by which `sub_features` should be grouped? This will usually be the feature columns corresponding to full-gene assignments, as well as any higher order assignments (e.g., chromosome). A `sub_feature` can be included in `grouping_features` if it never has the value of `unassigned_name` ("__no_feature" by default). Only one `sub_feature` should ever fulfill this criterion though.

`scale_factors` Data frame mapping samples to factors by which to multiply read counts so as ensure proper normalization between different RNA populations. Only relevant if you are modeling relationships between distinct RNA populations, for example RNA from nuclear and cytoplasmic fractions. Will eventually be inferred automatically.

`sample_feature` If different samples involve assaying different species, this must be the name of the metadf column that distinguishes the different classes of samples. For example, if analyzing a subcellular fractionation dataset, you likely included a column in your metadf called "compartment". This would then be your `sample_feature`, assuming you ran `AverageAndRegularize()`, with a `mean_formula` that included compartment as a term.

`modeled_to_measured`
If `sub_features` is not identical to the non "0" species names in `graph`, then you must specify the relationship between `sub_features`, `sample_feature` (if specified), and the species in `graph`. This is done through a list of formulas, whose left side is a `sub_feature` and whose right side is a function of species in `graph`. If `sample_feature` is not specified, then `modeled_to_measured` should be a nameless list of formulae. If `sample_feature` is specified, then `modeled_to_measured` must be a named list of formulas where the names correspond to unique values of `sample_feature`. In this latter case, the elements, should be the mapping of measured features (`sub_features`) to modeled species (names in `graph` that aren't "0").

For example, if your model is 0 -> P -> M -> 0, where P stands for premature RNA and M stands for mature RNA, and you have a column called GF that corresponds to assignment anywhere in the gene, and XF that corresponds to assignment of exclusively exon mapping reads, then your `modeled_to_measured` should be `list(GF ~ P + M, XF ~ M)`..

As another example, if your model is 0 -> N -> C -> 0, where N stands for "nuclear RNA" and C stands for "cytoplasmic RNA", and your `sample_feature` takes on values of "nuclear", "cytoplasmic", and "total", and you have a single `sub_feature` called XF, then your `modeled_to_measured` shouild be `list(nuclear = GF ~ N, cytoplasmic = GF ~ C, total = GF ~ C + N)`. This is interpreted as meaning in groups of samples for which `sample_feature` == "nuclear", reads assigned to a GF are from the N species (nuclear RNA). When `sample_feature` == "cytoplasmic", reads assigned to GF correspond to the C species (cytoplas-

	mic RNA). When `sample_feature` == "total", reads assigned to GF correspond to a combination of N and C (nuclear and cytoplasmic RNA).
`parameter_names`	Vector of names you would like to give to the estimated parameters. ith element should correspond to name of parameter given the ID i in `graph`. By default, this is just ki, where i is this numerical index.
`unassigned_name`	What value will a `sub_feature` column have if a read was not assigned to said feature? "__no_feature" by default.
`type`	What type of table would you like to use? Currently only supports "averages", but will support "fractions" in the near future.
`prior_means`	Mean of log-Normal prior for kinetic parameters. Should be vector where ith value is mean for ith parameter, i = index in `graph`
`prior_sds`	Std. dev. of log-Normal prior for kinetic parameter. Should be vector where ith value is mean for ith parameter, i = index in `graph`.
`avg_params_tokeep`	Names of parameters in averages table that you would like to keep. Other parameters will be discarded. Don't include the prefixes "mean_", "sd_", or "coverage_"; these will be imputed automatically. In other words, this should be the base parameter names.
`avg_params_todrop`	Names of parameters in averages table that you would like to drop. Other parameters will be kept. Don't include the prefixes "mean_", "sd_", or "coverage_"; these will be imputed automatically. In other words, this should be the base parameter names.
`label_time_name`	Name of relevant label time column that will be found in the parameter names. Defaults to the standard "tl".
`features`	Character vector of the set of features you want to stratify reads by and estimate proportions of each RNA population. The default of "all" will use all feature columns in the `obj`'s cB.
`populations`	Mutational populations that were analyzed to generate the fractions table to use. For example, this would be "TC" for a standard s4U-based nucleotide recoding experiment.
`fraction_design`	"Design matrix" specifying which RNA populations exist in your samples. By default, this will be created automatically and will assume that all combinations of the `mutrate_populations` you have requested to analyze are present in your data. If this is not the case for your data, then you will have to create one manually. See docs for `EstimateFractions` (run ?EstimateFractions()) for more details.
`parameter`	Parameter to average across replicates of a given condition. Has to be `logit_fraction_high<muttype>`, where <muttype> is the type of mutation modeled in `EstimateFractions()` (e.g, TC) in this case.
`repeatID`	If multiple `fractions` tables exist with the same metadata, then this is the numerical index by which they are distinguished.

`exactMatch`	If TRUE, then `features` and `populations` have to exactly match those for a given fractions table for that table to be used. Means that you can't specify a subset of features or populations by default, since this is TRUE by default.
`feature_lengths`	Table of effective lengths for each feature combination in your data. For example, if your analysis includes features named GF and XF, this should be a data frame with columns GF, XF, and length.
`use_coverage`	If TRUE, normalized read counts will be used to inform kinetic parameter estimates. If FALSE, only fraction news will be used, which will leave some parameters (e.g., synthesis rate) unidentifiable, though has the advantage of avoiding the potential biases induced by normalization problems.
`normalization_repeatID`	For extracting the `fractions` table needed for normalization of multi-sample data. If multiple `fractions` tables exist with the same metadata, then this is the numerical index by which they are distinguished.
`normalization_exactMatch`	For extracting the `fractions` table needed for normalization of multi-sample data. If TRUE, then `features` and `populations` have to exactly match those for a given fractions table for that table to be used. Means that you can't specify a subset of features or populations by default, since this is TRUE by default.
`species_to_sf`	List mapping individual RNA species in `graph` to different sample_feature values (sf). This is relevant if you are modeling both pre- and mature RNA dynamics in subcellular fractionation data. EZbakR can usually automatically infer this, but if not, then you can manually specify this mapping. Should be a list with one element per unique sample_feature type. Each element should be a vector of the RNA species in `graph` that belong to that sample_feature type. For example, if you have whole cell, cytoplasmic, and nuclear fraction data, this should have one element called "cytoplasmic" and one element called "nuclear". The whole cell sample_feature is a sum of cytoplasmic and nuclear and thus does not apply. The "cytoplasmic" element of this list should be the set of RNA species that are present in the cytoplasm, e.g. cytoplasmic pre-RNA (maybe referred to in `graph` as CP) and cytoplasmic mature RNA (maybe referred to in `graph` as CM).
`overwrite`	If TRUE and a fractions estimate output already exists that would possess the same metadata (features analyzed, populations analyzed, and fraction_design), then it will get overwritten with the new output. Else, it will be saved as a separate output with the same name + "_#" where "#" is a numerical ID to distinguish the similar outputs.

Details

When running `AverageAndRegularize()` to produce input for `EZDynamics()`, you must set `parameter` to `logit_fraction_high<muttype>` (<muttype> = type of mutation modeled by `EstimateFractions()`, e.g., TC). If you have multiple distinct label times, you must also include the label time (`tl` of your `metadf`) in your regression formula. `EZDynamics()` models the logit(fraction high <muttype>), and this will depend on the label time (longer label time = higher fraction), which is why these two conditions must be met. If you only have a single label time though, `EZDynamics` will be able to impute this one value for all samples from your `metadf`. You can also include additional interaction terms in your `AverageAndRegularize()` model for different experimental conditions in which experiments were conducted, so that inferred kinetic parameters can be compared across these conditions. Currently, more complex modeling beyond simple stratification of samples by one or more condition is not possible with `EZDynamics()`.

For normalization purposes, especially if analyzing pre-mRNA processing dynamics, you will need to provide `AverageAndRegularize()` with a table of feature lengths via the `feature_lengths` parameter. This will be used in all cases to length normalize read counts. Even in the case when you are just modeling mature mRNA dynamics, this is technically necessary for accurate estimation of scale factors.

The first step of `EZDynamics()` is attempted inference of normalization scale factors for read counts. If you have scale factors you calculated yourself, e.g. via specialized spike-in protocols, you can provide these via the `scale_factors` parameter. If not, `EZDynamics()` will try to infer these from the fraction labeled's in each `sample_feature` (e.g., in different subcellular compartments). This relies on having some `sample_feature`'s that are a combination of other `sample_feature`'s. For example, if analyzing subcellular fractionation data, you may have 1) total RNA; 2) cytoplasmic RNA; and 3) nuclear RNA. Total RNA = cytoplasmic + nuclear RNA, and thus the fraction of reads from labeled RNA is a function of the total cytoplasmic and nuclear fraction labeled's, as well as the relative molecular abundances of cytoplasmic and nuclear RNA. The latter is precisely the scale factors we need to estimate. If you do not have sufficient combinations of data to perform this scale factor estimation, `EZDynamics()` will only use the fraction labeled's for modeling kinetic parameters. It can then perform post-hoc normalization to estimate a single synthesis rate constant, using the downstream rate constants to infer the unknown normalization scale factor necessary to combine kinetic parameter estimates and read counts to infer this rate constant.

For estimating kinetic parameters, `EZDynamics()` infers the solution of the linear system of ODEs specified in your `graph` matrix input. This is done by representing the system of equations as a matrix, and deriving the general solution of the levels of each modeled species of RNA from the eigenvalues and eigenvectors of this matrix. While this makes `EZDynamics()` orders of magnitude more efficient than if it had to numerically infer the solution for each round of optimization, needing to compute eigenvalues and eigenvectors in each optimization iteration is still non-trivial, meaning that `EZDynamics()` may take anywhere from 10s of minutes to a couple hours to run, depending on how complex your model is and how many distinct set of samples and experimental conditions you have.

Value

`EZbakRData` object with an additional "dynamics" table.

Examples

```
##### MODELING CYTOPLASMIC TO NUCLEAR FLOW

### Simulate data and get replicate average fractions estimates
simdata <- EZSimulate(
  nfeatures = 30,
  ntreatments = 1,
  mode = "dynamics",
  label_time = c(1, 3),
  dynamics_preset = "nuc2cyto"
)

ezbdo <- EZbakRData(simdata$cB, simdata$metadf)
ezbdo <- EstimateFractions(ezbdo)
ezbdo <- AverageAndRegularize(ezbdo,
                              formula_mean = ~tl:compartment - 1,
                              type = "fractions",
                              parameter = "logit_fraction_highTC")

### ODE model: the graph and measured species
graph <- matrix(c(0, 1, 0,
                  0, 0, 2,
                  3, 0, 0),
                nrow = 3,
                ncol = 3,
                byrow = TRUE)
colnames(graph) <- c("0", "N", "C")
rownames(graph) <- colnames(graph)

modeled_to_measured <- list(
  nuclear = list(GF ~ N),
  cytoplasm = list(GF ~ C),
  total = list(GF ~ C + N) # total RNA is a combination of C and N
)

### Fit model
ezbdo <- EZDynamics(ezbdo,
                    graph = graph,
                    sub_features = "GF",
                    grouping_features = "GF",
                    sample_feature = "compartment",
                    modeled_to_measured = ode_models$nuc2cyto$formulas)
```

EZget *Easily get EZbakR table of estimates of interest*

Description

`EZget()` returns a table of interest from your `EZbakRData` object. It is meant to make it easier to find and access certain analyses, as a single `EZbakRData` object may include analyses of different features, kinetic parameters, dynamical systems models, comparisons, etc.

Usage

```
EZget(
  obj,
  type = c("fractions", "kinetics", "readcounts", "averages", "comparisons", "dynamics"),
  features = NULL,
  populations = NULL,
  fraction_design = NULL,
  isoforms = NULL,
  kstrat = NULL,
  parameter = NULL,
  counttype = NULL,
  design_factor = NULL,
  dynamics_design_factors = NULL,
  scale_factors = NULL,
  cstrat = NULL,
  feature_lengths = NULL,
  experimental = NULL,
  reference = NULL,
  param_name = NULL,
  param_function = NULL,
  formula_mean = NULL,
  sd_grouping_factors = NULL,
  fit_params = NULL,
  repeatID = NULL,
  sub_features = NULL,
  grouping_features = NULL,
  sample_feature = NULL,
  modeled_to_measured = NULL,
  graph = NULL,
  normalize_by_median = NULL,
  deconvolved = NULL,
  returnNameOnly = FALSE,
  exactMatch = FALSE,
  alwaysCheck = FALSE
)
```

Arguments

`obj`	EZbakRData object
`type`	The class of EZbakR outputs would you like to search through. Equivalent to the name of the list in the EZbakRData object that contains the tables of interest.
`features`	Features that must be present in the table of interest. If `exactMatch` is TRUE, then these features must also be the only features present in the table.
`populations`	Only relevant if `type` == "fractions". Mutational populations that must have been analyzed to generate the table of interest.
`fraction_design`	Only relevant if `type` == "fractions". Fraction design table used to generate the table of interest.
`isoforms`	If the relevant table is the result of isoform deconvolution
`kstrat`	Only relevant if `type` == "kinetics". Short for "kinetics strategy"; the strategy used to infer kinetic parameters.

