# Package 'hermes'

January 23, 2025

**Title** Preprocessing, analyzing, and reporting of RNA-seq data

**Type** Package

**Date** 2024-06-27

**Version** 1.11.0

**Description** Provides classes and functions for quality control, filtering, normalization and differential expression analysis of pre-processed `RNA-seq` data. Data can be imported from `SummarizedExperiment` as well as `matrix` objects and can be annotated from `BioMart`. Filtering for genes without too low expression or containing required annotations, as well as filtering for samples with sufficient correlation to other samples or total number of reads is supported. The standard normalization methods including cpm, rpkm and tpm can be used, and 'DESeq2` as well as voom differential expression analyses are available.

**License** Apache License 2.0

**URL** https://github.com/insightsengineering/hermes/

**BugReports** https://github.com/insightsengineering/hermes/issues

**Depends** ggfortify, R (>= 4.1), SummarizedExperiment (>= 1.16)

**Imports** assertthat, Biobase, BiocGenerics, biomaRt, checkmate (>= 2.1), circlize, ComplexHeatmap, DESeq2, dplyr, edgeR, EnvStats, forcats (>= 1.0.0), GenomicRanges, ggplot2, ggrepel (>= 0.9), IRanges, lifecycle, limma, magrittr, matrixStats, methods, MultiAssayExperiment, purrr, R6, Rdpack, rlang, S4Vectors, stats, tidyr, utils

**Suggests** BiocStyle, DelayedArray, DT, grid, httr, knitr, rmarkdown, statmod, testthat (>= 2.0), vdiffr

**VignetteBuilder** knitr

**RdMacros** lifecycle, Rdpack

**biocViews** RNASeq, DifferentialExpression, Normalization, Preprocessing, QualityControl

**Encoding** UTF-8

**Language** en-US

**LazyData** true

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.3.0

**Collate** 'GeneSpec-class.R' 'HermesData-validate.R'
'HermesData-class.R' 'HermesData-methods.R'
'argument_convention.R' 'assertthat.R' 'calc_cor.R'
'checkmate.R' 'connections.R' 'data.R' 'differential.R'
'dplyr_compatibility.R' 'draw_barplot.R' 'draw_boxplot.R'
'draw_heatmap.R' 'draw_scatterplot.R' 'graphs.R' 'join_cdisc.R'
'normalization.R' 'package.R' 'pca.R' 'pca_cor_samplevar.R'
'quality.R' 'top_genes.R' 'utils.R'

**git_url** https://git.bioconductor.org/packages/hermes

**git_branch** devel

**git_last_commit** 2f1b4a7

**git_last_commit_date** 2024-10-29

**Repository** Bioconductor 3.21

**Date/Publication** 2025-01-22

**Author** Daniel Sabanés Bové [aut, cre],
Namrata Bhatia [aut],
Stefanie Bienert [aut],
Benoit Falquet [aut],
Haocheng Li [aut],
Jeff Luong [aut],
Lyndsee Midori Zhang [aut],
Alex Richardson [aut],
Simona Rossomanno [aut],
Tim Treis [aut],
Mark Yan [aut],
Naomi Chang [aut],
Chendi Liao [aut],
Carolyn Zhang [aut],
Joseph N. Paulson [aut],
F. Hoffmann-La Roche AG [cph, fnd]

**Maintainer** Daniel Sabanés Bové <daniel.sabanes_bove@rconis.com>

# Contents

---

`hermes-package` `hermes` *Package*

---

## Description

`hermes` facilitates preprocessing, analyzing, and reporting of RNA-seq data.

## Author(s)

**Maintainer**: Daniel Sabanés Bové <daniel.sabanes_bove@rconis.com>

Authors:

- Namrata Bhatia
- Stefanie Bienert
- Benoit Falquet
- Haocheng Li
- Jeff Luong
- Lyndsee Midori Zhang <zhang.lyndsee@gene.com>
- Alex Richardson
- Simona Rossomanno

- Tim Treis
- Mark Yan
- Naomi Chang
- Chendi Liao <chendi.liao@roche.com>
- Carolyn Zhang
- Joseph N. Paulson <paulson.joseph@gene.com>

Other contributors:

- F. Hoffmann-La Roche AG [copyright holder, funder]


## See Also

Useful links:

- https://github.com/insightsengineering/hermes/
- Report bugs at https://github.com/insightsengineering/hermes/issues

---

`add_quality_flags` *Add Quality Flags*

---

## Description

**[Stable]**

The function `add_quality_flags()` adds quality flag information to a `AnyHermesData` object:

- `low_expression_flag`: for each gene, counts how many samples don't pass a minimum expression Counts per Million (`CPM`) threshold. If too many, then it flags this gene as a "low expression" gene.
- `tech_failure_flag`: first calculates the Pearson correlation matrix of the sample wise `CPM` values, resulting in a matrix measuring the correlation between samples. Then compares the average correlation per sample with a threshold - if it is too low, then the sample is flagged as a "technical failure" sample.
- `low_depth_flag`: computes the library size (total number of counts) per sample. If this number is too low, the sample is flagged as a "low depth" sample.

Separate helper functions are internally used to create the flags, and separate `getter` functions allow easy access to the quality control flags in an object.

## Usage

```
add_quality_flags(object, control = control_quality(), overwrite = FALSE)

h_low_expression_flag(object, control = control_quality())

h_low_depth_flag(object, control = control_quality())

h_tech_failure_flag(object, control = control_quality())

get_tech_failure(object)

get_low_depth(object)

get_low_expression(object)
```

## Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input. |
| control | (list)<br>list of settings (thresholds etc.) used to compute the quality control flags, produced by control_quality(). |
| overwrite | (flag)<br>whether previously added flags may be overwritten. |

## Details

While object already has the variables mentioned above as part of the rowData and colData (as this is enforced by the validation method for AnyHermesData), they are usually still NA after the initial object creation.

## Value

The input object with added quality flags.

## Functions

- h_low_expression_flag(): creates the low expression flag for genes given control settings.
- h_low_depth_flag(): creates the low depth (library size) flag for samples given control settings.
- h_tech_failure_flag(): creates the technical failure flag for samples given control settings.
- get_tech_failure(): get the technical failure flags for all samples.
- get_low_depth(): get the low depth failure flags for all samples.
- get_low_expression(): get the low expression failure flags for all genes.

## See Also

- control_quality() for the detailed settings specifications;
- set_tech_failure() to manually flag samples as technical failures.

## Examples

```
# Adding default quality flags to `AnyHermesData` object.
object <- hermes_data
result <- add_quality_flags(object)
which(get_tech_failure(result) != get_tech_failure(object))
head(get_low_expression(result))
head(get_tech_failure(result))
head(get_low_depth(result))

# It is possible to overwrite flags if needed, which will trigger a message.
result2 <- add_quality_flags(result, control_quality(min_cpm = 1000), overwrite = TRUE)

# Separate calculation of low expression flag.
low_expr_flag <- h_low_expression_flag(
  object,
  control_quality(min_cpm = 500, min_cpm_prop = 0.9)
)
length(low_expr_flag) == nrow(object)
head(low_expr_flag)

# Separate calculation of low depth flag.
low_depth_flag <- h_low_depth_flag(object, control_quality(min_depth = 5))
length(low_depth_flag) == ncol(object)
head(low_depth_flag)

# Separate calculation of technical failure flag.
tech_failure_flag <- h_tech_failure_flag(object, control_quality(min_corr = 0.35))
length(tech_failure_flag) == ncol(object)
head(tech_failure_flag)
head(get_tech_failure(object))
head(get_low_depth(object))
head(get_low_expression(object))
```

---

# all_na *Checks Whether All Missing*

---

## Description

Internal function to check whether a whole vector is NA.

## Usage

```
all_na(x)
```

### Arguments

| | |
|---|---|
| x | (vector)<br>vector to check. |

### Value

Corresponding flag.

---

`annotation,AnyHermesData-method`

*Annotation Accessor and Setter*

---

### Description

**[Stable]**

These methods access and set the gene annotations stored in a `AnyHermesData` object.

### Usage

```
## S4 method for signature 'AnyHermesData'
annotation(object, ...)

.row_data_annotation_cols

## S4 replacement method for signature 'AnyHermesData,DataFrame'
annotation(object) <- value
```

### Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>object to access the annotations from. |
| ... | not used. |
| value | (DataFrame)<br>what should the annotations be replaced with. |

### Format

The annotation column names are available in the exported character vector `.row_data_annotation_cols`.

### Value

The `S4Vectors::DataFrame` with the gene annotations:

- `symbol`
- `desc`
- `chromosome`
- `size`

## Note

When trying to replace the required annotations with completely missing values for any genes, a warning will be given and the corresponding gene IDs will be saved in the attribute `annotation.missing.genes`. Note also that additional annotations beyond the required ones may be supplied and will be stored.

## Examples

```
object <- hermes_data
head(annotation(object))
```

---

`assertions` *Additional Assertions for* `assert_that`

---

## Description

**[Experimental]**

We provide additional assertion functions which can be used together with `assertthat::assert_that()`.

**[Experimental]**

We provide additional assertion functions which can be used together with the `checkmate` functions. These are described in individual help pages linked below.

## Usage

```
is_class(x, class2)

is_hermes_data(x)

is_counts_vector(x)

is_list_with(x, elements)

one_provided(one, two)

is_constant(x)
```

## Arguments

| | |
|---|---|
| `x` | an object to check. |
| `class2` | (`character` or class definition)<br>the class to which `x` could belong. |
| `elements` | (`character`)<br>names of elements which should be in the list `x`. |
| `one` | first input. |
| `two` | second input. |

## Value

Depending on the function prefix.

- `assert_` functions return the object invisibly if successful, and otherwise throw an error message.
- `check_` functions return `TRUE` if successful, otherwise a string with the error message.
- `test_` functions just return `TRUE` or `FALSE`.

## Functions

- `is_class()`: checks the class.
- `is_hermes_data()`: checks whether `x` is an `AnyHermesData` object.
- `is_counts_vector()`: checks for a vector of counts (positive integers).
- `is_list_with()`: checks for a list containing elements.
- `one_provided()`: checks that exactly one of the two inputs `one`, `two` is not `NULL`.
- `is_constant()`: checks whether the vector `x` is constant (only supports `numeric`, `factor`, `character`, `logical`). `NA`s are removed first.

## See Also

`assert_proportion()`

## Examples

```
# Assert a general class.
a <- 5
is_class(a, "character")

# Assert a `AnyHermesData` object.
is_hermes_data(hermes_data)
is_hermes_data(42)

# Assert a counts vector.
a <- 5L
is_counts_vector(a)

# Assert a list containing certain elements.
b <- list(a = 5, b = 3)
is_list_with(b, c("a", "c"))
is_list_with(b, c("a", "b"))

# Assert that exactly one of two arguments is provided.
a <- 10
b <- 10
one_provided(a, b)
one_provided(a, NULL)

# Assert a constant vector.
is_constant(c(1, 2))
```

```
is_constant(c(NA, 1))
is_constant(c("a", "a"))
is_constant(factor(c("a", "a")))
```

---

`assertion_arguments` *Standard Assertion Arguments*

---

## Description

The documentation to this function lists all the conventional arguments in additional `checkmate` assertions.

