Package 'matrixset'

January 14, 2024

Title Creating, Manipulating and Annotating Matrix Ensemble

Version 0.3.0

Description Creates an object that stores a matrix ensemble, matrices that share the same common properties, where rows and columns can be annotated. Matrices must have the same dimension and dimnames. Operators to manipulate these objects are provided as well as mechanisms to apply functions to these objects.

Imports cli, crayon, dplyr, Matrix, methods, pillar, purrr, R6, Rcpp, rlang, stringr, tibble, tidyr, tidyselect, vctrs

License MIT + file LICENSE

URL https://github.com/pascalcroteau/matrixset

BugReports https://github.com/pascalcroteau/matrixset/issues

Encoding UTF-8

LazyData true

RoxygenNote 7.2.3

Suggests MASS, ggfortify, knitr, lme4, magrittr, patchwork, rmarkdown, testthat (>= 3.0.0), tidyverse

Config/testthat/edition 3

Depends R (>= 4.0)

VignetteBuilder knitr

LinkingTo Rcpp

NeedsCompilation yes

Author Pascal Croteau [aut, cre, cph]

Maintainer Pascal Croteau <croteaupascl@gmail.com>

Repository CRAN

Date/Publication 2024-01-14 15:40:03 UTC

R topics documented:

`add_matrix` *Add matrices to the* `matrixset` *object*

Description

Matrices to add must be of the same dimension and dimnames as `.ms`.

Either a named list of matrices can be supplied, or matrices can be specified separaely.

Usage

```
add_matrix(.ms, ...)
```

Arguments

`.ms`	A `matrixset` object.
`...`	A single list of matrices (must be a named list), or individual matrices, e.g. `mat1 = m1`, `mat2 = m2`, etc. `NULL` elements are accepted. This allows to create a placeholder that can be filled later on.

Value

A `matrixset` with updated matrices.

Examples

```
m1 <- matrix(1:60, 20, 3)
dimnames(m1) <- dimnames(student_results)
m2 <- matrix(101:160, 20, 3)
dimnames(m2) <- dimnames(student_results)

ms <- add_matrix(student_results, m1=m1, m2=m2)
ms2 <- add_matrix(student_results, list(m1=m1, m2=m2))
```

---

annotate *Create/modify/delete annotations of a* `matrixset` *object*

Description

An annotation is a trait that is stored in the meta (row or column) data frame of the `.ms` object.

Creating an annotation is done as when applying a `mutate()` on a data frame. Thus, annotations can be created from already existing annotations.

The usage is the same as for `dplyr::mutate()`, so see this function for instructions on how to create/modify or delete traits.

The only difference is that the tag is a special annotation that can't be deleted or modify (with one exception in case of modification). The tag is the column name of the meta data frame that holds the row or column names. The tag identity of the' object can be obtained via `row_tag()` or `column_tag()`. To modify a tag, see `rownames<-()` or `colnames<-()`.

Usage

```
annotate_row(.ms, ...)

annotate_column(.ms, ...)
```

Arguments

`.ms`	A `matrixset` object.
`...`	Name-value pairs, ala `dplyr`'s `dplyr::mutate()`.

Value

A `matrixset` with updated meta info.

See Also

`annotate_row_from_apply()`/`annotate_column_from_apply()`, a version that allows access to the `matrixset` matrices.

Examples

```
# You can create annotation from scrath or using already existing annotation
ms <- annotate_row(student_results,
                   dummy = 1,
                   passed = ifelse(previous_year_score >= 0.6, TRUE, FALSE))

# There is a direct access to matrix content with annotate_row_from_apply(),
# but here is an example on how it can be done with annotate_row()
ms <- annotate_row(student_results,
                   mn_fail = apply_matrix_dfl(student_results, mn=~ rowMeans(.m1),
                                              .matrix_wise = FALSE)$mn)
```

`annotate_from_matrix` *Apply functions to a single matrix of a matrixset and store results as annotation*

Description

This is in essence `apply_row_dfw()`/`apply_column_dfw()`, but with the results saved as new annotations. As such, the usage is almost identical to these functions, except that only a single matrix can be used, and must be specified (matrix specification differs also slightly).