`parameter`	Only relevant if `type` == "averages" or "comparisons". Which parameter was being averaged or compared?
`counttype`	String denoting what type of read count information you are looking for. Current options are "TMM_normalized", "transcript", and "matrix". TO-DO: Not sure this is being used in any way currently...
`design_factor`	design_factor specified in relevant run of `CompareParameters()`. Therefore, only relevant if type == "comparisons".
`dynamics_design_factors`	design_factors included in final `EZDynamics()` output. Therefore, only relevant if type == "dynamics".
`scale_factors`	Sample group scale factors used in `EZDynamics()`. Therefore, only relevant if type == "dynamics"
`cstrat`	Strategy used for comparative analyses. Can be: • contrast: If two parameters were compared via specifying the reference and experimental levels in `CompareParameters()` (for type == "averages"). • single_param: If a single parameter was passed to `CompareParameters()` via its `param_name` option. • dynamics: If output of `EZDynamics()` was passed to `CompareParameters()` • function: If function of multiple parameters was passed to `CompareParameter()` via its `param_function` option.
`feature_lengths`	Table of feature lengths used for length normalization.
`experimental`	Experimental condition specified in relevant run of `CompareParameters()`. Therefore, only relevant if type == "comparisons".
`reference`	Reference condition specified in relevant run of `CompareParameters()`. Therefore, only relevant if type == "comparisons".
`param_name`	Parameter name specified in relevant run of `CompareParameters()`. Therefore, only relevant if type == "comparisons"
`param_function`	Function of parameters specified in relevant run of `CompareParameters()`. Therefore, only relevant if type == "comparisons".
`formula_mean`	An R formula object specifying how the `parameter` of interest depends on the sample characteristics specified in `obj`'s metadf. Therefore, only relevant if type == "averages".
`sd_grouping_factors`	What metadf columns should data be grouped by when estimating standard deviations across replicates? Therefore, only relevant if type == "averages".
`fit_params`	Character vector of names of parameters in linear model fit. Therefore, only relevant if type == "averages".
`repeatID`	Numerical ID for duplicate objects with same metadata.
`sub_features`	Only relevant if type == "dynamics". Feature columns that distinguished between the different measured species when running `EZDynamics()`.
`grouping_features`	Only relevant if type == "dynamics. Features that were the overarching feature assignments by which `sub_features` were grouped when running `EZDynamics()`.

`sample_feature`	Only relevant if type == "dynamics". Name of the metadf column that distinguished the different classes of samples when running `EZDynamics()`.
`modeled_to_measured`	Only relevant if type == "dynamics". Specifies the relationship between `sub_features`, `sample_feature` (if specified), and the species in `graph`.
`graph`	Only relevant if type == "dynamics". NxN adjacency matrix, where N represents the number of species modeled when running `EZDynamics()`.
`normalize_by_median`	Whether median difference was subtracted from estimated kinetic parameter difference. Thus, only relevant if type == "comparisons".
`deconvolved`	Only relevant if type == "fractions". Boolean that is TRUE if fractions table is result of performing multi-feature deconvolution with `DeconvolveFractions()`.
`returnNameOnly`	If TRUE, then only the names of tables that passed your search criteria will be returned. Else, the single table passing your search criteria will be returned. If there is more than one table that passes your search criteria and `returnNameOnly` == `FALSE`, an error will be thrown.
`exactMatch`	If TRUE, then for `features` and `populations`, which can be vectors, ensure that provided vectors of features and populations exactly match the relevant metadata vectors.
`alwaysCheck`	If TRUE, then even if there is only a single table for the `type` of interest, still run all checks against queries.

Details

The input to `EZget()` is 1) the type of table you want to get ("fractions", "kinetics", "averages", "comparisons", or "dynamics") and 2) the metadata necessary to uniquely specify the table of interest. Above, every available piece of metadata that can be specified for this purpose is documented. You only need to specify the minimum information necessary. For example, if you would like to get a "fractions" table from an analysis of exon bins (feature == "exon_bins", and potentially other overarching features like "XF", "GF", or "rname"), and none of your other "fractions" tables includes exon_bins as a feature, then you can get this table with `EZget(ezbdo, type = "fractions", features = "exon_bins")`, where `ezbdo` is your `EZbakRData` object.

As another example, imagine you want to get a "kinetics" table from an analysis of gene-wise kinetic parameters (e.g., features == "XF"). You may have multiple "kinetics" tables, all with "XF" as at least one of their features. If all of the other tables have additional features though, then you can tell `EZget()` that "XF" is the only feature present in your table of interest by setting `exactMatch` to `TRUE`, which tells `EZget()` that the metadata you specify should exactly match the relevant metadata for the table of interest. So the call in this case would look like `EZget(ezbdo, type = "fractions", features = "XF", exactMatch = TRUE)`.

`EZget()` is used internally in almost every single EZbakR function to specify the input table for each analysis. Thus, the usage and metadata described here also applies to all functions that require you to specify which table you want to use (e.g., `EstimateKinetics()`, `AverageAndRegularize()`, `CompareParameters()`, etc.).

Value

Table of interest from the relevant `EZbakRdata` list (set by the type parameter).

Examples

```
# Simulate data to analyze
simdata <- EZSimulate(30)

# Create EZbakR input
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)

# Estimate Fractions
ezbdo <- EstimateFractions(ezbdo)

# Estimate Kinetics
ezbdo <- EstimateKinetics(ezbdo)

# Average log(kdeg) estimates across replicate
ezbdo <- AverageAndRegularize(ezbdo)

#' # Average log(ksyn) estimates across replicate
ezbdo <- AverageAndRegularize(ezbdo, parameter = "log_ksyn")

# Compare log(kdeg) across conditions
ezbdo <- CompareParameters(
ezbdo,
design_factor = "treatment",
reference = "treatment1",
experimental = "treatment2"
)

# Get the one and only fractions object
fxns <- EZget(ezbdo, type = "fractions")

# Get the log(ksyn) averages table
ksyn_avg <- EZget(ezbdo, type = "averages", parameter = "log_ksyn")
```

`EZMAPlot` *Make an MAPlot from EZbakR comparison*

Description

Make a plot of effect size (y-axis) vs. log10(read coverage) (x-axis), coloring points by position relative to user-defined decision cutoffs.

Usage

```
EZMAPlot(
  obj,
  parameter = "log_kdeg",
  design_factor = NULL,
  reference = NULL,
  experimental = NULL,
  param_name = NULL,
  param_function = NULL,
  features = NULL,
  condition = NULL,
  repeatID = NULL,
  exactMatch = TRUE,
  plotlog2 = TRUE,
  FDR_cutoff = 0.05,
  difference_cutoff = log(2),
  size = NULL,
  features_to_highlight = NULL,
  highlight_shape = 21,
  highlight_size_diff = 1,
  highlight_stroke = 0.7,
  highlight_fill = NA,
  highlight_color = "black"
)
```

Arguments

`obj`	An object of class `EZbakRCompare`, which is an `EZbakRData` object on which you have run `CompareParameters`
`parameter`	Name of parameter whose comparison you want to plot.
`design_factor`	Name of factor from `metadf` whose parameter estimates at different factor values you would like to compare.
`reference`	Name of reference `condition` factor level value.
`experimental`	Name of `condition` factor level value to compare to reference.
`param_name`	If you want to assess the significance of a single parameter, rather than the comparison of two parameters, specify that one parameter's name here.
`param_function`	NOT YET IMPLEMENTED. Will allow you to specify more complicated functions of parameters when hypotheses you need to test are combinations of parameters rather than individual parameters or simple differences in two parameters.
`features`	Character vector of feature names for which comparisons were made.
`condition`	Defunct parameter that has been replaced with `design_factor`. If provided gets passed to `design_factor` if `design_factor` is not already specified.
`repeatID`	If multiple `kinetics` or `fractions` tables exist with the same metadata, then this is the numerical index by which they are distinguished.
`exactMatch`	If TRUE, then `features` and `populations` have to exactly match those for a given fractions table for that table to be used. Means that you can't specify a subset of features or populations by default, since this is TRUE by default.
`plotlog2`	If TRUE, assume that log(parameter) difference is passed in and that you want to plot log2(parameter) difference.
`FDR_cutoff`	False discovery cutoff by which to color points.

`difference_cutoff`	Minimum absolute difference cutoff by which to color points.
`size`	Size of points, passed to `geom_point()` size parameter. If not specified, a point size is automatically chosen.
`features_to_highlight`	Features you want to highlight in the plot (black circle will be drawn around them). This can either be a data frame with one column per feature type in the comparison table you are visualizing, or a vector of feature names if the relevant comparison table will only have one feature type noted.
`highlight_shape`	Shape of points overlayed on highlighted features. Defaults to an open circle
`highlight_size_diff`	Sets how much larger should the points overlayed on the highlighted features be than the original plot points.
`highlight_stroke`	Stroke width of the points overlayed on the highlighted features.
`highlight_fill`	Fill color of the points overlayed on the highlighted features. Default is for them to be fill-less (`highlight_fill` == `NA`).
`highlight_color`	Color of the points overlayed on the highlighted points.

Details

`EZMAPlot()` accepts as input the output of `CompareParameters()`, i.e., an `EZbakRData` object with at least one "comparisons" table. It will plot the "avg_coverage" column in this table vs. the "difference" column. In the simplest case, "difference" represents a log-fold change in a kinetic parameter (e.g., kdeg) estimate. More complicated linear model fits and comparisons can yield different parameter estimates.

NOTE: some outputs of `CompareParameters()` are not meant for visualization via an MA plot. For example, when fitting certain interaction models, some of the parameter estimates may represent average log(kinetic paramter) in one condition. See discussion of one example of this here.

EZbakR estimates kinetic parameters in `EstimateKinetics()` and `EZDynamics()` on a log-scale. By default, since log2-fold changes are a bit easier to interpret and more common for these kind of visualizations, `EZMAPlot()` multiplies the y-axis value by log2(exp(1)), which is the factor required to convert from a log to a log2 scale. You can turn this off by setting `plotlog2` to `FALSE`.

Value

A `ggplot2` object. Y-axis = log2(estimate of interest (e.g., fold-change in degradation rate constant); X-axis = log10(average normalized read coverage); points colored by location relative to FDR and effect size cutoffs.

Examples

```
# Simulate data to analyze
simdata <- EZSimulate(30)
```

```
# Create EZbakR input
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)

# Estimate Fractions
ezbdo <- EstimateFractions(ezbdo)

# Estimate Kinetics
ezbdo <- EstimateKinetics(ezbdo)

# Average estimates across replicate
ezbdo <- AverageAndRegularize(ezbdo)

# Compare parameters across conditions
ezbdo <- CompareParameters(
ezbdo,
design_factor = "treatment",
reference = "treatment1",
experimental = "treatment2"
)

# Make MA plot (ggplot object that you can save and add/modify layers)
EZMAPlot(ezbdo)
```

EZpcaPlot *Make an MAPlot from EZbakR comparison*

Description

Make a plot of effect size (y-axis) vs. log10(read coverage) (x-axis), coloring points by position relative to user-defined decision cutoffs.

Usage

```
EZpcaPlot(
  obj,
  data_type = c("fraction_labeled", "reads"),
  features = NULL,
  exactMatch = TRUE,
  variance_decile = 7,
  center = TRUE,
  scale = TRUE,
  point_size = 3,
  metadf_cols_to_use = "all"
)
```

Arguments

`obj`	An object of class `EZbakRCompare`, which is an `EZbakRData` object on which you have run `CompareParameters`
`data_type`	Specifies what data to use for the PCA. Options are "fraction_labeled" (default; means using fraction high T-to-C or other mutation type estimate from EZbakR) or "reads" (means using log10(read counts + 1)).
`features`	Character vector of feature names for which comparisons were made.
`exactMatch`	If TRUE, then `features` and `populations` have to exactly match those for a given fractions table for that table to be used. Means that you can't specify a subset of features or populations by default, since this is TRUE by default.
`variance_decile`	Integer between (inclusive) 1 and 9. Features with sample-to-sample variance greater than the nth decile (n = `variance_decile`) will go into PCA.
`center`	From `prcomp()`: a logical value indicating whether the variables should be shifted to be zero centered. Alternately, a vector of length equal the number of columns of x can be supplied. The value is passed to scale.
`scale`	From `prcomp()`: a logical value indicating whether the variables should be scaled to have unit variance before the analysis takes place. Alternatively, a vector of length equal the number of columns of x can be supplied. The value is passed to scale.
`point_size`	Size of points in PCA plot
`metadf_cols_to_use`	Columns in the EZbakR metadf that will be used to color points in the PCA plot. Points will be colored by the interaction between all of these columns (i.e., samples with unique combinations of values of these columns will get unique colors). Default is to use all columns (except "sample"), specified as "all".