## Arguments

| | |
|---|---|
| `x` | an object to check. |
| `null.ok` | (`flag`)<br>whether `x` may also be `NULL`. |
| `.var.name` | (`string`)<br>name of the checked object to print in assertions; defaults to the heuristic implemented in `checkmate::vname()`. |
| `add` | (`AssertCollection` or `NULL`)<br>collection to store assertion messages, see `checkmate::AssertCollection`. |
| `info` | (`string`)<br>extra information to be included in the message for the `testthat` reporter, see `testthat::expect_that()`. |
| `label` | (`string`)<br>name of the checked object to print in messages. Defaults to the heuristic implemented in `checkmate::vname()`. |

---

`autoplot,AnyHermesData-method`

*All Standard Plots in Default Setting*

---

## Description

**[Experimental]**

This generates all standard plots - histogram and q-q plot of library sizes, density plot of the (log) counts distributions, boxplot of the number of number of non-zero expressed genes per sample, and a stacked barplot of low expression genes by chromosome at default setting.

## Usage

```
## S4 method for signature 'AnyHermesData'
autoplot(object)
```

### Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input. |

### Value

A list with the ggplot objects from draw_libsize_hist(), draw_libsize_qq(), draw_libsize_densities(), draw_nonzero_boxplot() and draw_genes_barplot() functions with default settings.

### Examples

```
result <- hermes_data
autoplot(result)
```

---

## calc_pca *Principal Components Analysis Calculation*

---

### Description

**[Experimental]**

The calc_pca() function performs principal components analysis of the gene count vectors across all samples.

A corresponding autoplot() method then can visualize the results.

### Usage

```
calc_pca(object, assay_name = "counts", n_top = NULL)
```

### Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input. |
| assay_name | (string)<br>name of the assay to use. |
| n_top | (count or NULL)<br>filter criteria based on number of genes with maximum variance. |

### Details

- PCA should be performed after filtering out low quality genes and samples, as well as normalization of counts.
- In addition, genes with constant counts across all samples are excluded from the analysis internally in calc_pca(). Centering and scaling is also applied internally.
- Plots can be obtained with the ggplot2::autoplot() function with the corresponding method from the ggfortify package to plot the results of a principal components analysis saved in a HermesDataPca object. See ggfortify::autoplot.prcomp() for details.

## Value

A HermesDataPca object which is an extension of the stats::prcomp class.

## See Also

Afterwards correlations between principal components and sample variables can be calculated, see `pca_cor_samplevar`.

## Examples

```
object <- hermes_data %>%
  add_quality_flags() %>%
  filter() %>%
  normalize()

result <- calc_pca(object, assay_name = "tpm")
summary(result)

result1 <- calc_pca(object, assay_name = "tpm", n_top = 500)
summary(result1)

# Plot the results.
autoplot(result)
autoplot(result, x = 2, y = 3)
autoplot(result, variance_percentage = FALSE)
autoplot(result, label = TRUE, label.repel = TRUE)
```

---

# cat_with_newline *Concatenate and Print with Newline*

---

## Description

**[Experimental]**

This function concatenates inputs like `cat()` and prints them with newline.

## Usage

```
cat_with_newline(...)
```

## Arguments

| | |
|---|---|
| ... | inputs to concatenate. |

## Value

None, only used for the side effect of producing the concatenated output in the R console.

## See Also

This is similar to cli::cat_line().

## Examples

```
cat_with_newline("hello", "world")
```

---

cbind *Column Binding of* AnyHermesData *Objects*

---

## Description

**[Stable]**

This method combines AnyHermesData objects with the same ranges but different samples (columns in assays).

## Arguments

| | |
|---|---|
| ... | (AnyHermesData)<br>objects to column bind. |

## Value

The combined AnyHermesData object.

## Note

- Note that this just inherits SummarizedExperiment::cbind,SummarizedExperiment-method(). When binding a AnyHermesData object with a SummarizedExperiment::SummarizedExperiment object, then the result will be a SummarizedExperiment::SummarizedExperiment object (the more general class).
- Note that the combined object needs to have unique sample IDs (column names).

## See Also

rbind to row bind objects.

## Examples

```
a <- hermes_data[, 1:10]
b <- hermes_data[, 11:20]
result <- cbind(a, b)
class(result)
```

check_proportion *Check for proportion*

## Description

**[Experimental]**

Check whether x is a (single) proportion.

## Usage

```
check_proportion(x, null.ok = FALSE)

assert_proportion(
  x,
  null.ok = FALSE,
  .var.name = checkmate::vname(x),
  add = NULL
)

test_proportion(x, null.ok = FALSE)

expect_proportion(x, null.ok = FALSE, info = NULL, label = vname(x))
```

## Arguments

| | |
|---|---|
| x | an object to check. |
| null.ok | (flag)<br>whether x may also be NULL. |
| .var.name | (string)<br>name of the checked object to print in assertions; defaults to the heuristic implemented in checkmate::vname(). |
| add | (AssertCollection or NULL)<br>collection to store assertion messages, see checkmate::AssertCollection. |
| info | (string)<br>extra information to be included in the message for the testthat reporter, see testthat::expect_that(). |
| label | (string)<br>name of the checked object to print in messages. Defaults to the heuristic implemented in checkmate::vname(). |

## Value

TRUE if successful, otherwise a string with the error message.

## See Also

assertions for more details.

## Examples

```
check_proportion(0.25)
```

---

colMeanZscores *Mean Z-score Gene Signature*

---

## Description

**[Experimental]**

This helper function returns the Z-score from an assay stored as a matrix.

## Usage

```
colMeanZscores(x)
```

## Arguments

| | |
|---|---|
| x | (matrix)<br>containing numeric data with genes in rows and samples in columns, no missing values are allowed. |

## Value

A numeric vector containing the mean Z-score values for each column in x.

## Examples

```
object <- hermes_data %>%
  add_quality_flags() %>%
  filter() %>%
  normalize() %>%
  assay("counts")

colMeanZscores(object)
```

## colPrinComp1 *First Principal Component (PC1) Gene Signature*

### Description

**[Experimental]**

This helper function returns the first principal component from an assay stored as a `matrix`.

### Usage

```
colPrinComp1(x, center = TRUE, scale = TRUE)
```

### Arguments

| | |
|---|---|
| `x` | (`matrix`)<br>containing numeric data with genes in rows and samples in columns, no missing values are allowed. |
| `center` | (`flag`)<br>whether the variables should be zero centered. |
| `scale` | (`flag`)<br>whether the variables should be scaled to have unit variance. |

### Value

A numeric vector containing the principal component values for each column in `x`.

### Examples

```
object <- hermes_data %>%
  add_quality_flags() %>%
  filter() %>%
  normalize() %>%
  assay("counts")

colPrinComp1(object)
```

## col_data_with_genes *Sample Variables with Selected Gene Information*

### Description

**[Experimental]**

This obtains the sample variables of a `HermesData` object together with selected gene information.

## Usage

```
col_data_with_genes(object, assay_name, genes)
```

## Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input experiment. |
| assay_name | (string)<br>which assay to use. |
| genes | (GeneSpec)<br>which genes or which gene signature should be extracted. |

## Value

The combined data set, where the additional attribute gene_cols contains the names of the columns obtained by extracting the genes information.

## Note

The class of the returned data set will depend on the class of colData, so usually will be S4Vectors::DFrame.

## Examples

```
result <- col_data_with_genes(hermes_data, "counts", gene_spec("GeneID:1820"))
tail(names(result))
result$GeneID.1820
```

---

# connect_biomart *Connection to* BioMart

---

## Description

**[Experimental]**

connect_biomart() creates a connection object of class ConnectionBiomart which contains the biomaRt object of class biomaRt::Mart and the prefix of the object which is used downstream for the query.

## Usage

```
connect_biomart(prefix = c("ENSG", "GeneID"), version = NULL)
```

## Arguments

| | |
|---|---|
| prefix | (string)<br>gene ID prefix. |
| version | (string or NULL)<br>optional Ensembl version to use. If NULL the latest available release is used. |

### Details

This connects to the Ensembl data base of BioMart for human genes. A specific version can be optionally chosen to ensure reproducibility of results once a new release is available, as accessed data might then change.

### Value

ConnectionBiomart object.

### Examples

```
if (interactive()) {
  connection <- connect_biomart("ENSG")
}
```

---

## control_normalize *Control Settings for Counts Normalization*

---

### Description

**[Stable]**

This control function allows for easy customization of the normalization settings.

### Usage

```
control_normalize(
  log = TRUE,
  lib_sizes = NULL,
  prior_count = 1,
  fit_type = "parametric"
)
```

### Arguments

| | |
|---|---|
| log | (flag)<br>whether log2 values are returned, otherwise original scale is used. |
| lib_sizes | (NULL or counts)<br>library sizes, if NULL the vector with the sum of the counts for each of the samples will be used. |
| prior_count | (non-negative number)<br>average count to be added to each observation to avoid taking log of zero, used only when log = TRUE. |
| fit_type | (string)<br>method to estimate dispersion parameters in Negative Binomial model, used only when normalize() methods include vst and/or rlog. See estimateDispersions for details. |

## Value

List with the above settings used to perform the normalization procedure.

## Note

To be used with the `normalize()` function.

## Examples

```
control_normalize()
control_normalize(log = FALSE, lib_sizes = rep(1e6L, 20))
```

---

`control_quality` *Control for Specified Quality Flags*

---

## Description

**[Stable]**

Control function which specifies the quality flag settings. One or more settings can be customized. Not specified settings are left at defaults.

## Usage

```
control_quality(
  min_cpm = 1,
  min_cpm_prop = 0.25,
  min_corr = 0.5,
  min_depth = NULL
)
```

## Arguments

| | |
|---|---|
| `min_cpm` | (non-negative `number`)<br>minimum Counts per Million (`CPM`) for each gene within the sample. |
| `min_cpm_prop` | (`proportion`)<br>minimum proportion of samples with acceptable `CPM` of certain gene for low expression flagging. |
| `min_corr` | (`proportion`)<br>minimum Pearson correlation coefficient of `CPM` between samples for technical failure flagging. |
| `min_depth` | (non-negative `count` or `NULL`)<br>minimum library depth for low depth flagging. If `NULL`, this will be calculated as the first quartile minus 1.5 times the inter-quartile range of the library size (depth) of all samples. (So anything below the usual lower boxplot whisker would be too low.) |

## Value

List with the above criteria to flag observations.

## Note

To be used with the add_quality_flags() function.

## Examples

```
# Default settings.
control_quality()

# One or more settings can be customized.
control_quality(min_cpm = 5, min_cpm_prop = 0.001)
```

---

# correlate *Generic Function for Correlation Calculations*

---

## Description

**[Experimental]**

New generic function to calculate correlations for one or two objects.

## Usage

```
correlate(object, ...)
```

## Arguments

| | |
|---|---|
| object | input of which the class will be used to decide the method. |
| ... | additional arguments. |

## Value

Corresponding object that contains the correlation results.

## See Also

pca_cor_samplevar and calc_cor which are the methods included for this generic function.