Usage

```
annotate_row_from_apply(
  .ms,
  .matrix,
  ...,
  names_prefix = "",
  names_sep = "_",
  names_glue = NULL,
  names_sort = FALSE,
  names_vary = "fastest",
  names_expand = FALSE
)

annotate_column_from_apply(
  .ms,
  .matrix,
  ...,
  names_prefix = "",
  names_sep = "_",
  names_glue = NULL,
  names_sort = FALSE,
  names_vary = "fastest",
  names_expand = FALSE
)
```

Arguments

`.ms`	`matrixset` object
`.matrix`	a tidyselect matrix name: matrix name as a bare name or a character.
`...`	expressions, separated by commas. They can be specified in one of the following way: • a function name, e.g., `mean`. • a function call, where you can use `.m` to represent the current matrix (for `apply_matrix`), `.i` to represent the current row (for `apply_row`) and `.j` for the current column (`apply_column`). Bare names of object traits can be used as well. For instance, `lm(.i ~ program)`. The pronouns are also available for the multivariate version, under certain circumstances, but they have a different meaning. See the "Multivariate" section for more details. • a formula expression. The pronouns `.m`, `.i` and `.j` can be used as well. See examples to see the usefulness of this. The expressions can be named; these names will be used to provide names to the results.
`names_prefix, names_sep, names_glue, names_sort, names_vary, names_expand`	See the same arguments of `tidyr::pivot_wider()`

Details

A conscious choice was made to provide this functionality only for `apply_*_dfw()`, as this is the only version for which the output dimension is guaranteed to respect the `matrixset` paradigm.

On that note, see the section 'Grouped `matrixset`'.

Value

A `matrixset` with updated meta info.

Grouped `matrixset`

In the context of grouping, the `apply_*_dfw()` functions stack the results for each group value.

In the case of `annotate_*_from_matrix()`, a `tidyr::pivot_wider()` is further applied to ensure compatibility of the dimension.

The `pivot_wider()` arguments `names_prefix`, `names_sep`, `names_glue`, `names_sort`, `names_vary` and `names_expand` can help you control the final annotation trait names.

See Also

`annotate_row()`/`annotate_column()`.

Examples

```
# This is the same example as in annotate_row(), but with the "proper" way
# of doing it
ms <- annotate_row_from_apply(student_results, "failure", mn = mean)
```

arrange *Re-order rows or columns of a* matrixset

Description

Orders the rows (arrange_row()) or columns (arrange_column()) by annotation values.

The mechanic is based on sorting the annotation data frame via dplyr's dplyr::arrange().

This means, for instance, that grouping is ignored by default. You must either specify the grouping annotation in the sorting annotation, or use .by_group = TRUE.

The handling of locales and handling of missing values is also governed by dplyr's arrange().

Usage

```
arrange_row(.ms, ..., .by_group = FALSE)

arrange_column(.ms, ..., .by_group = FALSE)
```

Arguments

.ms	A matrixset object
...	Name of traits to base sorting upon. Tidy selection is supported. Use dplyr::desc() to sort an annotation in descending order.
.by_group	logical. Defaults to FALSE and even if TRUE, has no impact on ungrouped margin. Otherwise, grouping annotation is used first for sorting.

Value

A matrixset with re-ordered rows or columns, including updated row or column meta info.