Details

`EZMAPlot()` accepts as input the output of `CompareParameters()`, i.e., an `EZbakRData` object with at least one "comparisons" table. It will plot the "avg_coverage" column in this table vs. the "difference" column. In the simplest case, "difference" represents a log-fold change in a kinetic parameter (e.g., kdeg) estimate. More complicated linear model fits and comparisons can yield different parameter estimates.

NOTE: some outputs of `CompareParameters()` are not meant for visualization via an MA plot. For example, when fitting certain interaction models, some of the parameter estimates may represent average log(kinetic paramter) in one condition. See discussion of one example of this here.

EZbakR estimates kinetic parameters in `EstimateKinetics()` and `EZDynamics()` on a log-scale. By default, since log2-fold changes are a bit easier to interpret and more common for these kind of visualizations, `EZMAPlot()` multiplies the y-axis value by log2(exp(1)), which is the factor required to convert from a log to a log2 scale. You can turn this off by setting `plotlog2` to `FALSE`.

Value

A `ggplot2` object.

Examples

```
# Simulate data to analyze
simdata <- EZSimulate(30)

# Create EZbakR input
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)

# Estimate Fractions
ezbdo <- EstimateFractions(ezbdo)


# Make MA plot (ggplot object that you can save and add/modify layers)
EZpcaPlot(ezbdo)
```

EZQC *Run quality control checks*

Description

`EZQC()` assesses multiple aspects of your NR-seq data and generates a number of plots visualizing dataset-wide trends.

Usage

```
EZQC(obj, ...)
```

Arguments

`obj`	EZbakRData or EZbakRFractions object.
`...`	Parameters passed to the class-specific method. If you have provided an EZbakRFractions object, then these can be (all play the same role as in `EstimateKinetics()`, that is they get passed to `EZget()` to find the fractions table you are interested in. See `?EstimateKinetics()` for details.): • `features` • `populations` • `fraction_design` If you have provided an EZbakRData object, then these can be (all same the same purpose as in `EstimateFractions`, so see `?EstimateFractions()` for details): • `mutrate_populations` • `features` • `filter_cols` • `filter_condition` • `remove_features`

Details

`EZQC()` checks the following aspects of your NR-seq data. If you have passed an `EZbakRData` object, then `EZQC()` checks:

- Raw mutation rates: In all sequencing reads, how many T's in the reference were a C in the read? The hope is that raw mutation rates are higher than -label controls in all +label samples. Higher raw mutation rates, especially when using standard label times (e.g., 2 hours or more in mammalian systems), are typically a sign of good label incorporation and low labeled RNA/read dropout. If you don't have -label samples, know that background mutation rates are typically less than 0.2%, so +label raw mutation rates several times higher than this would be preferable.
- Mutation rates in labeled and unlabeled reads: The raw mutation rate counts all mutations in all reads. In a standard NR-seq experiment performed with a single metabolic label, there are typically two populations of reads:
  1. Those from labeled RNA, having higher mutation rates due to chemical conversion/recoding of the metabolic label and 2) those from unlabeled RNA, having lower, background levels of mutations. `EZbakR` fits a two component mixture model to estimate the mutation rates in these two populations separately. A successful NR-seq experiment should have a labeled read mutation rate of > 1% and a low background mutation rate of < 0.3%.
  2. Read count replicate correlation: Simply the log10 read count correlation for replicates, as inferred from your `metadf`.

If you have passed an `EZbakRFractions` object, i.e., the output of `EstimateFractions()`, then in addition to the checks in the `EZbakRData` input case, `EZQC()` also checks:

- Fraction labeled distribution: This is the distribution of feature-wise fraction labeled's (or fraction high mutation content's) estimated by `EstimateFractions()`. The "ideal" is a distribution with mean around 0.5, as this maximizes the amount of RNA with synthesis and degradation kinetics within the dynamic range of the experiment. In practice, you will (and should) be at least a bit lower than this as longer label times risk physiological impacts of metabolic labeling.
- Fraction labeled replicate correlation: This is the logit(fraction labeled) correlation between replicates, as inferred from your `metadf`.

Value

A list of `ggplot2` objects visualizing the various aspects of your data assessed by `EZQC()`.

Examples

```
# Simulate data to analyze
simdata <- EZSimulate(30)

# Create EZbakR input
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)

# Estimate Fractions
ezbdo <- EstimateFractions(ezbdo)
```

```
# Run QC
QC <- EZQC(ezbdo)
```

EZQC.EZbakRArrowData *Run quality control checks*

Description

Run quality control checks

Usage

```
## S3 method for class 'EZbakRArrowData'
EZQC(
  obj,
  mutrate_populations = "all",
  features = "all",
  filter_cols = "all",
  filter_condition = `&`,
  remove_features = c("NA", "__no_feature"),
  ...
)
```

Arguments

`obj`	An EZbakRData object
`mutrate_populations`	Same as in `EstimateFractions()`. See `?EstimateFractions()` for details.
`features`	Same as in `EstimateFractions()`. See `?EstimateFractions()` for details.
`filter_cols`	Same as in `EstimateFractions()`. See `?EstimateFractions()` for details.
`filter_condition`	Same as in `EstimateFractions()`. See `?EstimateFractions()` for details.
`remove_features`	Same as in `EstimateFractions()`. See `?EstimateFractions()` for details.
`...`	Additional arguments. Currently goes unused.

Value

A list of `ggplot2` objects visualizing the various aspects of your data assessed by `EZQC()`.

`EZQC.EZbakRData` *Run quality control checks*

Description

Run quality control checks

Usage

```
## S3 method for class 'EZbakRData'
EZQC(
  obj,
  mutrate_populations = "all",
  features = "all",
  filter_cols = "all",
  filter_condition = `&`,
  remove_features = c("NA", "__no_feature"),
  ...
)
```

Arguments

`obj`	An EZbakRData object
`mutrate_populations`	Same as in `EstimateFractions()`. See `?EstimateFractions()` for details.
`features`	Same as in `EstimateFractions()`. See `?EstimateFractions()` for details.
`filter_cols`	Same as in `EstimateFractions()`. See `?EstimateFractions()` for details.
`filter_condition`	Same as in `EstimateFractions()`. See `?EstimateFractions()` for details.
`remove_features`	Same as in `EstimateFractions()`. See `?EstimateFractions()` for details.
`...`	Additional arguments. Currently goes unused.

Value

A list of `ggplot2` objects visualizing the various aspects of your data assessed by `EZQC()`.

EZQC.EZbakRFractions *Run quality control checks*

Description

Run quality control checks

Usage

```
## S3 method for class 'EZbakRFractions'
EZQC(obj, features = NULL, populations = NULL, fraction_design = NULL, ...)
```

Arguments

obj	EZbakRFractions object, which is an EZbakRData object on which EstimateFractions has been run.
features	Set of features analyzed in the fractions table you are interested QCing. This gets passed to EZget() to help find this table.
populations	Set of mutation types analyzed in the fractions table you are interested in QCing. This gets passed to EZget() to help find this table.
fraction_design	The fraction "design matrix" specified to get the fractions table you are interested in QCing. This gets passed to EZget() to help find this table.
...	Additional arguments. Currently goes unused.

Value

A list of ggplot2 objects visualizing the various aspects of your data assessed by EZQC().

EZSimulate *Simulate NR-seq data for multiple replicates of multiple biological conditions*

Description

EZSimulate() is a user friendly wrapper to SimulateMultiCondition(). It sets convenient defaults so as to quickly generate easy to interpret output. EZSimulate() has all of the same parameters as SimulateMultiCondition(), but it also has a number of additional parameters that guide its default behavior and allow you to simulate multi-condition data without specifying the multiple, sometimes complex, arguments that you would need to specify in SimulateMultiCondition() to get the same behavior. In particular, users only have to set a single parameter, nfeatures (number of features to simulate data for), by default. The EZSimulate()-unique parameters ntreatments and nreps have default values that guide the simulation in the case where only nfeatures is specified. In particular, nreps of ntreatments different conditions will be simulated, with the assumed model log(kdeg) ~ treatment and log(ksyn) ~ 1. In other words, Different kdeg values will be simulated for each treatment level, and ksyn values will not differ across conditions.

Usage

```
EZSimulate(
  nfeatures,
  mode = c("standard", "dynamics"),
  ntreatments = ifelse(mode == "standard", 2, 1),
  nreps = 3,
  nctlreps = 1,
  metadf = NULL,
  mean_formula = NULL,
  param_details = NULL,
  seqdepth = nfeatures * 2500,
  label_time = 2,
  pnew = 0.05,
  pold = 0.001,
  readlength = 200,
  Ucont_alpha = 25,
  Ucont_beta = 75,
  feature_prefix = "Gene",
  dispslope = 5,
  dispint = 0.01,
  logkdegsdtrend_slope = -0.3,
  logkdegsdtrend_intercept = -2.25,
  logksynsdtrend_slope = -0.3,
  logksynsdtrend_intercept = -2.25,
  logkdeg_mean = -1.9,
  logkdeg_sd = 0.7,
  logksyn_mean = 2.3,
  logksyn_sd = 0.7,
  logkdeg_diff_avg = 0,
  logksyn_diff_avg = 0,
  logkdeg_diff_sd = 0.5,
  logksyn_diff_sd = 0.5,
  pdiff_kd = 0.1,
  pdiff_ks = 0,
  pdiff_both = 0,
  pdo = 0,
 dynamics_preset = c("preRNA", "nuc2cyto", "preRNAwithPdeg", "nuc2cytowithNdeg",
    "subtlseq", "nuc2cytowithpreRNA"),
  unassigned_name = "__no_feature",
  dispersion = 1000,
  lfn_sd = 0.2,
  treatment_effects = NULL,
  effect_avg_default = 0,
  effect_sd_default = 0.5,
  fraction_affected_default = 0.5,
  log_means = NULL,
  log_sds = NULL
)
```

Arguments

`nfeatures`	Number of "features" (e.g., genes) for which to simulated data.
`mode`	Currently, EZSimulate can simulate in two modes: "standard" and "dynamics". The former is the default and involves simulating multiple conditions of standard NR-seq data. "dynamics" calls `SimulateDynamics()` under the hood to simulate a dynamical systems model of your choice. Most of the additional parameters do not apply if mode == "dynamics", except for those from dynamics_preset and on.
`ntreatments`	Number of distinct treatments to simulate. This parameter is only relevant if `metadf` is not provided.
`nreps`	Number of replicates of each treatment to simulate. This parameter is only relevant if `metadf` is not provided
`nctlreps`	Number of -s4U replicates of each treatment to simulate. This parameter is only relevant if `metadf` is not provided.

`metadf` A data frame with the following columns:

- sample: Names given to samples to simulate.
- `<details>`: Any number of columns with any names (not taken by other metadf columns) storing factors by which the samples can be stratified. These can be referenced in `mean_formula`, described below.

These parameters (described more below) can also be included in metadf to specify sample-specific simulation parameter:

- seqdepth
- label_time
- pnew
- pold
- readlength
- Ucont

`mean_formula` A formula object that specifies the linear model used to relate the factors in the `<details>` columns of `metadf` to average log(kdegs) and log(ksyns) in each sample.

`param_details` A data frame with one row for each column of the design matrix obtained from `model.matrix(mean_formula, metadf)` that describes how to simulate the linear model parameters. The columns of this data frame are:

- param: Name of linear model parameter as it appears in the column names of the design matrix from `model.matrix(mean_formula, metadf)`.
- reference: Boolean; TRUE if you want to treat that parameter as a "reference". This means that all other parameter values that aren't global parameters are set equal to this unless otherwise determined (see `pdiff_*` parameters for how it is determined if a parameter will differ from the reference).
- global: Boolean; TRUE if you want to treat that parameter as a global parameter. This means that a single value is used for all features.

- logkdeg_mean: If parameter is the reference, then its value for the log(kdeg) linear model will be drawn from a normal distribution with this mean. If it is a global parameter, then this value will be used. If it is neither of these, then its value in the log(kdeg) linear model will either be the reference (if there is no difference between this condition's value and the reference) or the reference's value + a normally distributed random variable centered on this value.
- logkdeg_sd: sd used for draws from normal distribution as described for `logkdeg_mean`.
- logksyn_mean: Same as `logkdeg_mean` but for log(ksyn) linear model.
- logksyn_sd: Same as `logkdeg_sd` but for log(kdeg) linear model.
- pdiff_ks: Proportion of features whose value of this parameter in the log(ksyn) linear model will differ from the reference's. Should be a number between 0 and 1, inclusive. For example, if `pdiff_ks` is 0.1, then for 10% of features, this parameter will equal the reference parameter + a normally distributed random variable with mean `logksyn_mean` and sd `logksyn_sd`. For the other 90% of features, this parameter will equal the reference.
- pdiff_kd: Same as `pdiff_ks` but for log(kdeg) linear model.
- pdiff_both: Proportion of features whose value for this parameter in BOTH the log(kdeg) and log(ksyn) linear models will differ from the reference. Value must be between 0 and min(c(pdiff_kd, pdiff_ks)) in that row.