## Examples

```
sample_cors <- correlate(hermes_data)
autoplot(sample_cors)

pca_sample_var_cors <- correlate(calc_pca(hermes_data), hermes_data)
autoplot(pca_sample_var_cors)
```

correlate,AnyHermesData-method
*Correlation between Sample Counts of* AnyHermesData

## Description

**[Experimental]**

The `correlate()` method can calculate the correlation matrix between the sample vectors of counts from a specified assay. This produces a `HermesDataCor` object, which is an extension of a `matrix` with additional quality flags in the slot `flag_data` (containing the `tech_failure_flag` and `low_depth_flag` columns describing the original input samples).

An `autoplot()` method then afterwards can produce the corresponding heatmap.

## Usage

```
## S4 method for signature 'AnyHermesData'
correlate(object, assay_name = "counts", method = "pearson", ...)

## S4 method for signature 'HermesDataCor'
autoplot(
  object,
  flag_colors = c(`FALSE` = "green", `TRUE` = "red"),
  cor_colors = circlize::colorRamp2(c(0, 0.5, 1), c("red", "yellow", "green")),
  ...
)
```

## Arguments

| | |
|---|---|
| `object` | (`AnyHermesData`)<br>object to calculate the correlation. |
| `assay_name` | (`string`)<br>the name of the assay to use. |
| `method` | (`string`)<br>the correlation method, see `stats::cor()` for details. |
| `...` | other arguments to be passed to `ComplexHeatmap::Heatmap()`. |
| `flag_colors` | (named `character`)<br>a vector that specifies the colors for `TRUE` and `FALSE` flag values. |
| `cor_colors` | (`function`)<br>color scale function for the correlation values in the heatmap, produced by `circlize::colorRamp2()`. |

## Value

A `HermesDataCor` object.

### Functions

- `autoplot(HermesDataCor)`: This `autoplot()` method uses the `ComplexHeatmap::Heatmap()` function to plot the correlations between samples saved in a `HermesDataCor` object.

### Examples

```
object <- hermes_data

# Calculate the sample correlation matrix.
correlate(object)

# We can specify another correlation coefficient to be calculated.
result <- correlate(object, method = "spearman")

# Plot the correlation matrix.
autoplot(result)

# We can customize the heatmap.
autoplot(result, show_column_names = FALSE, show_row_names = FALSE)

# Including changing the axis label text size.
autoplot(
  result,
  row_names_gp = grid::gpar(fontsize = 8),
  column_names_gp = grid::gpar(fontsize = 8)
)
```

---

## correlate,HermesDataPca-method

*Correlation of Principal Components with Sample Variables*

---

### Description

**[Stable]**

This `correlate()` method analyses the correlations (in R2 values) between all sample variables in a `AnyHermesData` object and the principal components of the samples.

A corresponding `autoplot()` method then can visualize the results in a heatmap.

### Usage

```
## S4 method for signature 'HermesDataPca'
correlate(object, data)

## S4 method for signature 'HermesDataPcaCor'
autoplot(
  object,
  cor_colors = circlize::colorRamp2(c(-1, 0, 1), c("blue", "white", "red")),
  ...
)
```

## Arguments

| | |
|---|---|
| object | (HermesDataPca)<br>input. It can be generated using calc_pca() function on AnyHermesData. |
| data | (AnyHermesData)<br>input that was used originally for the PCA. |
| cor_colors | (function)<br>color scale function for the correlation values in the heatmap, produced by circlize::colorRamp2(). |
| ... | other arguments to be passed to ComplexHeatmap::Heatmap(). |

## Value

A HermesDataPcaCor object with R2 values for all sample variables.

## Functions

- autoplot(HermesDataPcaCor): This plot method uses the ComplexHeatmap::Heatmap() function to visualize a HermesDataPcaCor object.

## See Also

h_pca_df_r2_matrix() which is used internally for the details.

## Examples

```
object <- hermes_data %>%
  add_quality_flags() %>%
  filter() %>%
  normalize()

# Perform PCA and then correlate the prinicipal components with the sample variables.
object_pca <- calc_pca(object)
result <- correlate(object_pca, object)

# Visualize the correlations in a heatmap.
autoplot(result)

# We can also choose to not reorder the columns.
autoplot(result, cluster_columns = FALSE)

# We can also choose break-points for color customization.
autoplot(
  result,
  cor_colors = circlize::colorRamp2(
    c(-0.5, -0.25, 0, 0.25, 0.5, 0.75, 1),
    c("blue", "green", "purple", "yellow", "orange", "red", "brown")
  )
)
```

counts,AnyHermesData-method
*Counts Accessor and Setter*

## Description

**[Stable]**

These methods access and set the counts assay in a AnyHermesData object.

## Usage

```
## S4 method for signature 'AnyHermesData'
counts(object, ...)

## S4 replacement method for signature 'AnyHermesData,matrix'
counts(object, ..., withDimnames = TRUE) <- value
```

## Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>object to access the counts from. |
| ... | not used. |
| withDimnames | (flag)<br>setting withDimnames =FALSE in the setter (counts<-) is required when the dimnames on the supplied counts assay are not identical to the dimnames on the AnyHermesData object; it does not influence actual assignment of dimnames to the assay (they're always stored as-is). |
| value | (matrix)<br>what should the counts assay be replaced with. |

## Value

The counts assay.

## Methods (by class)

- counts(object = AnyHermesData) <- value:

## Examples

```
a <- hermes_data
result <- counts(a)
class(result)
head(result)
counts(a) <- counts(a) + 100L
head(counts(a))
```

cut_quantile *Cutting a Numeric Vector into a Factor of Quantile Bins*

## Description

**[Experimental]**

This function transforms a numeric vector into a factor corresponding to the quantile intervals. The intervals are left-open and right-closed.

## Usage

```
cut_quantile(x, percentiles = c(1/3, 2/3), digits = 0)
```

## Arguments

| | |
|---|---|
| x | (numeric)<br>the continuous variable values which should be cut into quantile bins. NA values are not taken into account when computing quantiles and are attributed to the NA interval. |
| percentiles | (proportions)<br>the required percentiles for the quantile intervals to be generated. Duplicated values are removed. |
| digits | (integer)<br>the precision to use when formatting the percentages. |

## Value

The factor with a description of the available quantiles as levels.

## Examples

```
set.seed(452)
x <- runif(10, -10, 10)
cut_quantile(x, c(0.33333333, 0.6666666), digits = 4)

x[1:4] <- NA
cut_quantile(x)
```

df_cols_to_factor *Conversion of Eligible Columns to Factor Variables in a* DataFrame

## Description

**[Experimental]**

This utility function converts all eligible character and logical variables in a S4Vectors::DataFrame to factor variables. All factor variables get amended with an explicit missing level. This includes both NA and empty strings.

## Usage

```
df_cols_to_factor(data, omit_columns = NULL, na_level = "<Missing>")
```

## Arguments

| | |
|---|---|
| data | (DataFrame)<br>input S4Vectors::DataFrame. |
| omit_columns | (character or NULL)<br>which columns should be omitted from the possible conversion to factor and explicit missing level application. |
| na_level | (string)<br>explicit missing level to be used for factor variables. |

## Value

The modified data.

## Note

All required rowData and colData variables cannot be converted to ensure proper downstream behavior. These are automatically omitted if found in data and therefore do not need to be specified in omit_columns.

## Examples

```
dat <- colData(summarized_experiment)
any(vapply(dat, is.character, logical(1)))
any(vapply(dat, is.logical, logical(1)))
dat_converted <- df_cols_to_factor(dat)
any(vapply(dat_converted, function(x) is.character(x) || is.logical(x), logical(1)))
```

`diff_expression` *Differential Expression Analysis*

## Description

**[Experimental]**

The `diff_expression()` function performs differential expression analysis using a method of preference.

A corresponding `autoplot()` method is visualizing the results as a volcano plot.

## Usage

```
diff_expression(object, group, method = c("voom", "deseq2"), ...)

## S4 method for signature 'HermesDataDiffExpr'
autoplot(object, adj_p_val_thresh = 0.05, log2_fc_thresh = 2.5)
```

## Arguments

| | |
|---|---|
| `object` | (`AnyHermesData`)<br>input. Note that this function only uses the original counts for analysis, so this does not need to be normalized. |
| `group` | (`string`)<br>name of factor variable with 2 levels in `colData(object)`. These 2 levels will be compared in the differential expression analysis. |
| `method` | (`string`)<br>method for differential expression analysis, see details below. |
| `...` | additional arguments passed to the helper function associated with the selected method. |
| `adj_p_val_thresh` | (`proportion`)<br>threshold on the adjusted p-values (y-axis) to flag significance. |
| `log2_fc_thresh` | (`number`)<br>threshold on the absolute log2 fold-change (x-axis) to flag up- or down-regulation of transcription. |

## Details

Possible method choices are:

- voom: uses `limma::voom()`, see `h_diff_expr_voom()` for details.
- deseq2: uses `DESeq2::DESeq()`, see `h_diff_expr_deseq2()` for details.

## Value

A HermesDataDiffExpr object which is a data frame with the following columns for each gene in the HermesData object:

- log2_fc (the estimate of the log2 fold change between the 2 levels of the provided factor)
- stat (the test statistic, which one depends on the method used)
- p_val (the raw p-value)
- adj_p_val (the multiplicity adjusted p-value value)

## Functions

- autoplot(HermesDataDiffExpr): generates a volcano plot for a HermesDataDiffExpr object.

## Note

- We provide the df_cols_to_factor() utility function that makes it easy to convert the colData() character and logical variables to factors, so that they can be subsequently used as group inputs. See the example.
- In order to avoid a warning when using deseq2, it can be necessary to specify fitType = "local" as additional argument. This could e.g. be the case when only few samples are present in which case the default parametric dispersions estimation will not work.

## Examples

```
object <- hermes_data %>%
  add_quality_flags() %>%
  filter()

# Convert character and logical to factor variables in `colData`,
# including the below used `group` variable.
colData(object) <- df_cols_to_factor(colData(object))
res1 <- diff_expression(object, group = "SEX", method = "voom")
head(res1)
res2 <- diff_expression(object, group = "SEX", method = "deseq2")
head(res2)

# Pass method arguments to the internally used helper functions.
res3 <- diff_expression(object, group = "SEX", method = "voom", robust = TRUE, trend = TRUE)
head(res3)
res4 <- diff_expression(object, group = "SEX", method = "deseq2", fitType = "local")
head(res4)

# Create the corresponding volcano plots.
autoplot(res1)
autoplot(res3)
```

draw_barplot *Barplot for Gene Expression Percentiles*

## Description

**[Experimental]**

This produces a barplot of the dichotomized gene expression counts into two or three categories based on custom defined percentiles.

## Usage

```
draw_barplot(
  object,
  assay_name,
  x_spec,
  facet_var = NULL,
  fill_var = NULL,
  percentiles = c(1/3, 2/3)
)
```

## Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input. |
| assay_name | (string)<br>selects assay from input. |
| x_spec | (GeneSpec)<br>gene specification for the x-axis. |
| facet_var | (string or NULL)<br>optional faceting variable, taken from input sample variables. |
| fill_var | (string or NULL)<br>optional fill variable, taken from input sample variables. |
| percentiles | (vector)<br>lower and upper percentiles to dichotomize the gene counts into two or three categories. |

## Value

The ggplot barplot.