Examples

```
ms1 <- remove_row_annotation(student_results, class, teacher)

# this would not work
# remove_row_annotation(row_group_by(student_results, class), class)
```

as_matrixset *Coerce object into* matrixset

Description

Turns object into a matrixset. See specific methods for more details

Usage

```
as_matrixset(
  x,
  expand = NULL,
  row_info = NULL,
  column_info = NULL,
  row_key = "rowname",
  column_key = "colname",
  row_tag = ".rowname",
  column_tag = ".colname"
)
```

Arguments

x	an object to coerce to matrixset. See methods.
expand	By default (NULL), input matrix expansion is disabled. Setting this parameter to TRUE will enable the expansion feature. See the section 'Matrix Expansion' of matrixset() for more details of what it is, as well as other possible options for expand. Note as well that this argument is not available for all methods.
row_info	a data frame, used to annotate matrix rows. The link between the matrix row names and the data frame is given in column "rowname". A different column can be used if one provides a different row_key.
column_info	a data frame, used to annotate matrix columns. The link between the matrix column names and the data frame is given in column "colname". A different column can be used if one provides a different column_key.
row_key	column name in row_info data frame that will link the row names with the row information. A string is expected.
column_key	column name in col_info data frame that will link the column names with the row information. A string is expected.
row_tag	A string, giving the row annotation data frame column that will link the row names to the data frame. While row_key specifies the column name of the data frame at input, row_tag specifies the column name that will be used throughout in the matrixset object.
column_tag	A string, giving the column annotation data frame column that will link the row names to the data frame. While column_key specifies the column name of the data frame at input, column_tag specifies the column name that will be used throughout in the matrixset object.

Value

Returns a matrixset - see matrixset().

Methods

- matrix
  The matrix method is very similar to calling the matrixset construction function, with some key differences:
  - A matrix name will be provided automatically by as_matrixset. The name is "..1".
  - Because only matrix is provided, the expand argument is not available
- list
  The list method is nearly identical to calling the matrixset construction function. It only differs in that unnamed list element will be padded with a name. The new padded names are the element index, prefixed by "..". Already existing names will be made unique as well. If name modification needs to be performed, a warning will be issued.

Examples

```
# We're showing how 'as_matrixset' can differ. But first, show how they can
# yield the same result. Note that the list is named
lst <- list(a = matrix(1:6, 2, 3), b = matrix(101:106, 2, 3))
identical(matrixset(lst), as_matrixset(lst))

# Now it will differ: the list is unnamed. In fact, 'matrixset' will fail
lst <- list(matrix(1:6, 2, 3), matrix(101:106, 2, 3))
is(try(matrixset(lst), silent = TRUE), "try-error")
as_matrixset(lst)

# You need to name the matrix to use 'matrixset'. A name is provided for you
# with 'as_matrixset'. But you can't control what it is.
as_matrixset(matrix(1:6, 2, 3))
```

---

column_group_by_drop_default

Default value for .drop argument of function column_group_by()

Description

Default value for .drop argument of function column_group_by()

Usage

```
column_group_by_drop_default(.ms)
```

Arguments

.ms	a matrixset object

Value

Returns TRUE for column-ungrouped matrixsets. For column-grouped objects, the default is also TRUE unless .ms has been previously grouped with .drop = FALSE.

Examples

```
student_results |>
    row_group_by(class, .drop = FALSE) |>
    row_group_by_drop_default()
```

---

context *Contexts dependent functions*

Description

These functions are designed to work inside certain matrixset functions, to have access to current group/matrix/row/column. Because of that, they will not work in a general context.

The functions within which the context functions will work are apply_matrix(), apply_row() and apply_column() - as well as their *_dfl/*dfw variant.

Note that "current" refers to the current matrix/group/row/column, as applicable, and possibly combined.

The context functions are:

- current_n_row() and current_n_column(). They each give the number of rows and columns, respectively, of the current matrix.

  They are the context equivalent of nrow() and ncol().
- current_row_name() and current_column_name(). They provide the current row/column name. They are the context equivalent of rownames() and colnames().
- current_row_info() and current_column_info(). They give access to the current row/column annotation data frame. The are the context equivalent of row_info() and column_info().
- row_pos() and column_pos(). They give the current row/column indices. The indices are the the ones before matrix subsetting.
- row_rel_pos() and column_rel_pos(). They give the row/column indices relative to the current matrix. They are equivalent to seq_len(current_n_row())/seq_len(current_n_column()).