If param_details is not specified by the user, the first column of the design matrix is assumed to represent the reference parameter, all parameters are assumed to be non-global, logkdeg_mean and logksyn_mean are set to the equivalently named parameter values described below for the reference and `logkdeg_diff_avg` and `logksyn_diff_avg` for all other parameters, logkdeg_sd and logksyn_sd are set to the equivalently named parameter values described below for the reference and `logkdeg_diff_sd` and `logksyn_diff_sd` for all other parameters, and pdiff_kd, pdiff_ks, and pdiff_both are all set to the equivalently named parameter values.

`seqdepth`	Total number of reads in each sample.
`label_time`	Length of s^4^U feed to simulate.
`pnew`	Probability that a T is mutated to a C if a read is new.
`pold`	Probability that a T is mutated to a C if a read is old.
`readlength`	Length of simulated reads. In this simple simulation, all reads are simulated as being exactly this length.
`Ucont_alpha`	Probability that a nucleotide in a simulated read from a given feature is a U is drawn from a beta distribution with shape1 = `Ucont_alpha`.
`Ucont_beta`	Probability that a nucleotide in a simulated read from a given feature is a U is drawn from a beta distribution with shape2 = `Ucont_beta`.
`feature_prefix`	Name given to the i-th feature is `paste0(feature_prefix, i)`. Shows up in the `feature` column of the output simulated data table.
`dispslope`	Negative binomial dispersion parameter "slope" with respect to read counts. See DESeq2 paper for dispersion model used.

`dispint`	Negative binomial dispersion parameter "intercept" with respect to read counts. See DESeq2 paper for dispersion model used.
`logkdegsdtrend_slope`	Slope for log10(read count) vs. log(kdeg) replicate variability trend
`logkdegsdtrend_intercept`	Intercept for log10(read count) vs. log(kdeg) replicate variability trend
`logksynsdtrend_slope`	Slope for log10(read count) vs. log(ksyn) replicate variability trend
`logksynsdtrend_intercept`	Intercept for log10(read count) vs. log(ksyn) replicate variability trend
`logkdeg_mean`	Mean of normal distribution from which reference log(kdeg) linear model parameter is drawn from for each feature if `param_details` is not provided.
`logkdeg_sd`	Standard deviation of normal distribution from which reference log(kdeg) linear model parameter is drawn from for each feature if `param_details` is not provided.
`logksyn_mean`	Mean of normal distribution from which reference log(ksyn) linear model parameter is drawn from for each feature if `param_details` is not provided.
`logksyn_sd`	Standard deviation of normal distribution from which reference log(ksyn) linear model parameter is drawn from for each feature if `param_details` is not provided.
`logkdeg_diff_avg`	Mean of normal distribution from which non-reference log(kdeg) linear model parameters are drawn from for each feature if `param_details` is not provided.
`logksyn_diff_avg`	Mean of normal distribution from which reference log(ksyn) linear model parameter are drawn from for each feature if `param_details` is not provided.
`logkdeg_diff_sd`	Standard deviation of normal distribution from which reference log(kdeg) linear model parameter are drawn from for each feature if `param_details` is not provided.
`logksyn_diff_sd`	Standard deviation of normal distribution from which reference log(ksyn) linear model parameter are drawn from for each feature if `param_details` is not provided.
`pdiff_kd`	Proportion of features for which non-reference log(kdeg) linear model parameters differ from the reference.
`pdiff_ks`	Proportion of features for which non-reference log(ksyn) linear model parameters differ from the reference.
`pdiff_both`	Proportion of features for which BOTH non-reference log(kdeg) and log(ksyn) linear model parameters differ from the reference.
`pdo`	Dropout rate; think of this as the probability that a s4U containing molecule is lost during library preparation and sequencing. If `pdo` is 0 (default) then there is not dropout.

`dynamics_preset`
: Which preset model to use for simulation of dynamics. Therefore, only relevant if `mode == dynamics`. Options are:

  nuc2cyto Simplest model of nuclear and cytoplasmic RNA dynamics: 0 -> N -> C -> 0

  preRNA Simplest model of pre-RNA and mature RNA dynamics: 0 -> P -> M -> 0

  preRNAwithPdeg Same as preRNA, but now pre-RNA can also degrade.

  nuc2cytowithNdeg Same as nuc2cyto, but now nuclear RNA can also degrade.

  subtlseq Subcellular TimeLapse-seq model, similar to that described in Ietswaart et al., 2024. Simplest model discussed there, lacking nuclear degradation: 0 -> CH -> NP -> CY -> PL -> 0, and CY can also degrade.

  nuc2cytowithpreRNA Combination of nuc2cyto and preRNA where preRNA is first synthesized, then either processed or exported to the cytoplasm. Processing can also occur in the cytoplasm, and mature nuclear RNA can be exported to the cytoplasm. Only mature RNA degrades.

`unassigned_name`
: String to give to reads not assigned to a given feature.

`dispersion`
: Negative binomial `size` parameter to use for simulating read counts

`lfn_sd`
: Logit(fn) replicate variability.

`treatment_effects`
: Data frame describing effects of treatment on each parameter. Should have five columns: "parameter_index", "treatment_index", "mean", "sd", and "fraction_affected". Each row corresponds to the effect the ith (i = treatment_index) treatment has on the jth (j = parameter_index) kinetic parameter. Effect sizes, on a log-scale, are drawn from a Normal distribution with mean and standard deviation set by the mean and sd columns, respectively. The number of non-zero effects is set by "fraction_affected", and is equal to `ceiling(nfeatures * fraction_affected)`. treatment_index of 1 will be ignored and can either be included or not.

`effect_avg_default`
: If `ntreatments` > 1, and `treatment_effects` is not provided, this will be the value of `mean` for all treatments and parameters imputed in `treatment_effects`.

`effect_sd_default`
: If `ntreatments` > 1, and `treatment_effects` is not provided, this will be the value of `sd` for all treatments and parameters imputed in `treatment_effects`.

`fraction_affected_default`
: If `ntreatments > 1`, and `treatment_effects` is not provided, this will be the value of `fraction_affected` for all treatments and parameters imputed in `treatment_effects`.

`log_means`
: Vector of log-Normal logmeans from which the distribution of feature-specific parameters will be drawn from. Length of vector should be the same as max(entries in `graph`), i.e., the number of parameters in your specified model. If not provided, will by default be `c(1, seq(from = -0.3, to = -2.5, length.out = max(graph) - 1 ))`. `1` for the ksyn parameter (which is always denoted 1 in the preset `graph`) is arbitrary. Remaining parameters will make it so indices order parameters from fastest to slowest process.

`log_sds`	Vector of log-Normal logsds from which the distribution of feature-specific parameters will be drawn from. If not provided, will be 0.4 for all parameters.

Value

A list containing 5 elements:

- cB: Tibble that can be provided as the `cB` arg to `EZbakRData()`.
- metadf: Tibble that can be provided as the `metadf` arg to `EZbakRData()`.
- PerRepTruth: Tibble containing replicate-by-replicate simulated ground truth
- AvgTruth: Tibble containing average simulated ground truth
- param_details: Tibble containing information about simulated linear model parameters

Examples

```
# Simulate standard data
simdata_standard <- EZSimulate(30)

# Simulate dynamical systems data
simdata_ode <- EZSimulate(30,
                          mode = "dynamics",
                          ntreatments = 1,
                          label_time = c(1, 3),
                          dynamics_preset = "nuc2cyto")
```

`EZVolcanoPlot` *Make a VolcanoPlot from EZbakR comparison*

Description

Make a plot of effect size (x-axis) vs. multiple-test adjusted p-value (y-axis), coloring points by position relative to user-defined decision cutoffs.

Usage

```
EZVolcanoPlot(
  obj,
  parameter = "log_kdeg",
  design_factor = NULL,
  reference = NULL,
  experimental = NULL,
  param_name = NULL,
  param_function = NULL,
  features = NULL,
  condition = NULL,
  normalize_by_median = NULL,
  repeatID = NULL,
  exactMatch = TRUE,
  plotlog2 = TRUE,
  FDR_cutoff = 0.05,
  difference_cutoff = log(2),
  size = NULL,
  features_to_highlight = NULL,
  highlight_shape = 21,
  highlight_size_diff = 1,
  highlight_stroke = 0.7,
  highlight_fill = NA,
  highlight_color = "black"
)
```

Arguments

`obj`	An object of class `EZbakRCompare`, which is an `EZbakRData` object on which you have run `CompareParameters`
`parameter`	Name of parameter whose comparison you want to plot.
`design_factor`	Name of factor from `metadf` whose parameter estimates at different factor values you would like to compare.
`reference`	Name of reference `condition` factor level value.
`experimental`	Name of `condition` factor level value to compare to reference.
`param_name`	If you want to assess the significance of a single parameter, rather than the comparison of two parameters, specify that one parameter's name here.
`param_function`	NOT YET IMPLEMENTED. Will allow you to specify more complicated functions of parameters when hypotheses you need to test are combinations of parameters rather than individual parameters or simple differences in two parameters.
`features`	Character vector of feature names for which comparisons were made.
`condition`	Defunct parameter that has been replaced with `design_factor`. If provided gets passed to `design_factor` if `design_factor` is not already specified.
`normalize_by_median`	Whether or not the median was subtracted from the estimated parameter differences.
`repeatID`	If multiple `kinetics` or `fractions` tables exist with the same metadata, then this is the numerical index by which they are distinguished.
`exactMatch`	If TRUE, then `features` has to exactly match those for a given comparisons table for that table to be used. Means that you can't specify a subset of features by default, since this is TRUE by default.
`plotlog2`	If TRUE, assume that log(parameter) difference is passed in and that you want to plot log2(parameter) difference. TO-DO: probably best to change this to a more general scale parameter by which the parameter is multiplied. Default would be log2(exp(1)) to convert log() to log2().
`FDR_cutoff`	False discovery cutoff by which to color points.

`difference_cutoff`
: Minimum absolute difference cutoff by which to color points.

`size`
: Size of points, passed to `geom_point()` size parameter. If not specified, a point size is automatically chosen.

`features_to_highlight`
: Features you want to highlight in the plot (black circle will be drawn around them). This can either be a data frame with one column per feature type in the comparison table you are visualizing, or a vector of feature names if the relevant comparison table will only have one feature type noted.

`highlight_shape`
: Shape of points overlayed on highlighted features. Defaults to an open circle

`highlight_size_diff`
: Sets how much larger should the points overlayed on the highlighted features be than the original plot points.

`highlight_stroke`
: Stroke width of the points overlayed on the highlighted features.

`highlight_fill`
: Fill color of the points overlayed on the highlighted features. Default is for them to be fill-less (`highlight_fill` == `NA`).

`highlight_color`
: Stroke color of the points overlayed on the highlighted points.

Details

`EZVolcanoPlot()` accepts as input the output of `CompareParameters()`, i.e., an `EZbakRData` object with at least one "comparisons" table. It will plot the "difference" column in this table versus -log10 of the "padj" column. In the simplest case, "difference" represents a log-fold change in a kinetic parameter (e.g., kdeg) estimate. More complicated linear model fits and comparisons can yield different parameter estimates.

NOTE: some outputs of `CompareParameters()` are not meant for visualization via a volcano plot. For example, when fitting certain interaction models, some of the parameter estimates may represent average log(kinetic paramter) in one condition. See discussion of one example of this here.

EZbakR estimates kinetic parameters in `EstimateKinetics()` and `EZDynamics()` on a log-scale. By default, since log2-fold changes are a bit easier to interpret and more common for these kind of visualizations, `EZVolcanoPlot()` multiplies the x-axis value by log2(exp(1)), which is the factor required to convert from a log to a log2 scale. You can turn this off by setting `plotlog2` to `FALSE`.

Value

A `ggplot2` object. X-axis = log2(estimate of interest (e.g., fold-change in degradation rate constant); Y-axis = -log10(multiple test adjusted p-value); points colored by location relative to FDR and effect size cutoffs.