## Examples

```
object <- hermes_data

g <- genes(object)
```

```
draw_barplot(
  object,
  assay_name = "counts",
  x_spec = gene_spec(g[1]),
  facet_var = "SEX",
  fill_var = "AGE18"
)

draw_barplot(
  object,
  assay_name = "counts",
  x_spec = gene_spec(g[1:3], colMedians, "Median"),
  facet_var = "SEX",
  fill_var = "AGE18"
)

draw_barplot(
  object,
  assay_name = "counts",
  x_spec = gene_spec(g[1:3], colMeans, "Mean"),
  facet_var = "SEX",
  fill_var = "AGE18",
  percentiles = c(0.1, 0.9)
)
```

---

`draw_boxplot` *Boxplot for Gene Expression Values*

---

## Description

**[Experimental]**

This produces boxplots of the gene expression values of a single gene, multiple genes or a gene signature.

## Usage

```
draw_boxplot(
  object,
  assay_name,
  genes,
  x_var = NULL,
  color_var = NULL,
  facet_var = NULL,
  violin = FALSE,
  jitter = FALSE
)

h_draw_boxplot_df(object, assay_name, genes, x_var, color_var, facet_var)
```

## Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input. |
| assay_name | (string)<br>selects assay from input for the y-axis. |
| genes | (GeneSpec)<br>for which genes or which gene signature to produce boxplots. |
| x_var | (string or NULL)<br>optional stratifying variable for the x-axis, taken from input sample variables. |
| color_var | (string or NULL)<br>optional color variable, taken from input sample variables. |
| facet_var | (string or NULL)<br>optional faceting variable, taken from input sample variables. |
| violin | (flag)<br>whether to draw a violin plot instead of a boxplot. |
| jitter | (flag)<br>whether to add jittered original data points. |

## Value

The ggplot boxplot.

## Functions

- h_draw_boxplot_df(): Helper function to prepare the data frame required for plotting.

## Examples

```
object <- hermes_data
draw_boxplot(
  object,
  assay_name = "counts",
  genes = gene_spec(c(A = genes(object)[1])),
  violin = TRUE
)

object2 <- object %>%
  add_quality_flags() %>%
  filter() %>%
  normalize()
draw_boxplot(
  object2,
  assay_name = "tpm",
  x_var = "SEX",
  genes = gene_spec(setNames(genes(object2)[1:10], 1:10), fun = colMeans),
  facet_var = "RACE",
  color_var = "AGE18",
  jitter = TRUE
```

```
)

draw_boxplot(
  object,
  assay_name = "counts",
  x_var = "SEX",
  genes = gene_spec(genes(object)[1:3]),
  jitter = TRUE,
  facet_var = "AGE18"
)

draw_boxplot(
  object,
  assay_name = "counts",
  genes = gene_spec(c(A = "GeneID:11185", B = "GeneID:10677")),
  violin = TRUE
)
```

---

`draw_genes_barplot` *Stacked Barplot of Low Expression Genes by Chromosome*

---

## Description

**[Experimental]**

This creates a barplot of chromosomes for the AnyHermesData object with the proportions of low expression genes.

## Usage

```
draw_genes_barplot(
  object,
  chromosomes = c(seq_len(22), "X", "Y", "MT"),
  include_others = TRUE
)
```

## Arguments

| | |
|---|---|
| `object` | (`AnyHermesData`)<br>input. |
| `chromosomes` | (`character`)<br>names of the chromosomes which should be displayed. |
| `include_others` | (`flag`)<br>option to show the chromosomes not in `chromosomes` as "Others". |

## Value

The `ggplot` object with the histogram.

## Examples

```
object <- hermes_data

# Display chromosomes 1-22, X, Y, and MT. Other chromosomes are displayed in "Others".
# To increase readability, we can have flip the coordinate axes.
draw_genes_barplot(object) + coord_flip()

# Alternatively we can also rotate the x-axis tick labels.
draw_genes_barplot(object) + theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1))

# Display chromosomes 1 and 2. Other chromosomes are displayed in "Others".
draw_genes_barplot(object, chromosomes = c("1", "2"))

# Display chromosomes 1 and 2 only.
draw_genes_barplot(object, chromosomes = c("1", "2"), include_others = FALSE)
```

---

# draw_heatmap *Heatmap for Gene Expression Counts*

---

## Description

**[Experimental]**

This produces a heatmap of the chosen assay and groups by various sample variables.

## Usage

```
draw_heatmap(
  object,
  assay_name,
  color_extremes = c(0.01, 0.99),
  col_data_annotation = NULL,
  ...
)
```

## Arguments

| | |
|---|---|
| `object` | (`AnyHermesData`)<br>input. |
| `assay_name` | (`string`)<br>selects assay from input. |
| `color_extremes` | (`numeric`)<br>min and max percentiles to inform the color scheme of the heatmap as blue and red respectively. |
| `col_data_annotation` | (`character` or `NULL`)<br>optional grouping variable(s), taken from input sample variables. |
| `...` | additional arguments to pass to `ComplexHeatmap::Heatmap()`. |

## Value

The ComplexHeatmap::Heatmap heatmap

## Examples

```
result <- hermes_data %>%
  normalize(methods = "voom") %>%
  add_quality_flags() %>%
  filter(what = "genes")

draw_heatmap(
  object = result[1:10, ],
  assay_name = "counts",
  col_data_annotation = "COUNTRY"
)

draw_heatmap(
  object = result[1:10, ],
  assay_name = "counts",
  color_extremes = c(0.001, 0.999),
  col_data_annotation = "AGEGRP"
)
```

---

# draw_libsize_densities

*Density Plot of (Log) Counts Distributions*

---

## Description

**[Experimental]**

This creates a density plot of the (log) counts distributions of the AnyHermesData object where each line on the plot corresponds to a sample.

## Usage

```
draw_libsize_densities(object, log = TRUE)
```

## Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input. |
| log | (flag)<br>should the counts be log transformed (log2). |

## Value

The ggplot object with the density plot.

## Examples

```
result <- hermes_data
draw_libsize_densities(result)
draw_libsize_densities(result, log = FALSE)
```

---

`draw_libsize_hist` *Histogram of Library Sizes*

---

## Description

**[Experimental]**

This creates a histogram of the library sizes of the AnyHermesData object.

## Usage

```
draw_libsize_hist(object, bins = 30L, fill = "darkgrey")
```

## Arguments

| | |
|---|---|
| `object` | (`AnyHermesData`)<br>input. |
| `bins` | (`count`)<br>number of evenly distributed groups desired. |
| `fill` | (`string`)<br>color of the bars filling. |

## Value

The `ggplot` object with the histogram.

## Examples

```
result <- hermes_data
draw_libsize_hist(result)
draw_libsize_hist(result, bins = 10L, fill = "blue")
```

## draw_libsize_qq *Q-Q Plot of Library Sizes*

### Description

**[Experimental]**

This creates a Q-Q plot of the library sizes of the AnyHermesData object.

### Usage

```
draw_libsize_qq(object, color = "grey", linetype = "dashed")
```

### Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input. |
| color | (string)<br>color of Q-Q line. |
| linetype | (string)<br>line type of Q-Q line. |

### Value

The ggplot object with the Q-Q Plot.

### Examples

```
result <- hermes_data
draw_libsize_qq(result)
draw_libsize_qq(result, color = "blue", linetype = "solid")

# We can also add sample names as labels.
library(ggrepel)
draw_libsize_qq(result) + geom_text_repel(label = colnames(result), stat = "qq")
```

## draw_nonzero_boxplot *Boxplot of Non-Zero Genes*

### Description

**[Experimental]**

This draws a boxplot, with overlaid data points, of the number of non-zero expressed genes per sample.

### Usage

```
draw_nonzero_boxplot(object, position = position_jitter(0.2), alpha = 0.25)
```

### Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input. |
| position | (Position)<br>specifies x-axis position of points, e.g. for jittering. |
| alpha | (proportion)<br>specifies transparency of points. |

### Value

The ggplot object with the boxplot.

### Examples

```
# Default boxplot.
result <- hermes_data
draw_nonzero_boxplot(result)

# Reusing the same position for labeling.
library(ggrepel)
pos <- position_jitter(0.5)
draw_nonzero_boxplot(result, position = pos) +
  geom_text_repel(aes(label = samples(result)), position = pos)
```

---

## draw_scatterplot *Scatterplot for Gene Expression Values*

---

### Description

**[Experimental]**

This produces a scatterplot of two genes or gene signatures.

### Usage

```
draw_scatterplot(
  object,
  assay_name,
  x_spec,
  y_spec,
  color_var = NULL,
  facet_var = NULL,
  smooth_method = c("lm", "loess", "none")
)
```

## Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input. |
| assay_name | (string)<br>selects assay from input. |
| x_spec | (GeneSpec)<br>gene specification for the x-axis. |
| y_spec | (GeneSpec)<br>gene specification for the y-axis. |
| color_var | (string or NULL)<br>optional color variable, taken from input sample variables. |
| facet_var | (string or NULL)<br>optional faceting variable, taken from input sample variables. |
| smooth_method | (string)<br>smoothing method to use, either linear regression line (lm), local polynomial regression (loess) or none. |

## Value

The ggplot scatterplot.

## Examples

```
object <- hermes_data
g <- genes(object)

draw_scatterplot(
  object,
  assay_name = "counts",
  facet_var = NULL,
  x_spec = gene_spec(c(A = g[1])),
  y_spec = gene_spec(g[2]),
  color = "RACE"
)

object2 <- object %>%
  add_quality_flags() %>%
  filter() %>%
  normalize()
g2 <- genes(object2)

draw_scatterplot(
  object2,
  assay_name = "tpm",
  facet_var = "SEX",
  x_spec = gene_spec(g2[1:10], colMeans, "Mean"),
  y_spec = gene_spec(g2[11:20], colMedians, "Median"),
  smooth_method = "loess"
)
```

## expression_set *Example* ExpressionSet *Data*

### Description

**[Stable]**

This example data can be used to try out conversion of a Biobase::ExpressionSet object into a HermesData object.

### Usage

```
expression_set
```

### Format

A Biobase::ExpressionSet object with 20 samples covering 5085 features (Entrez gene IDs).

### Source

This is an artificial dataset designed to resemble real data.

### See Also

- SummarizedExperiment::makeSummarizedExperimentFromExpressionSet() to convert into a SummarizedExperiment::SummarizedExperiment.
- summarized_experiment which contains similar data already as a SummarizedExperiment::SummarizedExperiment

## extra_data_names *Extra Variable Names Accessor Methods*

### Description

**[Experimental]**

The methods access the names of the variables in colData() and rowData() of the object which are not required by design. So these can be additional sample or patient characteristics, or gene characteristics.

## Usage

```
extraColDataNames(x, ...)

## S4 method for signature 'AnyHermesData'
extraColDataNames(x, ...)

extraRowDataNames(x, ...)

## S4 method for signature 'AnyHermesData'
extraRowDataNames(x, ...)
```

## Arguments

| | |
|---|---|
| x | (AnyHermesData) object. |
| ... | not used. |

## Value

The character vector with the additional variable names in either `colData()` or `rowData()`.

## Examples

```
object <- hermes_data
extraColDataNames(object)
extraRowDataNames(object)
```

---

# filter *Filter* AnyHermesData *on Subset Passing Default QC Flags*

---

## Description

**[Stable]**

This filters a AnyHermesData object using the default QC flags and required annotations.