Usage

```
current_row_info()

current_column_info()

current_n_row()

current_n_column()
```

```
current_row_name()

row_pos()

row_rel_pos()

current_column_name()

column_pos()

column_rel_pos()
```

Value

See each individual functions for returned value when used in proper context. If used out of context, an error condition is issued.

Examples

```
# this will fail (as it should), because it is used out of context
is(try(current_n_row(), silent = TRUE), "try-error")

# this is one way to know the number of students per class in 'student_results'
student_results |>
    apply_matrix_dfl(n = ~ current_n_row(), .matrix = 1)
```

---

filter_column *Subset columns using annotation values*

Description

The `filter_column()` function subsets the columns of all matrices of a `matrixset`, retaining all columns that satisfy given condition(s). The function `filter_column` works like `dplyr`'s `dplyr::filter()`.

Usage

```
filter_column(.ms, ..., .preserve = FALSE)
```

Arguments

`.ms`	`matrixset` object to subset based on the filtering conditions
`...`	Condition, or expression, that returns a logical value, used to determine if columns are kept or discarded. The expression may refer to column annotations - columns of the `column_info` component of `.ms` More than one condition can be supplied and if multiple expressions are included, they are combined with the `&` operator. Only columns for which all conditions evaluate to TRUE are kept.

.preserve — logical, relevant only if .ms is column grouped. When .preserve is FALSE (the default), the column grouping is updated based on the new matrixset resulting from the filtering. Otherwise, the column grouping is kept as is.

Details

The conditions are given as expressions in ..., which are applied to columns of the annotation data frame (column_info) to determine which columns should be retained.

It can be applied to both grouped and ungrouped matrixset (see column_group_by()), and section 'Grouped matrixsets'.

Value

A matrixset, with possibly a subset of the columns of the original object. Groups will be updated if .preserve is TRUE.

Grouped matrixsets

Row grouping (row_group_by()) has no impact on column filtering.

The impact of column grouping (column_group_by()) on column filtering depends on the conditions. Often, column grouping will not have any impact, but as soon as an aggregating, lagging or ranking function is involved, then the results will differ.

For instance, the two following are not equivalent (except by pure coincidence).

student_results %>% filter_column(school_average > mean(school_average))

And it's grouped equivalent: student_results %>% column_group_by(program) %>% filter_column(school_average > mean(school_average))

In the ungrouped version, the mean of school_average is taken globally and filter_column keeps columns with school_average greater than this global average. In the grouped version, the average is calculated within each class and the kept columns are the ones with school_average greater than the within-class average.

Examples

```
# Filtering using one condition
filter_column(student_results, program == "Applied Science")

# Filetring using multiple conditions. These are equivalent
filter_column(student_results, program == "Applied Science" & school_average > 0.8)
filter_column(student_results, program == "Applied Science", school_average > 0.8)

# The potential difference between grouped and non-grouped.
filter_column(student_results, school_average > mean(school_average))
student_results |>
  column_group_by(program) |>
  filter_column(school_average > mean(school_average))
```

filter_row *Subset rows using annotation values*

Description

The `filter_row()` function subsets the rows of all matrices of a `matrixset`, retaining all rows that satisfy given condition(s). The function `filter_row` works like `dplyr`'s `dplyr::filter()`.

Usage

```
filter_row(.ms, ..., .preserve = FALSE)
```

Arguments

`.ms`	`matrixset` object to subset based on the filtering conditions
`...`	Condition, or expression, that returns a logical value, used to determine if rows are kept or discarded. The expression may refer to row annotations - columns of the `row_info` component of `.ms` More than one condition can be supplied and if multiple expressions are included, they are combined with the `&` operator. Only rows for which all conditions evaluate to TRUE are kept.
`.preserve`	`logical`, relevant only if `.ms` is row grouped. When `.preserve` is `FALSE` (the default), the row grouping is updated based on the new `matrixset` resulting from the filtering. Otherwise, the row grouping is kept as is.