Examples

```
# Simulate data to analyze
simdata <- EZSimulate(30)
```

```
# Create EZbakR input
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)

# Estimate Fractions
ezbdo <- EstimateFractions(ezbdo)

# Estimate Kinetics
ezbdo <- EstimateKinetics(ezbdo)

# Average estimates across replicate
ezbdo <- AverageAndRegularize(ezbdo)

# Compare parameters across conditions
ezbdo <- CompareParameters(
ezbdo,
design_factor = "treatment",
reference = "treatment1",
experimental = "treatment2"
)

# Make volcano plot (ggplot object that you can save and add/modify layers)
EZVolcanoPlot(ezbdo)
```

get_normalized_read_counts   *Get normalized read counts from either a cB table or* `EZbakRFractions` *object.*

Description

Uses TMM normalization strategy, similar to that used by DESeq2 and edgeR.

Usage

```
get_normalized_read_counts(
  obj,
  features_to_analyze,
  fractions_name = NULL,
  feature_lengths = NULL,
  scale_factors = NULL
)

## S3 method for class 'EZbakRFractions'
get_normalized_read_counts(
  obj,
  features_to_analyze,
  fractions_name = NULL,
  feature_lengths = NULL,
  scale_factors = NULL
)

## S3 method for class 'EZbakRData'
get_normalized_read_counts(
  obj,
  features_to_analyze,
  fractions_name = NULL,
  feature_lengths = NULL,
  scale_factors = NULL
)
```

Arguments

`obj`	An `EZbakRData` or `EZbakRFractions` object.
`features_to_analyze`	Features in relevant table
`fractions_name`	Name of fractions table to use
`feature_lengths`	Table of effective lengths for each feature combination in your data. For example, if your analysis includes features named GF and XF, this should be a data frame with columns GF, XF, and length.
`scale_factors`	Dataframe with two columns, one being "sample" (sample names) and the other being "scale_factor" (value to divide read counts by to normalize them)

Value

Data table of normalized read counts.

Methods (by class)

- `get_normalized_read_counts(EZbakRFractions)`: Method for class **EZbakRFractions** Get normalized read counts from fractions table.
- `get_normalized_read_counts(EZbakRData)`: Method for class **EZbakRData** Get normalized read counts from a cB table.

Examples

```
# Simulate data
simdata <- EZSimulate(30)

# Create EZbakRData object
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)

# Get normalized read counts
reads <- get_normalized_read_counts(ezbdo, features_to_analyze = "feature")
```

`ImportIsoformQuant` *Import transcript isoform quantification into EZbakRData object*

Description

A convenient wrapper to `tximport()` for importing isoform quantification data into an EZbakRData object. You need to run this before running `EstimateIsoformFractions`.

Usage

```
ImportIsoformQuant(
  obj,
  files,
 quant_tool = c("none", "salmon", "sailfish", "alevin", "piscem", "kallisto", "rsem",
    "stringtie"),
  txIn = TRUE,
  ...
)
```

Arguments

`obj`	An `EZbakRData` object.
`files`	A named vector of paths to all transcript quantification files that you would like to import. This will be passed as the first argument of `tximport::tximport()` (also named `files`). The names of this vector should be the same as the sample names as they appear in the metadf of the `EZbakRData` object.
`quant_tool`	String denoting the type of software used to generate the abundances. Will get passed to the `type` argument of `tximport::tximport()`. As described in the documentation for `tximport` 'Options are "salmon", "sailfish", "alevin", "piscem", "kallisto", "rsem", "stringtie", or "none". This argument is used to autofill the arguments below (geneIdCol, etc.) "none" means that the user will specify these columns. Be aware that specifying type other than "none" will ignore the arguments below (geneIdCol, etc.)'. Referenced 'arguments below' can be specified as part of `...`.
`txIn`	Whether or now you are providing isoform level quantification files. Alternative (`txIn = FALSE`) is gene-level quantification. In `ImportIsoformQuant`, `txIn` gets passed to BOTH the `txIn` and `txOut` parameters in `tximport()`.
`...`	Additional arguments to be passed to `tximport::tximport()`. Especially relevant if you set `quant_tool` to "none".

Value

An `EZbakRData` object with an additional element in the `readcounts` list named "isform_quant_<quant_tool>". It contains TPM, expected_count, and effective length information for each transcript_id and each sample.

Examples

```
# Dependencies for example
library(dplyr)
library(data.table)

# Simulate and analyze data
simdata <- EZSimulate(30)
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)
ezbdo <- EstimateFractions(ezbdo)

# Hack to generate example quantification files
savedir <- tempdir()
rsem_data <- tibble(
  transcript_id = paste0("tscript_feature", 1:30),
  gene_id = paste0("feature", 1:30),
  length = 1000,
  effective_length = 1000,
  expected_count = 1000,
  TPM = 10,
  FPKM = 10,
  IsoPct = 1
)

fwrite(rsem_data, file.path(savedir, "Sample_1.isoforms.results"), sep = '\t')

files <- file.path(savedir,"Sample_1.isoforms.results")
names(files) <- "Sample_1"

# Read in file
ezbdo <- ImportIsoformQuant(ezbdo, files, quant_tool = "rsem")
```

`new_EZbakRArrowData` `EZbakRArrowData` *object constructor for internal use*

Description

`new_EZbakRArrowData` efficiently creates an object of class `EZbakRArrowData`. It does not perform any rigorous checks of the legitimacy of this object.

Usage

```
new_EZbakRArrowData(cBds, metadf)
```

Arguments

`cBds`	Arrow dataset tracking the sample ID, mutational and nucleotide content, and feature assignment of sequencing reads.
`metadf`	Data frame tracking features of each of the samples included in `cBds`.

`new_EZbakRData` — `EZbakRDataobject` *constructor for internal use*

Description

`new_EZbakRData` efficiently creates an object of class `EZbakRData`. It does not perform any rigorous checks of the legitimacy of this object.

Usage

```
new_EZbakRData(cB, metadf)
```

Arguments

`cB`	Data frame tracking the sample ID, mutational and nucleotide content, and feature assignment of sequencing reads.
`metadf`	Data frame tracking features of each of the samples included in `cB`.

`new_EZbakRFractions` — `EZbakRFractions` *object constructor*

Description

`new_EZbakRFractions` efficiently creates an object of class `EZbakRFractions`. It does not perform any rigorous checks of the legitimacy of this object.

Usage

```
new_EZbakRFractions(fractions, metadf, name = NULL, character_limit = 20)
```

Arguments

`fractions`	Data frame containing information about the fraction of reads from each mutational population of interest.
`metadf`	Data frame reporting aspects of each of the samples included
`name`	Optional; name to give to fractions table.
`character_limit`	Maximum number of characters for naming out fractions output. EZbakR will try to name this as a "_" separated character vector of all of the features analyzed. If this name is greater than `character_limit`, then it will default to "fraction#", where "#" represents a simple numerical ID for the table. in `fractions`

`new_EZbakRKinetics` *`EZbakRKinetics` object constructor*

Description

`new_EZbakRKinetics` efficiently creates an object of class `EZbakRKinetics`. It does not perform any rigorous checks of the legitimacy of this object.

Usage

```
new_EZbakRKinetics(
  kinetics,
  features,
  metadf,
  name = NULL,
  character_limit = 20
)
```

Arguments

`kinetics`	Data frame containing information about the kinetic parameters of interest for each set of features tracked.
`features`	Features tracked in `kinetics` data frame. Needs to be specified explicitly as it cannot be automatically inferred.
`metadf`	Data frame describing each of the samples included
`name`	Optional; name to give to fractions table.
`character_limit`	Maximum number of characters for naming out fractions output. EZbakR will try to name this as a "_" separated character vector of all of the features analyzed. If this name is greater than `character_limit`, then it will default to "fraction#", where "#" represents a simple numerical ID for the table. in `kinetics`

`NormalizeForDropout` *Normalize for experimental/bioinformatic dropout of labeled RNA.*

Description

Uses the strategy described here, and similar to that originally presented in Berg et al. 2024, to normalize for dropout. Normalizing for dropout means identifying a reference sample with low dropout, and estimating dropout in each sample relative to that sample.

Usage

```
NormalizeForDropout(
  obj,
  normalize_across_tls = FALSE,
  grouping_factors = NULL,
  features = NULL,
  populations = NULL,
  fraction_design = NULL,
  repeatID = NULL,
  exactMatch = TRUE,
  read_cutoff = 25
)
```

Arguments

`obj`	An EZbakRFractions object, which is an EZbakRData object on which you have run `EstimateFractions()`.
`normalize_across_tls`	If TRUE, samples from different label times will be normalized by finding a max inferred degradation rate constant (kdeg) sample and using that as a reference. Degradation kinetics with this max will be assumed to infer reference fraction news at different label times
`grouping_factors`	Which sample-detail columns in the metadf should be used to group -s4U samples by for calculating the average -s4U RPM? The default value of `NULL` will cause no sample-detail columns to be used.
`features`	Character vector of the set of features you want to stratify reads by and estimate proportions of each RNA population. The default of `NULL` will expect there to be only one fractions table in the EZbakRFractions object.
`populations`	Mutational populations that were analyzed to generate the fractions table to use. For example, this would be "TC" for a standard s4U-based nucleotide recoding experiment.
`fraction_design`	"Design matrix" specifying which RNA populations exist in your samples. By default, this will be created automatically and will assume that all combinations of the `mutrate_populations` you have requested to analyze are present in your data. If this is not the case for your data, then you will have to create one manually. See docs for `EstimateFractions` (run ?EstimateFractions()) for more details.
`repeatID`	If multiple `fractions` tables exist with the same metadata, then this is the numerical index by which they are distinguished.
`exactMatch`	If TRUE, then `features` must exactly match the `features` metadata for a given fractions table for it to be used. Means that you cannot specify a subset of features by default. Set this to FALSE if you would like to specify a feature subset.
`read_cutoff`	Minimum number of reads for a feature to be used to fit dropout model.

Details

`NormalizeForDropout()` has a number of unique advantages relative to `CorrectDropout()`:

- `NormalizeForDropout()` doesn't require -label control data.
- `NormalizeForDropout()` compares an internally normalized quantity (fraction new) across samples, which has some advantages over the absolute dropout estimates derived from comparisons of normalized read counts in `CorrectDropout()`.
- `NormalizeForDropout()` may be used to normalize half-life estimates across very different biological contexts (e.g., different cell types).

There are also some caveats to be aware of when using `NormalizeForDropout()`:

- Be careful using `NormalizeForDropout()` when you have multiple different label times. Dropout normalization requires each sample be compared to a reference sample with the same label time. Thus, normalization will be performed separately for groups of samples with different label times. If the extent of dropout in the references with different label times is different, there will still be unaccounted for dropout biases between some of the samples.
- `NormalizeForDropout()` effectively assumes that there are no true global differences in turnover kinetics of RNA. If such differences actually exist (e.g., half-lives in one context are on average truly lower than those in another), `NormalizeForDropout()` risks normalizing away these real differences. This is similar to how statistical normalization strategies implemented in differential expression analysis software like DESeq2 assumes that there are no global differences in RNA levels.

By default, all samples with same label time are normalized with respect to a reference sample chosen from among them. If you want to further separate the groups of samples that are normalized together, specify the columns of your metadf by which you want to additionally group factors in the `grouping_factors` parameter. This behavior can be changed by setting `normalize_across_tls` to `TRUE`, which will

Value

An `EZbakRData` object with the specified "fractions" table replaced with a dropout corrected table.

Examples

```
# Simulate data to analyze
simdata <- EZSimulate(30)

# Create EZbakR input
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)

# Estimate Fractions
ezbdo <- EstimateFractions(ezbdo)

# Normalize for dropout
ezbdo <- NormalizeForDropout(ezbdo)
```

`ode_models` *Example ODE model graphs and formulas*

Description

A list of example "graphs" and specie formulas that can be passed to `EZDynamics` or `SimulateDynamics`, and that are used under the hood by `EZSimulate` to facilitate simulations of ODE models.

Usage

```
ode_models
```

Format

`ode_models`**:**

A list with 6 elements

- **nuc2cyto** Simplest model of nuclear and cytoplasmic RNA dynamics: 0 -> N -> C -> 0
- **preRNA** Simplest model of pre-RNA and mature RNA dynamics: 0 -> P -> M -> 0
- **preRNAwithPdeg** Same as preRNA, but now pre-RNA can also degrade.
- **nuc2cytowithNdeg** Same as nuc2cyto, but now nuclear RNA can also degrade.
- **subtlseq** Subcellular TimeLapse-seq model, similar to that described in Ietswaart et al., 2024. Simplest model discussed there, lacking nuclear degradation: 0 -> CH -> NP -> CY -> PL -> 0, and CY can also degrade.
- **nuc2cytowithpreRNA** Combination of nuc2cyto and preRNA where preRNA is first synthesized, then either processed or exported to the cytoplasm. Processing can also occur in the cytoplasm, and mature nuclear RNA can be exported to the cytoplasm. Only mature RNA degrades.

Each element of list has two items

- **graph** Matrix representation of ODE system graph.
- **formulas** Formula objects relating measured species to modeled species.