## Usage

```
filter(object, ...)

## S4 method for signature 'AnyHermesData'
filter(object, what = c("genes", "samples"), annotation_required = "size")

## S4 method for signature 'data.frame'
filter(object, ...)

## S4 method for signature 'ts'
filter(object, ...)
```

## Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>object to filter. |
| ... | additional arguments. |
| what | (character)<br>specify whether to apply the filter on genes and / or samples. |
| annotation_required | (character)<br>names of required annotation columns for genes. Only used when genes are filtered. |

## Details

- Only genes without low expression (low_expression_flag) and samples without low depth (low_depth_flag) or technical failure (tech_failure_flag) remain in the returned filtered object.
- Also required gene annotation columns can be specified, so that genes which are not complete for these columns are filtered out. By default this is the size column, which is needed for default normalization of the object.

## Value

The filtered AnyHermesData object.

## Note

The internal implementation cannot use the subset() method since that requires non-standard evaluation of arguments.

## Examples

```
a <- hermes_data
dim(a)

# Filter genes and samples on default QC flags.
result <- filter(a)
dim(result)

# Filter only genes without low expression.
result <- filter(a, what = "genes")

# Filter only samples with low depth and technical failure.
result <- filter(a, what = "samples")

# Filter only genes, and require certain annotations to be present.
result <- filter(a, what = "genes", annotation_required = c("size"))
```

## genes *Gene IDs Accessor*

### Description

**[Stable]**

Access the gene IDs, i.e. row names, of a AnyHermesData object with a nicely named accessor method.

### Usage

```
genes(object)

## S4 method for signature 'AnyHermesData'
genes(object)
```

### Arguments

| | |
|---|---|
| object | (AnyHermesData) input. |

### Value

The character vector with the gene IDs.

### See Also

samples() to access the sample IDs.

### Examples

```
a <- hermes_data
genes(a)
```

## GeneSpec *R6 Class Representing a Gene (Signature) Specification*

### Description

**[Experimental]**

A GeneSpec consists of the gene IDs (possibly named with labels), the summary function and the name of the summary function.

## Methods

### Public methods:

- GeneSpec$new()
- GeneSpec$get_genes()
- GeneSpec$get_gene_labels()
- GeneSpec$returns_vector()
- GeneSpec$get_label()
- GeneSpec$extract()
- GeneSpec$extract_data_frame()
- GeneSpec$clone()

**Method** `new()`**:** Creates a new GeneSpec object.

*Usage:*

```
GeneSpec$new(genes = NULL, fun = NULL, fun_name = deparse(substitute(fun)))
```

*Arguments:*

`genes` (named `character` or `NULL`)
: the gene IDs, where the names are used as labels if available.

`fun` (`function` or `NULL`)
: summary function. If `NULL` is used then multiple genes are not summarized but returned as a matrix from the `extract` method.

`fun_name` (`string`)
: name of the summary function.

*Returns:* A new GeneSpec object.

**Method** `get_genes()`**:** Returns the genes.

*Usage:*

```
GeneSpec$get_genes()
```

**Method** `get_gene_labels()`**:** Returns the gene labels (substituted by gene IDs if not available).

*Usage:*

```
GeneSpec$get_gene_labels(genes = self$get_genes())
```

*Arguments:*

`genes` (`character`)
: for which subset of genes the labels should be returned.

**Method** `returns_vector()`**:** Predicate whether the extract returns a vector or not.

*Usage:*

```
GeneSpec$returns_vector()
```

**Method** `get_label()`**:** Returns a string which can be used e.g. for plot labels.

*Usage:*

```
GeneSpec$get_label(genes = self$get_genes())
```

*Arguments:*

genes (character)
: for which subset of genes the labels should be returned.

**Method** `extract()`**:** Extract the gene values from an assay as specified.

*Usage:*

```
GeneSpec$extract(assay)
```

*Arguments:*

assay (matrix)
: original matrix with rownames containing the specified genes.

*Returns:* Either a vector with one value per column, or a matrix with multiple genes in the rows.

**Method** `extract_data_frame()`**:** Extract the gene values as a `data.frame`.

*Usage:*

```
GeneSpec$extract_data_frame(assay)
```

*Arguments:*

assay (matrix)
: original matrix with rownames containing the specified genes.

*Returns:* A `data.frame` with the genes in the columns and the samples in the rows.

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
GeneSpec$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

## Examples

```
# Minimal specification if only one gene is used.
x_spec <- gene_spec("GeneID:1820")

# Using multiple genes with a signature.
x_spec <- gene_spec(c("GeneID:1820", "GeneID:52"), fun = colMeans)
x_spec <- gene_spec(c("GeneID:1820", "GeneID:52"), fun = colPrinComp1)
x_spec$returns_vector()
x_spec$get_genes()
x_spec$get_gene_labels()
x_spec$get_label()

# Using multiple genes with partial labels, without a signature.
x_spec <- gene_spec(c(A = "GeneID:1820", "GeneID:52"))
x_spec$returns_vector()
x_spec$get_gene_labels()

# Use the gene specification to extract genes from a matrix.
mat <- matrix(
  data = rpois(15, 10),
```

```
  nrow = 3, ncol = 5,
  dimnames = list(c("GeneID:1820", "GeneID:52", "GeneID:523"), NULL)
)
x_spec$extract(mat)

# We can also extract these as a `data.frame`.
x_spec$extract_data_frame(mat)
```

---

gene_spec GeneSpec *Constructor*

---

## Description

**[Experimental]**

Creates a new GeneSpec object.

## Usage

```
gene_spec(genes = NULL, fun = NULL, fun_name = deparse(substitute(fun)))
```

## Arguments

| | |
|---|---|
| genes | (named character or NULL)<br>the gene IDs, where the names are used as labels if available. |
| fun | (function or NULL)<br>summary function. If NULL is used then multiple genes are not summarized but returned as a matrix from the extract method. |
| fun_name | (string)<br>name of the summary function. |

## Value

A new GeneSpec object.

## Examples

```
gene_spec("GeneID:11185")
gene_spec(c("GeneID:11185", "GeneID:10677", "GeneID:101928428"), fun = colMeans)
```

HermesData-class HermesData *and* RangedHermesData

## Description

**[Experimental]**

The HermesData class is an extension of SummarizedExperiment::SummarizedExperiment with additional validation criteria.

## Usage

```
HermesData(object)

HermesDataFromMatrix(counts, ...)
```

## Arguments

| | |
|---|---|
| object | (SummarizedExperiment)<br>input to create the HermesData object from. If this is a RangedSummarizedExperiment, then the result will be RangedHermesData. |
| counts | (matrix)<br>counts to create the HermesData object from. |
| ... | additional arguments, e.g. rowData, colData, etc. passed to SummarizedExperiment::SummarizedExpe internally. Note that if rowRanges is passed instead of rowData, then the result will be a RangedHermesData object. |

## Details

The additional criteria are:

- The first assay must be counts containing non-missing, integer, non-negative values. Note that rename() can be used to edit the assay name to counts if needed.
- The following columns must be in rowData:
  - symbol (also often called HGNC or similar, example: "INMT")
  - desc (the gene name, example: "indolethylamine N-methyltransferase")
  - chromosome (the chromosome as string, example: "7")
  - size (the size of the gene in base pairs, e.g 5468)
  - low_expression_flag (can be populated with add_quality_flags())
- The following columns must be in colData:
  - low_depth_flag (can be populated with add_quality_flags())
  - tech_failure_flag (can be populated with add_quality_flags())
- The object must have unique row and column names. The row names are the gene names and the column names are the sample names.

Analogously, RangedHermesData is an extension of SummarizedExperiment::RangedSummarizedExperiment and has the same additional validation requirements. Methods can be defined for both classes at the same time with the AnyHermesData signature.

A Biobase::ExpressionSet object can be imported by using the SummarizedExperiment::makeSummarizedExperimentF function to first convert it to a SummarizedExperiment::SummarizedExperiment object before converting it again into a HermesData object.

## Value

An object of class AnyHermesData (HermesData or RangedHermesData).

## Slots

`prefix` common prefix of the gene IDs (row names).

## Note

- Note that we use S4Vectors::setValidity2() to define the validity method, which allows us to turn off the validity checks in internal functions where intermediate objects may not be valid within the scope of the function.
- It can be helpful to convert character and logical variables to factors in `colData()` (before or after the `HermesData` creation). We provide the utility function df_cols_to_factor() to simplify this task, but leave it to the user to allow for full control of the details.

## See Also

rename() for renaming columns of the input data.

## Examples

```
# Convert an `ExpressionSet` to a `RangedSummarizedExperiment`.
ranged_summarized_experiment <- makeSummarizedExperimentFromExpressionSet(expression_set)

# Then convert to `RangedHermesData`.
HermesData(ranged_summarized_experiment)

# Create objects starting from a `SummarizedExperiment`.
hermes_data <- HermesData(summarized_experiment)
hermes_data

# Create objects from a matrix. Note that additional arguments are not required but possible.
counts_matrix <- assay(summarized_experiment)
counts_hermes_data <- HermesDataFromMatrix(counts_matrix)
```

## hermes_data *Example* HermesData *Data*

### Description

**[Stable]**

This example HermesData is created from the underlying SummarizedExperiment::SummarizedExperiment object by renaming descriptors to align with standard specification. It already contains the required columns in rowData and colData.

### Usage

```
hermes_data
```

### Format

A HermesData object with 20 samples covering 5085 features (Entrez gene IDs).

### Source

This is an artificial dataset designed to resemble real data.

### See Also

summarized_experiment for the underlying SummarizedExperiment::SummarizedExperiment object.

## h_all_duplicated *Finding All Duplicates in Vector*

### Description

The difference here to duplicated() is that also the first occurrence of a duplicate is flagged as TRUE.

### Usage

```
h_all_duplicated(x)
```

### Arguments

| | |
|---|---|
| x | a vector or a data frame or an array or NULL. |

### Value

Logical vector flagging all occurrences of duplicate values as TRUE.

## Examples

```
h_all_duplicated(c("a", "a", "b"))
duplicated(c("a", "a", "b"))
```

---

# h_df_factors_with_explicit_na

*Conversion to Factors with Explicit Missing Level in a* `data.frame`

---

## Description

**[Experimental]**

This helper function converts all character and logical variables to factor variables in a `data.frame`. It also sets an explicit missing data level for all factor variables that have at least one `NA`. Empty strings are handled as `NA`.

## Usage

```
h_df_factors_with_explicit_na(data, na_level = "<Missing>")
```

## Arguments

| | |
|---|---|
| `data` | (`data.frame`)<br>input data with at least one column. |
| `na_level` | (`string`)<br>explicit missing level to be used. |

## Value

The modified data.

## Examples

```
dat <- data.frame(
  a = c(NA, 2),
  b = c("A", NA),
  c = c("C", "D"),
  d = factor(c(NA, "X")),
  e = factor(c("Y", "Z"))
)
h_df_factors_with_explicit_na(dat)
```

h_diff_expr_deseq2 DESeq2 *Differential Expression Analysis*

## Description

**[Experimental]**

This helper functions performs the differential expression analysis with DESeq2::DESeq() for a given AnyHermesData input and design matrix.