Details

The conditions are given as expressions in `...`, which are applied to columns of the annotation data frame (`row_info`) to determine which rows should be retained.

It can be applied to both grouped and ungrouped `matrixset` (see `row_group_by()`), and section 'Grouped matrixsets'.

Value

A `matrixset`, with possibly a subset of the rows of the original object. Groups will be updated if `.preserve` is `TRUE`.

Grouped matrixsets

Column grouping (`column_group_by()`) has no impact on row filtering.

The impact of row grouping (`row_group_by()`) on row filtering depends on the conditions. Often, row grouping will not have any impact, but as soon as an aggregating, lagging or ranking function is involved, then the results will differ.

For instance, the two following are not equivalent (except by pure coincidence).

`student_results %>% filter_row(previous_year_score > mean(previous_year_score))`

And it's grouped equivalent: `student_results %>% row_group_by(class) %>% filter_row(previous_year_score > mean(previous_year_score))`

In the ungrouped version, the mean of `previous_year_score` is taken globally and `filter_row` keeps rows with `previous_year_score` greater than this global average. In the grouped version, the average is calculated within each `class` and the kept rows are the ones with `previous_year_score` greater than the within-class average.

Examples

```
# Filtering using one condition
filter_row(student_results, class == "classA")

# Filetring using multiple conditions. These are equivalent
filter_row(student_results, class == "classA" & previous_year_score > 0.75)
filter_row(student_results, class == "classA", previous_year_score > 0.75)

# The potential difference between grouped and non-grouped.
filter_row(student_results, previous_year_score > mean(previous_year_score))
student_results |>
  row_group_by(teacher) |>
  filter_row(previous_year_score > mean(previous_year_score))
```

---

group_by *Group rows/columns of a matrixset by one or more variables*

Description

Applying `row_group_by()` or `column_group_by()` to a `matrixset` object registers this object as one where certain operations are performed per (row or column) group.

To (partly) remove grouping, use `row_ungroup()` or `column_ungroup()`.

These functions are the `matrixset` equivalent of `dplyr`'s `dplyr::group_by()` and `dplyr::ungroup()`

Usage

```
row_group_by(.ms, ..., .add = FALSE, .drop = row_group_by_drop_default(.ms))

column_group_by(
  .ms,
  ...,
  .add = FALSE,
  .drop = column_group_by_drop_default(.ms)
)

row_ungroup(.ms, ...)

column_ungroup(.ms, ...)
```

Arguments

.ms	A matrixset object
...	In row_group_by() or column_group_by(), annotation variables to use for grouping. These variables are the ones returned by row_traits() or column_traits(). In row_ungroup() or column_ungroup(), variables to remove from grouping. If not provided, grouping is removed altogether.
.add	logical. The default, FALSE, means that previous groups are overwritten. Setting .add to TRUE will add to the existing groups.
.drop	logical. When grouping by factor annotations, should levels that do not appear in the data be dropped? The default is TRUE, unless .ms has been previously grouped with .drop = FALSE.

Value

A grouped matrixset with class row_grouped_ms, unless .ms was already column-grouped via column_group_by(), in which case a dual_grouped_ms matrixset is returned.

If the combination of ... and .add yields an empty set of grouping columns, a regular matrixsetor a col_grouped_ms, as appropriate, will be returned.