References

Ietswaart et al. (2024) Molecular Cell. 84(14), 2765-2784.

print.EZbakRArrowData *Print method for* EZbakRArrowData *objects*

Description

Print method for EZbakRArrowData objects

Usage

```
## S3 method for class 'EZbakRArrowData'
print(x, max_name_chars = 60, ...)
```

Arguments

x	An EZbakRArrowData object.
max_name_chars	Maximum number of characters to print on each line
...	Ignored

Value

The input EZbakRData object, invisibly

Examples

```
# Simulate data to analyze
simdata <- SimulateOneRep(30)

# Create EZbakR input
metadf <- data.frame(sample = "sampleA", tl = 2)
ezbdo <- EZbakRData(simdata$cB, metadf)

# Print
print(ezbdo)
```

print.EZbakRData *Print method for* EZbakRData *objects*

Description

Print method for EZbakRData objects

Usage

```
## S3 method for class 'EZbakRData'
print(x, max_name_chars = 60, ...)
```

Arguments

`x`	An `EZbakRData` object.
`max_name_chars`	Maximum number of characters to print on each line
`...`	Ignored

Value

The input `EZbakRData` object, invisibly

Examples

```
# Simulate data to analyze
simdata <- SimulateOneRep(30)

# Create EZbakR input
metadf <- data.frame(sample = "sampleA", tl = 2)
ezbdo <- EZbakRData(simdata$cB, metadf)

# Print
print(ezbdo)
```

`SimpleSim` *Simple simulation function*

Description

Simple simulation function

Usage

```
SimpleSim(
  nreads = 1000,
  fn = 0.5,
  pnew = 0.05,
  pold = 0.001,
  rlen = 100,
  Ucont = 0.25
)
```

Arguments

`nreads`	Number of reads to simulate
`fn`	Fraction of reads that are new in simulation. Whether a read will be new will be determined by a draw from a Bernoulli(fn) distribution.
`pnew`	T-to-C mutation rate in new reads

`pold`	T-to-C mutation rate in old reads
`rlen`	Length of simulated reads
`Ucont`	Fraction of nucleotides in simulated reads that are Ts (U in RNA)

Value

Tibble with 3 columns:

- nT: Simulated number of Ts
- TC: Simulated number of T-to-C mutations
- n: Number of simulated reads with nT Ts and TC mutations.

Examples

```
# Simulate 1 gene worth of data data
simdata <- SimpleSim()
```

`SimulateDynamics` *Simulation of generalized dynamical system model.*

Description

`SimulateDynamics()` simulates any specified dynamical system of interconverting RNA species. Its required input is similar to that of `EstimateDynamics()`, i.e., an adjacency matrix describing the set of species and how they are related to one another and a list of formula relating actually assayed species to the modeled species. Currently, `SimulateDynamics()` implements a naive heteroskedastic replicate variability simulation and is not designed to simulate multiple experimental conditions.

Usage

```
SimulateDynamics(
  nfeatures,
  graph,
  metadf,
  log_means,
  log_sds,
  ntreatments = 1,
  treatment_effects = NULL,
  formula_list = NULL,
  unassigned_name = "__no_feature",
  seqdepth = nfeatures * 2500,
  dispersion = 100,
  lfn_sd = 0.2,
  effect_avg_default = 0,
  effect_sd_default = 0.5,
  fraction_affected_default = 0.5,
  ...
)
```

Arguments

`nfeatures`	Number of "features" to simulate data for. A "feature" in this case may contain a number of "sub-features". For example, you may want to simulate pre-RNA and mature RNA for a set of "genes", in which case the number of features is the number of genes.
`graph`	An adjacency matrix describing the reaction diagram graph relating the various RNA species to one another.
`metadf`	Data frame with two required columns (`sample` and `tl`). `sample` represents names given to each simulated sample. `tl` represents the label time for that sample. Additional columns can specify other features of the sample, like what subcellular compartment the sample is taken from. **NOTE: Not sure I am actually using these optional columns in any useful capacity anymore**.
`log_means`	Vector of log-Normal logmeans from which the distribution of feature-specific parameters will be drawn from. Length of vector should be the same as max(entries in `graph`), i.e., the number of parameters in your specified model.
`log_sds`	Vector of log-Normal logsds from which the distribution of feature-specific parameters will be drawn from.
`ntreatments`	Number of distinct experimental treatments to simulate. By default, only a single "treatment" (you might refer to this as wild-type, or control) is simulated. Increase this if you would like to explore performing comparative dynamical systems modeling.
`treatment_effects`	Data frame describing effects of treatment on each parameter. Should have five columns: "parameter_index", "treatment_index", "mean", "sd", and "fraction_affected". Each row corresponds to the effect the ith (i = treatment_index) treatment has on the jth (j = parameter_index) kinetic parameter. Effect sizes, on a log-scale, are drawn from a Normal distribution with mean and standard deviation set by the mean and sd columns, respectively. The number of nonzero effects is set by "fraction_affected", and is equal to `ceiling(nfeatures * fraction_affected)`. treatment_index of 1 will be ignored and can either be included or not.
`formula_list`	A list of named lists. The names of each sub-list should be the same as the sample names as they are found in `metadf`. Each sub-list should be a list of formula relating feature names that will show up as columns of the simulated cB to species modeled in your `graph`. This only needs to be specified if you want to simulate the scenario where some of the measured species are a sum of modeled species.
`unassigned_name`	String to give to reads not assigned to a given feature.
`seqdepth`	Total number of reads in each sample.
`dispersion`	Negative binomial `size` parameter to use for simulating read counts

`lfn_sd`	Logit(fn) replicate variability.
`effect_avg_default`	If `ntreatments` > 1, and `treatment_effects` is not provided, this will be the value of `mean` for all treatments and parameters imputed in `treatment_effects`.
`effect_sd_default`	If `ntreatments` > 1, and `treatment_effects` is not provided, this will be the value of `sd` for all treatments and parameters imputed in `treatment_effects`.
`fraction_affected_default`	If `ntreatments` > 1, and `treatment_effects` is not provided, this will be the value of `fraction_affected` for all treatments and parameters imputed in `treatment_effects`.
`...`	Parameters passed to `SimulateOneRep()`.

`SimulateIsoforms` *Simulation of transcript isoform kinetic parameters.*

Description

`SimulateIsoforms()` performs a simple simulation of isoform-specific kinetic parameters to showcase and test `EstimateIsoformFractions()`. It assumes that there are a set of reads (fraction of total set by `funique` parameter) which map uniquely to a given isoform, while the rest are ambiguous to all isoforms from that gene. Mutational content of these reads are simulated as in `SimulateOneRep()`.

Usage

```
SimulateIsoforms(
  nfeatures,
  nt = NULL,
  seqdepth = nfeatures * 2500,
  label_time = 4,
  sample_name = "sampleA",
  feature_prefix = "Gene",
  pnew = 0.1,
  pold = 0.002,
  funique = 0.2,
  readlength = 200,
  Ucont = 0.25,
  avg_numiso = 2,
  psynthdiff = 0.5,
  logkdeg_mean = -1.9,
  logkdeg_sd = 0.7,
  logksyn_mean = 2.3,
  logksyn_sd = 0.7
)
```

Arguments

`nfeatures`	Number of "features" to simulate data for. Each feature will have a simulated number of transcript isoforms
`nt`	(Optional), can provide a vector of the number of isoforms you would like to simulate for each of the `nfeatures` features. Vector can either be length 1, in which case that many isoforms will be simulated for all features, or length equal to `nfeatures`.
`seqdepth`	Total number of sequencing reads to simulate
`label_time`	Length of s^4^U feed to simulate.
`sample_name`	Character vector to assign to `sample` column of output simulated data table (the cB table).
`feature_prefix`	Name given to the i-th feature is `paste0(feature_prefix, i)`. Shows up in the `feature` column of the output simulated data table.
`pnew`	Probability that a T is mutated to a C if a read is new.
`pold`	Probability that a T is mutated to a C if a read is old.
`funique`	Fraction of reads that uniquely "map" to a single isoform.
`readlength`	Length of simulated reads. In this simple simulation, all reads are simulated as being exactly this length.
`Ucont`	Percentage of nucleotides simulated to be U's.
`avg_numiso`	Average number of isoforms for each feature. Feature-specific isoform counts are drawn from a Poisson distribution with this average. NOTE: to insure that all features have multiple isoforms, the simulated number of isoforms drawn from a Poisson distribution is incremented by 2. Thus, the actual average number of isoforms from each feature is `avg_numiso` + 2.
`psynthdiff`	Percentage of genes for which all isoform abundance differences are synthesis driven. If not synthesis driven, then isoform abundance differences will be driven by differences in isoform kdegs.
`logkdeg_mean`	meanlog of a log-normal distribution from which kdegs are simulated
`logkdeg_sd`	sdlog of a log-normal distribution from which kdegs are simulated
`logksyn_mean`	meanlog of a log-normal distribution from which ksyns are simulated
`logksyn_sd`	sdlog of a log-normal distribution from which ksyns are simulated

Value

List with two elements:

- cB: Tibble that can be passed as the `cB` arg to `EZbakRData()`.
- ground_truth: Tibble containing simulated ground truth.

Examples

```
simdata <- SimulateIsoforms(30)
```

SimulateMultiCondition

Simulate NR-seq data for multiple replicates of multiple biological conditions

Description

`SimulateMultiCondition` is a highly flexibly simulator that combines linear modeling of log(kdeg)'s and log(ksyn)'s with `SimulateOneRep` to simulate an NR-seq dataset. The linear model allows you to simulate multiple distinct treatments, batch effects, interaction effects, etc. The current downside for its flexibility is its relative complexity to implement. Easier to use simulators are on the way to EZbakR.

Usage

```
SimulateMultiCondition(
  nfeatures,
  metadf,
  mean_formula,
  param_details = NULL,
  seqdepth = nfeatures * 2500,
  label_time = 2,
  pnew = 0.05,
  pold = 0.001,
  readlength = 200,
  Ucont_alpha = 25,
  Ucont_beta = 75,
  feature_prefix = "Gene",
  dispslope = 5,
  dispint = 0.01,
  logkdegsdtrend_slope = -0.3,
  logkdegsdtrend_intercept = -2.25,
  logksynsdtrend_slope = -0.3,
  logksynsdtrend_intercept = -2.25,
  logkdeg_mean = -1.9,
  logkdeg_sd = 0.7,
  logksyn_mean = 2.3,
  logksyn_sd = 0.7,
  logkdeg_diff_avg = 0,
  logksyn_diff_avg = 0,
  logkdeg_diff_sd = 0.5,
  logksyn_diff_sd = 0.5,
  pdiff_kd = 0.1,
  pdiff_ks = 0,
  pdiff_both = 0,
  pdo = 0
)
```

Arguments

`nfeatures` Number of "features" (e.g., genes) to simulate data for

`metadf` A data frame with the following columns:

- sample: Names given to samples to simulate.
- `<details>`: Any number of columns with any names (not taken by other metadf columns) storing factors by which the samples can be stratified. These can be referenced in `mean_formula`, described below.

These parameters (described more below) can also be included in metadf to specify sample-specific simulation parameter:

- seqdepth
- label_time
- pnew
- pold
- readlength
- Ucont

`mean_formula` A formula object that specifies the linear model used to relate the factors in the `<details>` columns of `metadf` to average log(kdegs) and log(ksyns) in each sample.

`param_details` A data frame with one row for each column of the design matrix obtained from `model.matrix(mean_formula, metadf)` that describes how to simulate the linear model parameters. The columns of this data frame are:

- param: Name of linear model parameter as it appears in the column names of the design matrix from `model.matrix(mean_formula, metadf)`.
- reference: Boolean; TRUE if you want to treat that parameter as a "reference". This means that all other parameter values that aren't global parameters are set equal to this unless otherwise determined (see `pdiff_*` parameters for how it is determined if a parameter will differ from the reference).
- global: Boolean; TRUE if you want to treat that parameter as a global parameter. This means that a single value is used for all features.
- logkdeg_mean: If parameter is the reference, then its value for the log(kdeg) linear model will be drawn from a normal distribution with this mean. If it is a global parameter, then this value will be used. If it is neither of these, then its value in the log(kdeg) linear model will either be the reference (if there is no difference between this condition's value and the reference) or the reference's value + a normally distributed random variable centered on this value.
- logkdeg_sd: sd used for draws from normal distribution as described for `logkdeg_mean`.
- logksyn_mean: Same as `logkdeg_mean` but for log(ksyn) linear model.
- logksyn_sd: Same as `logkdeg_sd` but for log(kdeg) linear model.
- pdiff_ks: Proportion of features whose value of this parameter in the log(ksyn) linear model will differ from the reference's. Should be a number between 0 and 1, inclusive. For example, if `pdiff_ks` is 0.1, then for 10% of features,

this parameter will equal the reference parameter + a normally distributed random variable with mean `logksyn_mean` and sd `logksyn_sd`. For the other 90% of features, this parameter will equal the reference.