## Usage

```
h_diff_expr_deseq2(object, design, ...)
```

## Arguments

| | |
|---|---|
| object | (HermesData)<br>input. |
| design | (matrix)<br>design matrix. |
| ... | additional arguments internally passed to DESeq2::DESeq() (fitType, sfType, minReplicatesForReplace, useT, minmu). |

## Value

A data frame with columns log2_fc (estimated log2 fold change), stat (Wald statistic), p_val (raw p-value), adj_p_pval (Benjamini-Hochberg adjusted p-value).

## References

Love MI, Huber W, Anders S (2014). "Moderated estimation of fold change and dispersion for RNA-seq data with DESeq2." *Genome Biology*, **15**(12), 550. doi:10.1186/s1305901405508.

## Examples

```
object <- hermes_data

# Create the design matrix corresponding to the factor of interest.
design <- model.matrix(~SEX, colData(object))

# Then perform the `DESeq2` differential expression analysis.
result <- h_diff_expr_deseq2(object, design)
head(result)

# Change of the `fitType` can be required in some cases.
result2 <- h_diff_expr_deseq2(object, design, fitType = "local")
head(result2)
```

h_diff_expr_voom limma/*voom Differential Expression Analysis*

## Description

**[Experimental]**

This helper functions performs the differential expression analysis with the voom method from the limma package (via `limma::voom()`, `limma::lmFit()` and `limma::eBayes()`) for given counts in a AnyHermesData object and a corresponding design matrix.

## Usage

```
h_diff_expr_voom(object, design, ...)
```

## Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input. |
| design | (matrix)<br>design matrix. |
| ... | additional arguments internally passed to `limma::eBayes()` (robust, trend, proportion, winsor.tail.p, stdev.coef.lim). |

## Value

A data frame with columns log2_fc (estimated log2 fold change), stat (moderated t-statistic), p_val (raw p-value), adj_p_pval (Benjamini-Hochberg adjusted p-value).

## References

Ritchie ME, Phipson B, Wu D, Hu Y, Law CW, Shi W, Smyth GK (2015). "limma powers differential expression analyses for RNA-sequencing and microarray studies." *Nucleic Acids Research*, **43**(7), e47. doi:10.1093/nar/gkv007.

Law CW, Chen Y, Shi W, Smyth GK (2014). "voom: precision weights unlock linear model analysis tools for RNA-seq read counts." *Genome Biology*, **15**(2), R29. doi:10.1186/gb2014152r29.

## Examples

```
object <- hermes_data

# Create the design matrix corresponding to the factor of interest.
design <- model.matrix(~SEX, colData(object))

# Then perform the differential expression analysis.
result <- h_diff_expr_voom(object, design)
head(result)
```

```
# Sometimes we might want to specify method details.
result2 <- h_diff_expr_voom(object, design, trend = TRUE, robust = TRUE)
head(result2)
```

---

h_ensembl_to_entrez_ids

*Translation of* Ensembl *to* Entrez *Gene IDs*

---

## Description

**[Experimental]**

This helper function queries BioMart to translate Ensembl to Entrez Gene IDs.

## Usage

```
h_ensembl_to_entrez_ids(gene_ids, mart)
```

## Arguments

| | |
|---|---|
| gene_ids | (character)<br>Ensembl gene IDs. |
| mart | (Mart)<br>given biomaRt::Mart object. |

## Value

Character vector of Entrez gene IDs.

## Examples

```
if (interactive()) {
  mart <- biomaRt::useMart("ensembl", dataset = "hsapiens_gene_ensembl")
  h_ensembl_to_entrez_ids(c("ENSG00000135407", "ENSG00000241644"), mart)
}
```

---

h_get_annotation_biomart

*Get Annotations from* BioMart

---

## Description

**[Experimental]**

Helper function to query annotations from biomaRt, for cleaned up gene IDs of a specific ID variable and given biomaRt::Mart.

### Usage

```
h_get_annotation_biomart(gene_ids, id_var, mart)
```

### Arguments

| | |
|---|---|
| gene_ids | (character)<br>gene IDs, e.g. 10329, i.e. already without the Entrez GeneID prefix, or ENSG00000241644 for Ensembl gene ID. |
| id_var | (string)<br>corresponding gene ID variable name in BioMart, i.e. entrezgene_id or ensembl_gene_id. |
| mart | (Mart)<br>given biomaRt::Mart object. |

### Value

A data frame with columns:

- id_var (depending on what was used)
- hgnc_symbol
- entrezgene_description
- chromosome_name
- size
- refseq_mrna
- refseq_peptide

### Examples

```
if (interactive()) {
  mart <- biomaRt::useMart("ensembl", dataset = "hsapiens_gene_ensembl")
  h_get_annotation_biomart(c("11185", "10677"), id_var = "entrezgene_id", mart = mart)
}
```

---

## h_get_granges_by_id *Conversion of* BioMart *Coordinates into* GRanges

---

### Description

**[Experimental]**

This function extracts the chromosome number, the start position and the end position of transcripts in given data.frame with coordinates as returned by biomaRt::getBM() and converts them to a GRanges object.

### Usage

```
h_get_granges_by_id(coords, id)
```

### Arguments

| | |
|---|---|
| coords | (data.frame)<br>as returned by biomaRt::getBM(), containing the columns ensembl_gene_id, chromosome_name, exon_chrom_start, exon_chrom_end. |
| id | (string)<br>single Ensembl gene ID to convert the coordinates for. |

### Value

GRange objects for the respective single gene ID.

### Examples

```
if (interactive()) {
  mart <- biomaRt::useMart("ensembl", dataset = "hsapiens_gene_ensembl")
  attrs <- c(
    "ensembl_gene_id",
    "ensembl_exon_id",
    "chromosome_name",
    "exon_chrom_start",
    "exon_chrom_end"
  )
  coords <- biomaRt::getBM(
    filters = "entrezgene_id",
    attributes = attrs,
    values = c("11185", "10677"),
    mart = mart
  )
  h_get_granges_by_id(coords, "ENSG00000135407")
}
```

---

## h_get_size_biomart *Total Length of All Exons for Genes*

---

### Description

**[Experimental]**

This helper function queries BioMart for lengths of genes by adding up all exon lengths after reducing overlaps.

### Usage

```
h_get_size_biomart(gene_ids, id_var, mart)
```

### Arguments

| | |
|---|---|
| gene_ids | (character)<br>gene IDs, e.g. 10329, i.e. already without the Entrez GeneID prefix, or ENSG00000241644 for Ensembl gene ID. |
| id_var | (string)<br>corresponding gene ID variable name in BioMart, i.e. entrezgene_id or ensembl_gene_id. |
| mart | (Mart)<br>given biomaRt::Mart object. |

### Value

Named integer vector indicating the gene lengths.

### Examples

```
if (interactive()) {
  mart <- biomaRt::useMart("ensembl", dataset = "hsapiens_gene_ensembl")
  h_get_size_biomart("11185", "entrezgene_id", mart)
  h_get_size_biomart("ENSG00000215417", "ensembl_gene_id", mart)
  h_get_size_biomart(c("11185", "10677"), "entrezgene_id", mart)
  h_get_size_biomart(c("ENSG00000135407", "ENSG00000215417"), "ensembl_gene_id", mart)
}
```

---

## h_has_req_annotations *Predicate for Required Annotations*

---

### Description

**[Experimental]**

This helper function determines for each gene in the object whether all required annotation columns are filled.

### Usage

```
h_has_req_annotations(object, annotation_required)
```

### Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input object. |
| annotation_required | (character)<br>names of required annotation columns for genes. |

### Value

Named logical vector with one value for each gene in object, which is TRUE if all required annotation columns are filled, and otherwise FALSE.

## See Also

filter() where this is used internally.

## Examples

```
object <- hermes_data
result <- h_has_req_annotations(object, "size")
all(result)
rowData(object)$size[1] <- NA # nolint
which(!h_has_req_annotations(object, "size"))
```

---

h_map_pos *Helper Function For Matching Map Values to Names*

---

## Description

This is used by the rename method. It wraps the assertions and the matching used several times.

## Usage

```
h_map_pos(names, map)
```

## Arguments

| | |
|---|---|
| names | (character)<br>original names. |
| map | (named character)<br>the mapping vector from old (value) to new (name) names. All values must be included in names. |

## Value

Integer vector of the positions of the map values in the names.

## Examples

```
h_map_pos(c("a", "b"), c(d = "b"))
```

## h_parens *Parenthesize a Character Vector*

### Description

**[Experimental]**

This helper function adds parentheses around each element of a character vector.

### Usage

```
h_parens(x)
```

### Arguments

| | |
|---|---|
| x | (character)<br>inputs which should be parenthesized. |

### Value

Character vector with parentheses, except when x is a blank string in which case it is returned unaltered.

### Examples

```
h_parens("bla")
h_parens("")
h_parens(c("bla", "bli"))
```

## h_pca_df_r2_matrix *Calculation of R2 Matrix between Sample Variables and Principal Components*

### Description

**[Stable]**

This function processes sample variables from AnyHermesData and the corresponding principal components matrix, and then generates the matrix of R2 values.

### Usage

```
h_pca_df_r2_matrix(pca, df)
```

## Arguments

| | |
|---|---|
| pca | (matrix)<br>comprises principal components generated by calc_pca(). |
| df | (data.frame)<br>from the SummarizedExperiment::colData() of a AnyHermesData object. |

## Details

- Note that only the df columns which are numeric, character, factor or logical are included in the resulting matrix, because other variable types are not supported.
- In addition, df columns which are constant, all NA, or character or factor columns with too many levels are also dropped before the analysis.

## Value

A matrix with R2 values for all combinations of sample variables and principal components.

## See Also

h_pca_var_rsquared() which is used internally to calculate the R2 for one sample variable.

## Examples

```
object <- hermes_data %>%
  add_quality_flags() %>%
  filter() %>%
  normalize()

# Obtain the principal components.
pca <- calc_pca(object)$x

# Obtain the `colData` as a `data.frame`.
df <- as.data.frame(colData(object))

# Correlate them.
r2_all <- h_pca_df_r2_matrix(pca, df)
str(r2_all)

# We can see that only about half of the columns from `df` were
# used for the correlations.
ncol(r2_all)
ncol(df)
```

h_pca_var_rsquared *Calculation of R2 between Sample Variable and Principal Components*

## Description

**[Stable]**

This helper function calculates R2 values between one sample variable from AnyHermesData and all Principal Components (PCs) separately (one linear model is fit for each PC).

## Usage

```
h_pca_var_rsquared(pca, x)
```

## Arguments

| | |
|---|---|
| pca | (matrix)<br>principal components matrix generated by calc_pca(). |
| x | (vector)<br>values of one sample variable from a AnyHermesData object. |

## Details

Note that in case there are estimation problems for any of the PCs, then NA will be returned for those.

## Value

A vector with R2 values for each principal component.

## Examples

```
object <- hermes_data %>%
  add_quality_flags() %>%
  filter() %>%
  normalize()

# Obtain the principal components.
pca <- calc_pca(object)$x

# Obtain the sample variable.
x <- colData(object)$AGE18

# Correlate them.
r2 <- h_pca_var_rsquared(pca, x)
```

## h_short_list *Make a Short List of a Character Vector*

### Description

**[Experimental]**

This helper function makes a short list string, e.g. "a, b, ..., z" out of a character vector, e.g. `letters`.