Examples

```
by_class <- row_group_by(student_results, class)

# On it's own, a grouped `matrixset` looks like a regular `matrixset`, except
# that the grouping structure is listed
by_class

# Grouping changes how some functions operates
filter_row(by_class, previous_year_score > mean(previous_year_score))

# You can group by expressions: you end-up grouping by the new annotation:
row_group_by(student_results, sqrt_score = sqrt(previous_year_score))

# By default, grouping overrides existing grouping
row_group_vars(row_group_by(by_class, teacher))

# Use .add = TRUE to instead append
row_group_vars(row_group_by(by_class, teacher, .add = TRUE))
# To removing grouping, use ungroup
row_ungroup(by_class)
```

---

join *Add meta info from another* matrixset *or a* data.frame

---

Description

The operation is done through a join operation between the row meta info data.frame (join_row_info()) of .ms and y (or its row meta info data.frame if it is a matrixset object). The function join_column_info() does the equivalent operation for column meta info.

The default join operation is a left join (type == 'left'), but most of dplyr's joins are available ('left', 'inner', 'right', 'full', 'semi' or 'anti').

The matrixset paradigm of unique row/column names is enforced so if a .ms data.frame row matches multiple ones in y, this results in an error.

Usage

```
join_row_info(
  .ms,
  y,
  type = "left",
  by = NULL,
  adjust = FALSE,
  suffix = c(".x", ".y"),
  na_matches = c("na", "never")
)

join_column_info(
  .ms,
  y,
  type = "left",
  by = NULL,
  adjust = FALSE,
  suffix = c(".x", ".y"),
  na_matches = c("na", "never")
)
```

Arguments

.ms	A matrixset object
y	A matrixset object or a data.frame.
type	Joining type, one of 'left', 'inner', 'right', 'full', 'semi' or 'anti'.
by	The names of the variable to join by. The default, NULL, results in slightly different behavior depending if y is a matrixset or a data.frame. If a matrixset, the meta info tag of each object (the tag is the column that holds the row names/column names in the meta info data frame - typically ".rowname" or ".colname" unless specified otherwise at matrixset creation) is used for by. If a data.frame, a natural join is used. For more details, see dplyr's dplyr::join(). Note that the cross-join is not available.
adjust	A logical. By default (FALSE), the join operation is not permitted to filter or augment the number of rows of the meta info data frame. If TRUE, this will be allowed. In the case where the data frame is augmented, the matrices of .ms will be augmented accordingly by padding with NAs (except for the NULL matrices).

Alternatively, adjust can be a single string, one of 'pad_x' or 'from_y'. Choosing "pad_x" is equivalent to TRUE. When choosing "from_y", padding is done using values from y, but only

1. if y is a matrixset
2. for y matrices that are named the same in x
3. If padding rows, only columns common between x and y will use y values. The same logic is applied when padding columns.

Other values are padded with NA.

suffix — Suffixes added to disambiguate trait variables. See dplyr's dplyr::join().

na_matches — How to handle missing values when matching. See dplyr's dplyr::join().

Value

A matrixset with updated row or column meta info, with all .ms traits and y traits. If some traits share the same names - and were not included in by - suffixes will be appended to these names.

If adjustment was allowed, the dimensions of the new matrixset may differ from the original one.

Groups

When y is a matrixset, only groups from .ms are used, if any. Group update is the same as in dplyr.

Examples

```
ms1 <- remove_row_annotation(student_results, class, teacher)
ms <- join_row_info(ms1, student_results)

ms <- join_row_info(ms1, student_results, by = c(".rowname", "previous_year_score"))

# This will throw an error
ms2 <- remove_row_annotation(filter_row(student_results, class %in% c("classA", "classC")),
                             class, teacher, previous_year_score)
ms <- ms <- tryCatch(join_row_info(ms2, student_results, type = "full"),
                     error = function(e) e)
is(ms, "error") # TRUE
ms$message

# Now it works.
ms <- join_row_info(ms2, student_results, type = "full", adjust = TRUE)
dim(ms2)
dim(ms)
matrix_elm(ms, 1)
```

loop *Apply functions to each matrix of a matrixset*

Description

The apply_matrix function applies functions to each matrix of a matrixset. The apply_row/apply_column functions do the same but separately for each row/column. The functions can be applied to all matrices or only a subset.

The dfl/dfw versions differ in their output format and when possible, always return a tibble().