- pdiff_kd: Same as `pdiff_ks` but for log(kdeg) linear model.
- pdiff_both: Proportion of features whose value for this parameter in BOTH the log(kdeg) and log(ksyn) linear models will differ from the reference. Value must be between 0 and min(c(pdiff_kd, pdiff_ks)) in that row.

If param_details is not specified by the user, the first column of the design matrix is assumed to represent the reference parameter, all parameters are assumed to be non-global, logkdeg_mean and logksyn_mean are set to the equivalently named parameter values described below for the reference and `logkdeg_diff_avg` and `logksyn_diff_avg` for all other parameters, logkdeg_sd and logksyn_sd are set to the equivalently named parameter values described below for the reference and `logkdeg_diff_sd` and `logksyn_diff_sd` for all other parameters, and pdiff_kd, pdiff_ks, and pdiff_both are all set to the equivalently named parameter values.

`seqdepth`	Only relevant if `read_vect` is not provided; in that case, this is the total number of reads to simulate.
`label_time`	Length of s^4^U feed to simulate.
`pnew`	Probability that a T is mutated to a C if a read is new.
`pold`	Probability that a T is mutated to a C if a read is old.
`readlength`	Length of simulated reads. In this simple simulation, all reads are simulated as being exactly this length.
`Ucont_alpha`	Probability that a nucleotide in a simulated read from a given feature is a U is drawn from a beta distribution with shape1 = `Ucont_alpha`.
`Ucont_beta`	Probability that a nucleotide in a simulated read from a given feature is a U is drawn from a beta distribution with shape2 = `Ucont_beta`.
`feature_prefix`	Name given to the i-th feature is `paste0(feature_prefix, i)`. Shows up in the `feature` column of the output simulated data table.
`dispslope`	Negative binomial dispersion parameter "slope" with respect to read counts. See DESeq2 paper for dispersion model used.
`dispint`	Negative binomial dispersion parameter "intercept" with respect to read counts. See DESeq2 paper for dispersion model used.
`logkdegsdtrend_slope`	Slope for log10(read count) vs. log(kdeg) replicate variability trend
`logkdegsdtrend_intercept`	Intercept for log10(read count) vs. log(kdeg) replicate variability trend
`logksynsdtrend_slope`	Slope for log10(read count) vs. log(ksyn) replicate variability trend
`logksynsdtrend_intercept`	Intercept for log10(read count) vs. log(ksyn) replicate variability trend
`logkdeg_mean`	Mean of normal distribution from which reference log(kdeg) linear model parameter is drawn from for each feature if `param_details` is not provided.

`logkdeg_sd`	Standard deviation of normal distribution from which reference log(kdeg) linear model parameter is drawn from for each feature if `param_details` is not provided.
`logksyn_mean`	Mean of normal distribution from which reference log(ksyn) linear model parameter is drawn from for each feature if `param_details` is not provided.
`logksyn_sd`	Standard deviation of normal distribution from which reference log(ksyn) linear model parameter is drawn from for each feature if `param_details` is not provided.
`logkdeg_diff_avg`	Mean of normal distribution from which non-reference log(kdeg) linear model parameters are drawn from for each feature if `param_details` is not provided.
`logksyn_diff_avg`	Mean of normal distribution from which reference log(ksyn) linear model parameter are drawn from for each feature if `param_details` is not provided.
`logkdeg_diff_sd`	Standard deviation of normal distribution from which reference log(kdeg) linear model parameter are drawn from for each feature if `param_details` is not provided.
`logksyn_diff_sd`	Standard deviation of normal distribution from which reference log(ksyn) linear model parameter are drawn from for each feature if `param_details` is not provided.
`pdiff_kd`	Proportion of features for which non-reference log(kdeg) linear model parameters differ from the reference.
`pdiff_ks`	Proportion of features for which non-reference log(ksyn) linear model parameters differ from the reference.
`pdiff_both`	Proportion of features for which BOTH non-reference log(kdeg) and log(ksyn) linear model parameters differ from the reference. ksyns are simulated
`pdo`	Dropout rate; think of this as the probability that a s4U containing molecule is lost during library preparation and sequencing. If `pdo` is 0 (default) then there is not dropout.

Value

A list containing 6 elements:

- cB: Tibble that can be provided as the `cB` arg to `EZbakRData()`.
- metadf: Tibble that can be provided as the `metadf` arg to `EZbakRData()`.
- PerRepTruth: Tibble containing replicate-by-replicate simulated ground truth
- AvgTruth: Tibble containing average simulated ground truth
- param_details: Tibble containing information about simulated linear model parameters
- UnbiasedFractions: Tibble containing no dropout ground truth

Examples

```
simdata <- SimulateMultiCondition(30,
                                  data.frame(sample = c('sampleA', 'sampleB'),
                                  treatment = c('treatment1', 'treatment2')),
                                  mean_formula = ~treatment-1)
```

SimulateMultiLabel   *Simulate one replicate of multi-label NR-seq data*

Description

Generalizes SimulateOneRep() to simulate any combination of mutation types. Currently, no kinetic model is used to relate certain parameters to the fractions of reads belonging to each simulated mutational population. Instead these fractions are drawn from a Dirichlet distribution with gene-specific parameters.

Usage

```
SimulateMultiLabel(
  nfeatures,
  populations = c("TC"),
  fraction_design = create_fraction_design(populations),
  fractions_matrix = NULL,
  read_vect = NULL,
  sample_name = "sampleA",
  feature_prefix = "Gene",
  kdeg_vect = NULL,
  ksyn_vect = NULL,
  logkdeg_mean = -1.9,
  logkdeg_sd = 0.7,
  logksyn_mean = 2.3,
  logksyn_sd = 0.7,
  phighs = stats::setNames(rep(0.05, times = length(populations)), populations),
  plows = stats::setNames(rep(0.002, times = length(populations)), populations),
  seqdepth = nfeatures * 2500,
  readlength = 200,
  alpha_min = 3,
  alpha_max = 6,
  Ucont = 0.25,
  Acont = 0.25,
  Gcont = 0.25,
  Ccont = 0.25
)
```

Arguments

`nfeatures`	Number of "features" (e.g., genes) to simulate data for
`populations`	Vector of mutation populations you want to simulate.
`fraction_design`	Fraction design matrix, specifying which potential mutational populations should actually exist. See ?EstimateFractions for more details.
`fractions_matrix`	Matrix of fractions of each mutational population to simulate. If not provided, this will be simulated. One row for each feature, one column for each mutational population, rows should sum to 1.
`read_vect`	Vector of length = `nfeatures`; specifies the number of reads to be simulated for each feature. If this is not provided, the number of reads simulated is equal to `round(seqdepth * (ksyn_i/kdeg_i)/sum(ksyn/kdeg))`. In other words, the normalized steady-state abundance of a feature is multiplied by the total number of reads to be simulated and rounded to the nearest integer.
`sample_name`	Character vector to assign to `sample` column of output simulated data table (the cB table).
`feature_prefix`	Name given to the i-th feature is `paste0(feature_prefix, i)`. Shows up in the `feature` column of the output simulated data table.
`kdeg_vect`	Vector of length = `nfeatures`; specifies the degradation rate constant to use for each feature's simulation. If this is not provided and `fn_vect` is, then `kdeg_vect = -log(1 - fn_vect)/label_time`. If both `kdeg_vect` and `fn_vect` are not provided, each feature's `kdeg_vect` value is drawn from a log-normal distrubition with meanlog = `logkdeg_mean` and sdlog = `logkdeg_sd`. `kdeg_vect` is actually only simulated in the case where `read_vect` is also not provided, as it will be used to simulate read counts as described above.
`ksyn_vect`	Vector of length = `nfeatures`; specifies the synthesis rate constant to use for each feature's simulation. If this is not provided, and `read_vect` is also not provided, then each feature's `ksyn_vect` value is drawn from a log-normal distribution with meanlog = `logksyn_mean` and sdlog = `logksyn_sd`. ksyn's do not need to be simulated if `read_vect` is provided, as they only influence read counts.
`logkdeg_mean`	If necessary, meanlog of a log-normal distribution from which kdegs are simulated
`logkdeg_sd`	If necessary, sdlog of a log-normal distribution from which kdegs are simulated
`logksyn_mean`	If necessary, meanlog of a log-normal distribution from which ksyns are simulated
`logksyn_sd`	If necessary, sdlog of a log-normal distribution from which ksyns are simulated
`phighs`	Vector of probabilities of mutation rates in labeled reads of each type denoted in `populations`. Should be a named vector, with names being the corresponding `population`.
`plows`	Vector of probabilities of mutation rates in unlabeled reads of each type denoted in `populations`. Should be a named vector, with names being the corresponding `population`.

`seqdepth`	Only relevant if `read_vect` is not provided; in that case, this is the total number of reads to simulate.
`readlength`	Length of simulated reads. In this simple simulation, all reads are simulated as being exactly this length.
`alpha_min`	Minimum possible value of alpha element of Dirichlet random variable
`alpha_max`	Maximum possible value of alpha element of Dirichlet random variable
`Ucont`	Probability that a nucleotide in a simulated read is a U.
`Acont`	Probability that a nucleotide in a simulated read is an A.
`Gcont`	Probability that a nucleotide in a simulated read is a G.
`Ccont`	Probability that a nucleotide in a simulated read is a C.

Value

List with two elements:

- cB: Tibble that can be passed as the `cB` arg to `EZbakRData()`.
- ground_truth: Tibble containing simulated ground truth.

Examples

```
simdata <- SimulateMultiLabel(3)
```

`SimulateOneRep` *Simulate a single replicate of NR-seq data*

Description

In `SimulateOneRep`, users have the option to either provide vectors of feature-specific read counts, fraction news, kdegs, and ksyns for the simulation, or to have those drawn from relevant distributions whose properties can be tuned by the various optional parameters of `SimulateOneRep`. The number of mutable nucleotides (nT) in a read is drawn from a binomial distribution with `readlength` trials and a probability of "success" equal to `Ucont`. A read's status as new or old is drawn from a Bernoulli distribution with probability of "success" equal to the feature's fraction new. If a read is new, the number of mutations in the read is drawn from a binomial distribution with probability of mutation equal to pnew. If a read is old, the number of mutations is instead drawn from a binomial distribution with probability of mutation equal to pold.

Usage

```
SimulateOneRep(
  nfeatures,
  read_vect = NULL,
  label_time = 2,
  sample_name = "sampleA",
  feature_prefix = "Gene",
  fn_vect = NULL,
  kdeg_vect = NULL,
  ksyn_vect = NULL,
  pnew = 0.05,
  pold = 0.002,
  logkdeg_mean = -1.9,
  logkdeg_sd = 0.7,
  logksyn_mean = 2.3,
  logksyn_sd = 0.7,
  seqdepth = nfeatures * 2500,
  readlength = 200,
  Ucont_alpha = 25,
  Ucont_beta = 75,
  feature_pnew = FALSE,
  pnew_kdeg_corr = FALSE,
  logit_pnew_mean = -2.5,
  logit_pnew_sd = 0.1
)
```