### Usage

```
h_short_list(x, sep = ", ", thresh = 3L)
```

### Arguments

| | |
|---|---|
| `x` | (`character`)<br>input which should be listed. |
| `sep` | (`string`)<br>separator to use. |
| `thresh` | (`count`)<br>threshold to use, if the length of `x` is larger then the list will be shortened using the `...` ellipsis. |

### Value

String with the short list.

### Examples

```
h_short_list(letters)
h_short_list(letters[1:3])
h_short_list(LETTERS[1:5], sep = ";", thresh = 5L)
```

## h_strip_prefix *Stripping Prefix from Gene IDs*

### Description

**[Experimental]**

This helper function removes the prefix and possible delimiter from a vector of gene IDs, such that only the digits are returned.

### Usage

```
h_strip_prefix(gene_ids, prefix)
```

### Arguments

| | |
|---|---|
| gene_ids | (character)<br>original gene IDs including prefix and optional delimiter before the digits. |
| prefix | (string)<br>common prefix to be stripped away from gene_ids. |

### Value

Character vector that contains only the digits for each gene ID.

### Note

This is currently used to strip away the GeneID prefix from Entrez gene IDs so that they can be queried from BioMart

### Examples

```
h_strip_prefix(c("GeneID:11185", "GeneID:10677"), prefix = "GeneID")
```

---

## h_unique_labels *Creation of Unique Labels*

---

### Description

This helper function generates a set of unique labels given unique IDs and not necessarily unique names.

### Usage

```
h_unique_labels(ids, nms = NULL)
```

### Arguments

| | |
|---|---|
| ids | (character or NULL)<br>unique IDs. |
| nms | (character or NULL)<br>not necessarily unique names if provided. |

### Value

Character vector where empty names are replaced by the IDs and non-unique names are made unique by appending the IDs in parentheses.

### Examples

```
h_unique_labels(c("1", "2", "3"), c("A", "B", "A"))
h_unique_labels(NULL)
h_unique_labels(c("1", "2", "3"))
```

inner_join_cdisc *Inner Joining a Genes with a CDISC Data Set*

## Description

**[Experimental]**

This is a useful function when trying to join genetic with CDISC data sets.

## Usage

```
inner_join_cdisc(
  gene_data,
  cdisc_data,
  patient_key = "USUBJID",
  additional_keys = character()
)
```

## Arguments

| | |
|---|---|
| gene_data | (data.frame or DataFrame)<br>genetic data. |
| cdisc_data | (data.frame)<br>CDISC data (typically patient level data). |
| patient_key | (string)<br>patient identifier. |
| additional_keys | (character)<br>potential additional keys for the two data sets. |

## Value

A data.frame which contains columns from both data sets merged by the keys.

## Note

Columns which are contained in both data sets but are not specified as keys are taken from gene_data and not from cdisc_data.

## Examples

```
gene_data <- col_data_with_genes(hermes_data, "counts", gene_spec("GeneID:1820"))
cdisc_data <- data.frame(
  USUBJID = head(gene_data$USUBJID, 10),
  extra = 1:10
)
result <- inner_join_cdisc(gene_data, cdisc_data)
result
```

---

`isEmpty,SummarizedExperiment-method`
*Checking for Empty* `SummarizedExperiment`

---

## Description

**[Experimental]**

This method checks whether a `SummarizedExperiment::SummarizedExperiment` object is empty.

## Usage

```
## S4 method for signature 'SummarizedExperiment'
isEmpty(x)
```

## Arguments

| | |
|---|---|
| `x` | (`SummarizedExperiment`) object to check. |

## Value

Flag whether the `object` is empty.

## Examples

```
isEmpty(summarized_experiment)
isEmpty(summarized_experiment[NULL, ])
isEmpty(hermes_data)
```

---

`lapply,MultiAssayExperiment-method`
`lapply` *method for* `MultiAssayExperiment`

---

## Description

**[Experimental]**

Apply a function on all experiments in an MAE.

## Usage

```
## S4 method for signature 'MultiAssayExperiment'
lapply(X, FUN, safe = TRUE, ...)
```

### Arguments

| | |
|---|---|
| X | (MultiAssayExperiment)<br>input. |
| FUN | (function) to be applied to each experiment in X. |
| safe | (flag)<br>whether this method should skip experiments where the function fails. |
| ... | additional arguments passed to FUN. |

### Value

MultiAssayExperiment object with specified function applied.

### Examples

```
object <- multi_assay_experiment
result <- lapply(object, normalize, safe = TRUE)
# Similarly, all experiments in an MAE can be converted to HermesData class:
result <- lapply(object, HermesData, safe = TRUE)
```

---

## metadata *Metadata Accessor and Setter*

---

### Description

**[Stable]**

These methods access or set the metadata in a AnyHermesData object.

### Arguments

| | |
|---|---|
| x | (AnyHermesData)<br>object to access the metadata from. |
| value | (list)<br>the list to replace the current metadata with. |

### Value

The metadata which is a list.

### Note

Note that this just inherits S4Vectors::metadata,Annotated-method().

### Examples

```
a <- hermes_data
metadata(a)
metadata(a) <- list(new = "my metadata")
metadata(a)
```

---

multi_assay_experiment

*Example* MultiAssayExperiment *Data*

---

## Description

**[Experimental]**

This example MultiAssayExperiment::MultiAssayExperiment can be used as test data.

## Usage

```
multi_assay_experiment
```

## Format

A MultiAssayExperiment::MultiAssayExperiment object with 3 separate HermesData objects.

- The first object contains 5 samples and covers 1000 features (Entrez gene IDs).
- The second object contains 9 samples with 2500 features.
- The third object contains 6 samples with 1300 features.

## Source

This is an artificial dataset designed to resemble real data.

---

normalize,AnyHermesData-method

*Normalization of* AnyHermesData *Objects*

---

## Description

**[Stable]**

The normalize() method is normalizing the input AnyHermesData according to one or more specified normalization methods. The results are saved as additional assays in the object.

Possible normalization methods (which are implemented with separate helper functions):

- cpm: Counts per Million (CPM). Separately by sample, the original counts of the genes are divided by the library size of this sample, and multiplied by one million. This is the appropriate normalization for between-sample comparisons.
- rpkm: Reads per Kilobase of transcript per Million reads mapped (RPKM). Each gene count is divided by the gene size (in kilobases) and then again divided by the library sizes of each sample (in millions). This allows for within-sample comparisons, as it takes into account the gene sizes - longer genes will always have more counts than shorter genes.

- tpm: Transcripts per Million (TPM). This addresses the problem of RPKM being inconsistent across samples (which can be seen that the sum of all RPKM values will vary from sample to sample). Therefore here we divide the RPKM by the sum of all RPKM values for each sample, and multiply by one million.
- voom: VOOM normalization. This is essentially just a slight variation of `CPM` where a `prior_count` of 0.5 is combined with `lib_sizes` increased by 1 for each sample. Note that this is not required for the corresponding differential expression analysis, but just provided as a complementary experimental normalization approach here.
- `vst`: Variance stabilizing transformation. This is to transform the normalized count data for all genes into approximately homoskedastic values (having constant variance).
- `rlog`: The transformation to the log2 scale values with approximately homoskedastic values.

## Usage

```
## S4 method for signature 'AnyHermesData'
normalize(
  object,
  methods = c("cpm", "rpkm", "tpm", "voom", "vst"),
  control = control_normalize(),
  ...
)

h_cpm(object, control = control_normalize())

h_rpkm(object, control = control_normalize())

h_tpm(object, control = control_normalize())

h_voom(object, control = control_normalize())

h_vst(object, control = control_normalize())

h_rlog(object, control = control_normalize())
```

## Arguments

| | |
|---|---|
| `object` | (`AnyHermesData`)<br>object to normalize. |
| `methods` | (`character`)<br>which normalization methods to use, see details. |
| `control` | (named `list`)<br>settings produced by `control_normalize()`. |
| `...` | not used. |

## Value

The `AnyHermesData` object with additional assays containing the normalized counts. The `control` is saved in the `metadata` of the object for future reference.

## Functions

- `h_cpm()`: calculates the Counts per Million (`CPM`) normalized counts.
- `h_rpkm()`: calculates the Reads per Kilobase per Million (RPKM) normalized counts.
- `h_tpm()`: calculates the Transcripts per Million (TPM) normalized counts.
- `h_voom()`: calculates the VOOM normalized counts. **[Experimental]**
- `h_vst()`: variance stabilizing transformation (`vst`) from `DESeq2` package.
- `h_rlog()`: regularized log transformation (`rlog`) from `DESeq2` package.

## See Also

`control_normalize()` to define the normalization method settings.

## Examples

```
a <- hermes_data

# By default, log values are used with a prior count of 1 added to original counts.
result <- normalize(a)
assayNames(result)
tpm <- assay(result, "tpm")
tpm[1:3, 1:3]

# We can also work on original scale.
result_orig <- normalize(a, control = control_normalize(log = FALSE))
tpm_orig <- assay(result_orig, "tpm")
tpm_orig[1:3, 1:3]

# Separate calculation of the CPM normalized counts.
counts_cpm <- h_cpm(a)
str(counts_cpm)

# Separate calculation of the RPKM normalized counts.
counts_rpkm <- h_rpkm(a)
str(counts_rpkm)

# Separate calculation of the TPM normalized counts.
counts_tpm <- h_tpm(a)
str(counts_tpm)

# Separate calculation of the VOOM normalized counts.
counts_voom <- h_voom(a)
str(counts_voom)

# Separate calculation of the vst transformation.
counts_vst <- h_vst(a)
str(counts_vst)

# Separate calculation of the rlog transformation.
counts_rlog <- h_rlog(a)
str(counts_rlog)
```

prefix *Prefix Accessor*

## Description

**[Experimental]**

Generic function to access the prefix from an object.

## Usage

```
prefix(object, ...)
```

## Arguments

| | |
|---|---|
| object | (AnyHermesData) input. |
| ... | additional arguments. |

## Value

The prefix slot contents.

## Examples

```
a <- hermes_data
prefix(a)
```

query *Query Gene Annotations from a Connection*

## Description

**[Experimental]**

The generic function query() is the interface for querying gene annotations from a data base connection.

## Usage

```
query(genes, connection)

## S4 method for signature 'character,ConnectionBiomart'
query(genes, connection)
```

### Arguments

| | |
|---|---|
| genes | (character)<br>gene IDs. |
| connection | (connection class)<br>data base connection object. |

### Details

- A method is provided for the ConnectionBiomart class. However, the framework is extensible: It is simple to add new connections and corresponding query methods for other data bases, e.g. company internal data bases. Please make sure to follow the required format of the returned value.
- The BioMart queries might not return information for all the genes. This can be due to different versions being used in the gene IDs and the queried Ensembl data base.

### Value

A S4Vectors::DataFrame with the gene annotations. It is required that:

- The rownames are identical to the input genes.
- The colnames are equal to the annotation columns .row_data_annotation_cols.
- Therefore, missing information needs to be properly included in the DataFrame with NA entries.

### Examples

```
if (interactive()) {
  object <- hermes_data
  connection <- connect_biomart(prefix(object))
  result <- query(genes(object), connection)
  head(result)
  head(annotation(object))
}
```

---

## rbind *Row Binding of* AnyHermesData *Objects*

---

### Description

**[Stable]**

This method combines AnyHermesData objects with the same samples but different features of interest (rows in assays).