Empty matrices are simply left unevaluated. How that impacts the returned result depends on which flavor of apply_* has been used. See 'Value' for more details.

If .matrix_wise is FALSE, the function (or expression) is multivariate in the sense that all matrices are accessible at once, as opposed to each of them in turn.

See section "Multivariate".

Usage

```
apply_row(.ms, ..., .matrix = NULL, .matrix_wise = TRUE, .input_list = FALSE)

apply_row_dfl(
  .ms,
  ...,
  .matrix = NULL,
  .matrix_wise = TRUE,
  .input_list = FALSE,
  .force_name = FALSE
)

apply_row_dfw(
  .ms,
  ...,
  .matrix = NULL,
  .matrix_wise = TRUE,
  .input_list = FALSE,
  .force_name = FALSE
)

apply_column(
  .ms,
  ...,
  .matrix = NULL,
  .matrix_wise = TRUE,
  .input_list = FALSE
)
```

```
apply_column_dfl(
  .ms,
  ...,
  .matrix = NULL,
  .matrix_wise = TRUE,
  .input_list = FALSE,
  .force_name = FALSE
)

apply_column_dfw(
  .ms,
  ...,
  .matrix = NULL,
  .matrix_wise = TRUE,
  .input_list = FALSE,
  .force_name = FALSE
)

apply_matrix(
  .ms,
  ...,
  .matrix = NULL,
  .matrix_wise = TRUE,
  .input_list = FALSE
)

apply_matrix_dfl(
  .ms,
  ...,
  .matrix = NULL,
  .matrix_wise = TRUE,
  .input_list = FALSE,
  .force_name = FALSE
)

apply_matrix_dfw(
  .ms,
  ...,
  .matrix = NULL,
  .matrix_wise = TRUE,
  .input_list = FALSE,
  .force_name = FALSE
)
```

Arguments

.ms	matrixset object
...	expressions, separated by commas. They can be specified in one of the following

way:

- a function name, e.g., `mean`.
- a function call, where you can use .m to represent the current matrix (for `apply_matrix`),.ito represent the current row (forapply_row) and.j for the current column (`apply_column`). Bare names of object traits can be used as well. For instance, `lm(.i ~ program)`.
  The pronouns are also available for the multivariate version, under certain circumstances, but they have a different meaning. See the "Multivariate" section for more details.
- a formula expression. The pronouns .m, .i and .j‘ can be used as well. See examples to see the usefulness of this.

The expressions can be named; these names will be used to provide names to the results.

`.matrix` matrix indices of which matrix to apply functions to. The default, `NULL`, means all the matrices are used.

If not `NULL`, index is numeric or character vectors.

Numeric values are coerced to integer as by `as.integer()` (and hence truncated towards zero).

Character vectors will be matched to the matrix names of the object.

Can also be logical vectors, indicating elements/slices to replace. Such vectors are *NOT* recycled, which is an important difference with usual matrix replacement. It means that the `logical` vector must match the number of matrices in length.

Can also be negative integers, indicating elements/slices to leave out of the replacement.

`.matrix_wise` `logical`. By default (`TRUE`), matrices are provided one by one, in turn, to the functions/expressions. But if `.matrix_wise` is `FALSE`, the functions/expressions have access to all matrices. See "Multivariate" for details, including how to reference the matrices.

`.input_list` `logical`. If multivariate (`.matrix_wise == FALSE`), the matrices are provided as a single list, where each element is a matrix (or matrix row or column). The list elements are the matrix names.

`.force_name` `logical`. Used only for the simplified output versions (dfl/dfw). By default (`FALSE`), function IDs will be provided only if the function outcome is a vector of length 2 or more. If `.force_name` is `TRUE` then function IDs are provided in all situations.

This can be useful in situation of grouping. As the functions are evaluated independently within each group, there could be situations where function outcomes are of length 1 for some groups and lenght 2 or more in other groups.

See examples.