Arguments

`nfeatures`	Number of "features" (e.g., genes) to simulate data for
`read_vect`	Vector of length = `nfeatures`; specifies the number of reads to be simulated for each feature. If this is not provided, the number of reads simulated is equal to `round(seqdepth * (ksyn_i/kdeg_i)/sum(ksyn/kdeg))`. In other words, the normalized steady-state abundance of a feature is multiplied by the total number of reads to be simulated and rounded to the nearest integer.
`label_time`	Length of s^4^U feed to simulate.
`sample_name`	Character vector to assign to `sample` column of output simulated data table (the cB table).
`feature_prefix`	Name given to the i-th feature is `paste0(feature_prefix, i)`. Shows up in the `feature` column of the output simulated data table.
`fn_vect`	Vector of length = `nfeatures`; specifies the fraction new to use for each feature's simulation. If this is not provided and `kdeg_vect` is, then `fn_vect = 1 - exp(-kdeg_vect*label_time)`. If both `fn_vect` and `kdeg_vect` are not provided, then kdegs are simulated from a joint distribution as described below and converted to a `fn_vect` as when `kdeg_vect` is user-provided.
`kdeg_vect`	Vector of length = `nfeatures`; specifies the degradation rate constant to use for each feature's simulation. If this is not provided and `fn_vect` is, then `kdeg_vect = -log(1 - fn_vect)/label_time`. If both `kdeg_vect` and `fn_vect` are not provided, each feature's `kdeg_vect` value is drawn from a log-normal distrubition with meanlog = `logkdeg_mean` and sdlog = `logkdeg_sd`. `kdeg_vect` is actually only simulated in the case where `read_vect` is also not provided, as it will be used to simulate read counts as described above.
`ksyn_vect`	Vector of length = `nfeatures`; specifies the synthesis rate constant to use for each feature's simulation. If this is not provided, and `read_vect` is also not

	provided, then each feature's `ksyn_vect` value is drawn from a log-normal distribution with meanlog = `logksyn_mean` and sdlog = `logksyn_sd`. ksyn's do not need to be simulated if `read_vect` is provided, as they only influence read counts.
`pnew`	Probability that a T is mutated to a C if a read is new.
`pold`	Probability that a T is mutated to a C if a read is old.
`logkdeg_mean`	If necessary, meanlog of a log-normal distribution from which kdegs are simulated
`logkdeg_sd`	If necessary, sdlog of a log-normal distribution from which kdegs are simulated
`logksyn_mean`	If necessary, meanlog of a log-normal distribution from which ksyns are simulated
`logksyn_sd`	If necessary, sdlog of a log-normal distribution from which ksyns are simulated
`seqdepth`	Only relevant if `read_vect` is not provided; in that case, this is the total number of reads to simulate.
`readlength`	Length of simulated reads. In this simple simulation, all reads are simulated as being exactly this length.
`Ucont_alpha`	Probability that a nucleotide in a simulated read from a given feature is a U is drawn from a beta distribution with shape1 = `Ucont_alpha`.
`Ucont_beta`	Probability that a nucleotide in a simulated read from a given feature is a U is drawn from a beta distribution with shape2 = `Ucont_beta`.
`feature_pnew`	Boolean; if TRUE, simulate a different pnew for each feature
`pnew_kdeg_corr`	Boolean; only relevant if `feature_pnew` is TRUE. If so, then setting `pnew_kdeg_corr` to TRUE will ensure that higher kdeg transcripts have a higher pnew.
`logit_pnew_mean`	If `feature_pnew` is TRUE, then the logit(pnew) for each feature will be drawn from a normal distribution with this mean.
`logit_pnew_sd`	If `feature_pnew` is TRUE, then the logit(pnew) for each feature will be drawn from a normal distribution with this standard deviation.

Value

List with two elements:

- cB: Tibble that can be passed as the `cB` arg to `EZbakRData()`.
- ground_truth: Tibble containing simulated ground truth.

Examples

```
simdata <- SimulateOneRep(30)
```

`standard_fraction_design` *Standard* `fraction_design` *table for* `EstimateFractions`

Description

An example `fraction_design` table for a standard NR-seq experiment with s^4U labeling. This table tells `EstimateFractions` that there are two populations of reads, one with high T-to-C mutation content and one with low T-to-C mutation content

Usage

```
standard_fraction_design
```

Format

`standard_fraction_design`**:**

A tibble with 2 rows and 2 columns:

TC Boolean denoting if population represented by that row has high T-to-C mutational content

present Boolean denoting if population represented by that row is expected to be present in this dataset

---

`tilac_fraction_design` *TILAC* `fraction_design` *table for* `EstimateFractions`

Description

An example `fraction_design` table for a TILAC experiment. TILAC was originally described in Courvan et al., 2022. In this method, two populations of RNA, one from s^4U fed cells and one from s^6G fed cells, are pooled and prepped for sequencing together. This allows for internally controlled comparisons of RNA abundance without spike-ins. s^4U is recoded to a cytosine analog by TimeLapse chemistry (or similar chemistry) and s^6G is recoded to an adenine analog. Thus, `fraction_design` includes columns called `TC` and `GA`. A unique aspect of the TILAC `fraction_design` table is that one of the possible populations, `TC` and `GA` both `TRUE`, is denoted as not present (`present` = `FALSE`). This is because there is no RNA was exposed to both s^4U and s^6G, thus a population of reads with both high T-to-C and G-to-A mutational content should not exist.

Usage

```
tilac_fraction_design
```

Format

`tilac_fraction_design`:

A tibble with 4 rows and 3 columns:

TC Boolean denoting if population represented by that row has high T-to-C mutational content

GA Boolean denoting if population represented by that row has high G-to-A mutational content

present Boolean denoting if population represented by that row is expected to be present in this dataset

---

`validate_EZbakRArrowData` `EZbakRArrowData` *EZbakRArrowData validator*

Description

`validate_EZbakRArrowData` ensures that input for `EZbakRArrowData` object construction is valid.

Usage

```
validate_EZbakRArrowData(obj)
```

Arguments

`obj`	An object of class `EZbakRArrowData`

`validate_EZbakRData` `EZbakRDataobject` *validator*

Description

`validate_EZbakRData` ensures that input for `EZbakRData` construction is valid.

Usage

```
validate_EZbakRData(obj)
```

Arguments

`obj`	An object of class `EZbakRData`

validate_EZbakRFractions

`EZbakRFractions` *object validator*

Description

`validate_EZbakRFractions` ensures that input for `EZbakRFractions` construction is valid.

Usage

```
validate_EZbakRFractions(obj)
```

Arguments

`obj`	An object of class `EZbakRFractions`

validate_EZbakRKinetics

`EZbakRKinetics` *object validator*

Description

`validate_EZbakRKinetics` ensures that input for `EZbakRKinetics` construction is valid.

Usage

```
validate_EZbakRKinetics(obj, features)
```

Arguments

`obj`	An object of class `EZbakRKinetics`
`features`	Features tracked in `kinetics` data frame. Needs to be specified explicitly as it cannot be automatically inferred.

VectSimulateMultiLabel

Vectorized simulation of one replicate of multi-label NR-seq data

Description

Generalizes SimulateOneRep() to simulate any combination of mutation types. Currently, no kinetic model is used to relate certain parameters to the fractions of reads belonging to each simulated mutational population. Instead these fractions are drawn from a Dirichlet distribution with gene-specific parameters.

Usage

```
VectSimulateMultiLabel(
  nfeatures,
  populations = c("TC"),
  fraction_design = create_fraction_design(populations),
  fractions_matrix = NULL,
  read_vect = NULL,
  sample_name = "sampleA",
  feature_prefix = "Gene",
  kdeg_vect = NULL,
  ksyn_vect = NULL,
  logkdeg_mean = -1.9,
  logkdeg_sd = 0.7,
  logksyn_mean = 2.3,
  logksyn_sd = 0.7,
  phighs = stats::setNames(rep(0.05, times = length(populations)), populations),
  plows = stats::setNames(rep(0.002, times = length(populations)), populations),
  seqdepth = nfeatures * 2500,
  readlength = 200,
  alpha_min = 3,
  alpha_max = 6,
  Ucont = 0.25,
  Acont = 0.25,
  Gcont = 0.25,
  Ccont = 0.25
)
```

Arguments

`nfeatures`	Number of "features" (e.g., genes) to simulate data for
`populations`	Vector of mutation populations you want to simulate.
`fraction_design`	Fraction design matrix, specifying which potential mutational populations should actually exist. See ?EstimateFractions for more details.

`fractions_matrix`	Matrix of fractions of each mutational population to simulate. If not provided, this will be simulated. One row for each feature, one column for each mutational population, rows should sum to 1.
`read_vect`	Vector of length = `nfeatures`; specifies the number of reads to be simulated for each feature. If this is not provided, the number of reads simulated is equal to `round(seqdepth * (ksyn_i/kdeg_i)/sum(ksyn/kdeg))`. In other words, the normalized steady-state abundance of a feature is multiplied by the total number of reads to be simulated and rounded to the nearest integer.
`sample_name`	Character vector to assign to `sample` column of output simulated data table (the cB table).
`feature_prefix`	Name given to the i-th feature is `paste0(feature_prefix, i)`. Shows up in the `feature` column of the output simulated data table.
`kdeg_vect`	Vector of length = `nfeatures`; specifies the degradation rate constant to use for each feature's simulation. If this is not provided and `fn_vect` is, then `kdeg_vect = -log(1 - fn_vect)/label_time`. If both `kdeg_vect` and `fn_vect` are not provided, each feature's `kdeg_vect` value is drawn from a log-normal distrubition with meanlog = `logkdeg_mean` and sdlog = `logkdeg_sd`. `kdeg_vect` is actually only simulated in the case where `read_vect` is also not provided, as it will be used to simulate read counts as described above.
`ksyn_vect`	Vector of length = `nfeatures`; specifies the synthesis rate constant to use for each feature's simulation. If this is not provided, and `read_vect` is also not provided, then each feature's `ksyn_vect` value is drawn from a log-normal distribution with meanlog = `logksyn_mean` and sdlog = `logksyn_sd`. ksyn's do not need to be simulated if `read_vect` is provided, as they only influence read counts.
`logkdeg_mean`	If necessary, meanlog of a log-normal distribution from which kdegs are simulated
`logkdeg_sd`	If necessary, sdlog of a log-normal distribution from which kdegs are simulated
`logksyn_mean`	If necessary, meanlog of a log-normal distribution from which ksyns are simulated
`logksyn_sd`	If necessary, sdlog of a log-normal distribution from which ksyns are simulated
`phighs`	Vector of probabilities of mutation rates in labeled reads of each type denoted in `populations`. Should be a named vector, with names being the corresponding `population`.
`plows`	Vector of probabilities of mutation rates in unlabeled reads of each type denoted in `populations`. Should be a named vector, with names being the corresponding `population`.
`seqdepth`	Only relevant if `read_vect` is not provided; in that case, this is the total number of reads to simulate.
`readlength`	Length of simulated reads. In this simple simulation, all reads are simulated as being exactly this length.
`alpha_min`	Minimum possible value of alpha element of Dirichlet random variable
`alpha_max`	Maximum possible value of alpha element of Dirichlet random variable

`Ucont`	Probability that a nucleotide in a simulated read is a U.
`Acont`	Probability that a nucleotide in a simulated read is an A.
`Gcont`	Probability that a nucleotide in a simulated read is a G.
`Ccont`	Probability that a nucleotide in a simulated read is a C.

Value

List with two elements:

- cB: Tibble that can be passed as the `cB` arg to `EZbakRData()`.
- ground_truth: Tibble containing simulated ground truth.

Examples

```
simdata <- VectSimulateMultiLabel(30)
```

`VisualizeDropout` *Make plots to visually assess dropout trends*

Description

Plots a measure of dropout (the ratio of -label to +label RPM) as a function of feature fraction new, with the model fit depicted. Use this function to qualitatively assess model fit and whether the modeling assumptions are met.

Usage

```
VisualizeDropout(
  obj,
  plot_type = c("grandR", "bakR"),
  grouping_factors = NULL,
  features = NULL,
  populations = NULL,
  fraction_design = NULL,
  repeatID = NULL,
  exactMatch = TRUE,
  n_min = 50,
  dropout_cutoff = 5,
  ...
)
```

Arguments

`obj`	An EZbakRFractions object, which is an EZbakRData object on which you have run `EstimateFractions()`.
`plot_type`	Which type of plot to make. Options are: • bakR: X-axis is fraction new (a.k.a. NTR) and Y-axis is dropout (no label n / label n) • grandR: X-axis is fraction new rank (a.k.a. NTR rank) and Y-axis is log(dropout)
`grouping_factors`	Which sample-detail columns in the metadf should be used to group -s4U samples by for calculating the average -s4U RPM? The default value of `NULL` will cause all sample-detail columns to be used.
`features`	Character vector of the set of features you want to stratify reads by and estimate proportions of each RNA population. The default of `NULL` will expect there to be only one fractions table in the EZbakRFractions object.
`populations`	Mutational populations that were analyzed to generate the fractions table to use. For example, this would be "TC" for a standard s4U-based nucleotide recoding experiment.
`fraction_design`	"Design matrix" specifying which RNA populations exist in your samples. By default, this will be created automatically and will assume that all combinations of the `mutrate_populations` you have requested to analyze are present in your data. If this is not the case for your data, then you will have to create one manually. See docs for `EstimateFractions` (run ?EstimateFractions()) for more details.
`repeatID`	If multiple `fractions` tables exist with the same metadata, then this is the numerical index by which they are distinguished.
`exactMatch`	If TRUE, then `features` must exactly match the `features` metadata for a given fractions table for it to be used. Means that you cannot specify a subset of features by default. Set this to FALSE if you would like to specify a feature subset.
`n_min`	Minimum raw number of reads to make it to plot
`dropout_cutoff`	Maximum dropout value included in plot.
`...`	Parameters passed to internal `calculate_dropout()` function;

Value

A list of `ggplot2` objects, one for each +label sample.

Examples

```
# Simulate data to analyze
simdata <- EZSimulate(30)

# Create EZbakR input
ezbdo <- EZbakRData(simdata$cB, simdata$metadf)
```

```
# Estimate Fractions
ezbdo <- EstimateFractions(ezbdo)

# Visualize Dropout
ezbdo <- VisualizeDropout(ezbdo)
```

Index