### Arguments

| | |
|---|---|
| ... | (AnyHermesData)<br>objects to row bind. |

## Value

The combined AnyHermesData object.

## Note

- Note that this just inherits SummarizedExperiment::rbind,SummarizedExperiment-method(). When binding a AnyHermesData object with a SummarizedExperiment::SummarizedExperiment object, then the result will be a SummarizedExperiment::SummarizedExperiment object (the more general class).
- Note that we need to have unique gene IDs (row names) and the same prefix across the combined object.

## See Also

cbind to column bind objects.

## Examples

```
a <- hermes_data[1:2542, ]
b <- hermes_data[2543:5085, ]
result <- rbind(a, b)
class(result)
```

---

# rename,SummarizedExperiment-method

*Renaming Contents of* SummarizedExperiment *Objects*

---

## Description

**[Experimental]**

This method renames columns of the rowData and colData, as well as assays, of SummarizedExperiment::SummarizedExp objects. This increases the flexibility since renaming can be done before conversion to a HermesData object.

## Usage

```
## S4 method for signature 'SummarizedExperiment'
rename(
  x,
  row_data = character(),
  col_data = character(),
  assays = character(),
  ...
)

## S4 method for signature 'data.frame'
rename(x, ...)
```

## Arguments

| | |
|---|---|
| x | (SummarizedExperiment)<br>object to rename contents in. |
| row_data | (named character)<br>mapping from existing (right-hand side values) to new (left-hand side names) column names of rowData. |
| col_data | (named character)<br>mapping from existing (right-hand side values) to new (left-hand side names) column names of colData. |
| assays | (named character)<br>mapping from existing (right-hand side values) to new (left-hand side names) assay names. |
| ... | additional arguments (not used here). |

## Value

The SummarizedExperiment::SummarizedExperiment object with renamed contents.

## Examples

```
x <- summarized_experiment
# Use deliberately a non-standard assay name in this example.
assayNames(x) <- "count"

# Rename `HGNC` to `symbol` in the `rowData`.
x <- rename(x, row_data = c(symbol = "HGNC"))
head(names(rowData(x)))

# Rename `LowDepthFlag` to `low_depth_flag` in `colData`.
x <- rename(x, col_data = c(low_depth_flag = "LowDepthFlag"))
tail(names(colData(x)))

# Rename assay `count` to `counts`.
x <- rename(x, assays = c(counts = "count"))
assayNames(x)
```

---

# samples,AnyHermesData-method

*Sample IDs Accessor*

---

## Description

**[Stable]**

Access the sample IDs, i.e. col names, of a AnyHermesData object with a nicely named accessor method.

### Usage

```
## S4 method for signature 'AnyHermesData'
samples(object)
```

### Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input. |

### Value

The character vector with the sample IDs.

### See Also

genes() to access the gene IDs.

### Examples

```
a <- hermes_data
samples(a)
```

---

## set_tech_failure *Set Technical Failure Flags*

---

### Description

**[Experimental]**

Setter function which allows the user to define a sample manually as a technical failure.

### Usage

```
set_tech_failure(object, sample_ids)
```

### Arguments

| | |
|---|---|
| object | (AnyHermesData)<br>input. |
| sample_ids | (character)<br>sample IDs to be flagged as technical failures. |

### Value

AnyHermesData object with modified technical failure flags.

## See Also

add_quality_flags() which automatically sets all (gene and sample) quality flags, including these technical failure flags.

## Examples

```
# Manually flag technical failures in a `AnyHermesData` object.
object <- hermes_data
get_tech_failure(object)["06520101B0017R"]
result <- set_tech_failure(object, c("06520101B0017R", "06520047C0017R"))
get_tech_failure(result)["06520101B0017R"]
```

---

# show,HermesData-method

*Show Method for* AnyHermesData *Objects*

---

## Description

**[Experimental]**

A show method that displays high-level information of AnyHermesData objects.

## Usage

```
## S4 method for signature 'HermesData'
show(object)

## S4 method for signature 'RangedHermesData'
show(object)
```

## Arguments

| | |
|---|---|
| object | (AnyHermesData) input. |

## Value

None (invisible NULL), only used for the side effect of printing to the console.

## Note

The same method is used for both HermesData and RangedHermesData objects. We need to define this separately to have this method used instead of the one inherited from SummarizedExperiment::SummarizedExperiment

## Examples

```
object <- hermes_data
object
```

subset *Subsetting* AnyHermesData *Objects*

## Description

**[Stable]**

This method subsets AnyHermesData objects, based on expressions involving the rowData columns and the colData columns.

## Arguments

| | |
|---|---|
| x | (AnyHermesData)<br>object to subset from. |
| subset | (expression)<br>logical expression based on the rowData columns to select genes. |
| select | (expression)<br>logical expression based on the colData columns to select samples. |

## Value

The subsetted AnyHermesData object.

## Note

Note that this just inherits SummarizedExperiment::subset,SummarizedExperiment-method().

## Examples

```
a <- hermes_data
a

# Subset both genes and samples.
subset(a, subset = low_expression_flag, select = DISCSTUD == "N")

# Subset only genes.
subset(a, subset = chromosome == "2")

# Subset only samples.
subset(a, select = AGE > 18)
```

## summarized_experiment *Example* SummarizedExperiment *Data*

### Description

**[Stable]**

This example SummarizedExperiment::SummarizedExperiment can be used to create a HermesData object. It already contains the required columns in rowData and colData.

### Usage

```
summarized_experiment
```

### Format

A SummarizedExperiment::SummarizedExperiment object with 20 samples covering 5085 features (Entrez gene IDs).

### Source

This is an artificial dataset designed to resemble real data.

### See Also

expression_set which contains similar data as a Biobase::ExpressionSet.

## summary *Summary Method for* AnyHermesData *Objects*

### Description

**[Experimental]**

Provides a concise summary of the content of AnyHermesData objects.

### Usage

```
summary(object, ...)

## S4 method for signature 'AnyHermesData'
summary(object)

## S4 method for signature 'HermesDataSummary'
show(object)
```

### Arguments

| | |
|---|---|
| object | (HermesDataSummary)<br>result from the summary method applied to AnyHermesData object. |
| ... | not used. |

### Value

An object of the corresponding summary class, here HermesDataSummary.

### Methods (by class)

- summary(AnyHermesData): A summary method for AnyHermesData object that creates a HermesDataSummary object.
- show(HermesDataSummary): A show method prints summary description of HermesDataSummary object generated by the summary() method.

### Examples

```
object <- hermes_data
object_summary <- summary(object)

# We can access parts of this S4 object with the `slot` function.
str(object_summary)
slotNames(object_summary)
slot(object_summary, "lib_sizes")

# Just calling the summary method like this will use the `show()` method.
summary(object)
```

---

## top_genes *Derivation of Top Genes*

---

### Description

**[Experimental]**

top_genes() creates a HermesDataTopGenes object, which extends data.frame. It contains two columns:

- expression: containing the statistic values calculated by summary_fun across columns.
- name: the gene names.

The corresponding autoplot() method then visualizes the result as a barplot.

## Usage

```
top_genes(
  object,
  assay_name = "counts",
  summary_fun = rowMeans,
  n_top = if (is.null(min_threshold)) 10L else NULL,
  min_threshold = NULL
)

## S4 method for signature 'HermesDataTopGenes'
autoplot(
  object,
  x_lab = "HGNC gene names",
  y_lab = paste0(object@summary_fun_name, "(", object@assay_name, ")"),
  title = "Top most expressed genes"
)
```

## Arguments

| | |
|---|---|
| `object` | (`AnyHermedData`)<br>input. |
| `assay_name` | (`string`)<br>name of the assay to use for the sorting of genes. |
| `summary_fun` | (`function`)<br>summary statistics function to apply across the samples in the assay resulting in a numeric vector with one value per gene. |
| `n_top` | (`count` or `NULL`)<br>selection criteria based on number of entries. |
| `min_threshold` | (`number` or `NULL` )<br>selection criteria based on a minimum summary statistics threshold. |
| `x_lab` | (`string`)<br>x-axis label. |
| `y_lab` | (`string`)<br>y-axis label. |
| `title` | (`string`)<br>plot title. |

## Details

- The data frame is sorted in descending order of `expression` and only the top entries according to the selection criteria are included.
- Note that exactly one of the arguments `n_top` and `min_threshold` must be provided.

## Value

A HermesDataTopGenes object.

### Functions

- `autoplot(HermesDataTopGenes)`: Creates a bar plot from a HermesDataTopGenes object, where the y axis shows the expression statistics for each of the top genes on the x-axis.

### Examples

```
object <- hermes_data

# Default uses average of raw counts across samples to rank genes.
top_genes(object)

# Instead of showing top 10 genes, can also set a minimum threshold on average counts.
top_genes(object, n_top = NULL, min_threshold = 50000)

# We can also use the maximum of raw counts across samples, by specifying a different
# summary statistics function.
result <- top_genes(object, summary_fun = rowMax)

# Finally we can produce barplots based on the results.
autoplot(result, title = "My top genes")
autoplot(result, y_lab = "Counts", title = "My top genes")
```

---

## validate *Internal Helper Functions for Validation of* `AnyHermesData` *Objects*

---

### Description

These functions are used internally only and therefore not exported. They work on `SummarizedExperiment::SummarizedEx` objects, and `AnyHermesData` objects are defined by successfully passing these validation checks.

### Usage

```
validate_counts(object)

validate_cols(required, actual)

validate_row_data(object)

validate_col_data(object)

validate_names(object)

validate_prefix(object)
```

### Arguments

| | |
|---|---|
| object | (SummarizedExperiment)<br>object to validate. |
| required | (character)<br>required column names. |
| actual | (actual)<br>actual column names. |

### Value

A character vector with the validation failure messages, or NULL in case validation passes.

### Functions

- validate_counts(): validates that the first assay is counts containing non-missing, integer, non-negative values.
- validate_cols(): validates that required column names are contained in actual column names.
- validate_row_data(): validates that the object contains rowData with required columns.
- validate_col_data(): validates that the object contains colData with required columns.
- validate_names(): validates that the object contains row and column names.
- validate_prefix(): validates that the object prefix is a string and only contains alphabetic characters.

---

## wrap_in_mae *Wrap in MAE*

---

### Description

**[Experimental]**

This helper function wraps SummarizedExperiment objects into an a MultiAssayExperiment (MAE) object.

### Usage

```
wrap_in_mae(x, name = deparse(substitute(x)))
```

### Arguments

| | |
|---|---|
| x | (SummarizedExperiment)<br>input to create the MAE object from. |
| name | (string)<br>experiment name to use in the MAE for x. |

## Value

The MAE object with the only experiment being `x` having the given `name`.

## Examples

```
mae <- wrap_in_mae(summarized_experiment)
mae[["summarized_experiment"]]
```

---

`%>%` *Pipe operator*

---

## Description

**[Stable]**

See `magrittr::%>%` for details.

## Usage

```
lhs %>% rhs
```

## Value

The result of the corresponding function call.

## Examples

```
hermes_data %>%
  filter() %>%
  normalize() %>%
  summary()
```

# Index