Value

A list for every matrix in the matrixset object. Each list is itself a list. For `apply_matrix`, it is a list of the function values - `NULL` if the matrix was empty. Otherwise, it is a list with one element for

each row/column - these elements will be NULL if the corresponding matrix was empty. And finally, for apply_row/apply_column, each of these sub-list is a list, the results of each function.

If each function returns a vector of the same dimension, you can use either the _dfl or the _dfw version. What they do is to return a list of tibbles. The dfl version will stack the function results in a long format while the dfw version will put them side-by-side, in a wide format. An empty matrix will be returned for empty input matrices.

If the functions returned vectors of more than one element, there will be a column to store the values and one for the function ID (dfl), or one column per combination of function/result (dfw)

See the grouping section to learn about the result format in the grouping context.

Pronouns

The rlang pronouns .data and .env are available. Two scenarios for which they can be useful are:

- The annotation names are stored in a character variable. You can make use of the variable by using .data[[var]]. See the example for an illustration of this.
- You want to make use of a global variable that has the same name as an annotation. You can use .env[[var]] or .env$var to make sure to use the proper variable.

The matrixset package defines its own pronouns: .m, .i and .j, which are discussed in the function specification argument (...).

It is not necessary to import any of the pronouns (or load rlang in the case of .data and .env) in a interactive session.

It is useful however when writing a package to avoid the R CMD check notes. As needed, you can import .data and .env (from rlang) or any of .m, .i or .j from matrixset.

Multivariate

The default behavior is to apply a function or expression to a single matrix and each matrices of the matrixset object are provided sequentially to the function/expression.

If .matrix_wise is FALSE, all matrices are provided at once to the functions/expressions. They can be provided in two fashions:

- separately (default behavior). Each matrix can be referred by .m1, ..., .mn, where n is the number of matrices. Note that this is the number as determined by .matrix.

  For apply_row (and dfl/dfw variants), use .i1, .i2 and so on instead. What the functions/expressions have access to in this case is the first row of the first matrix, the first row of the second matrix and so on. Then, continuing the loop, the second row of each matrix will be accessible, and so on

  Similarly, use .j1 and so on for the apply_column family.

  Anonymous functions will be understood as a function with multiple arguments. In the example apply_row(ms, mean, .matrix_wise = FALSE), if there are 3 matrices in the ms object, mean is understood as mean(.i1, .i2, .i3). Note that this would fail because of the mean function.
- In a list (.list_input = TRUE). The list will have an element per matrix. The list can be referred using the same pronouns (.m, .i, .j), and the matrix, by the matrix names or position.

For the multivariate setting, empty matrices are given as is, so it is important that provided functions can deal with such a scenario. An alternative is to skip the empty matrices with the `.matrix` argument.

Grouped matrixsets

If groups have been defined, functions will be evaluated within them. When both row and column grouping has been registered, functions are evaluated at each cross-combination of row/column groups.

The output format is different when the `.ms` matrixset object is grouped. A list for every matrix is still returned, but each of these lists now holds a tibble.

Each tibble has a column called `.vals`, where the function results are stored. This column is a list, one element per group. The group labels are given by the other columns of the tibble. For a given group, things are like the ungrouped version: further sub-lists for rows/columns - if applicable - and function values.

The dfl/dfw versions are more similar in their output format to their ungrouped version. The format is almost identical, except that additional columns are reported to identify the group labels.

See the examples.

Examples

```
# The firs example takes the whole matrix average, while the second takes
# every row average
(mn_mat <- apply_matrix(student_results, mean))
(mn_row <- apply_row(student_results, mean))

# More than one function can be provided. It's a good idea in this case to
# name them
(mn_col <- apply_column(student_results, avr=mean, med=median))

# the dfl/dfw versions returns nice tibbles - if the functions return values
# of the same length.
(mn_l <- apply_column_dfl(student_results, avr=mean, med=median))
(mn_w <- apply_column_dfw(student_results, avr=mean, med=median))

# There is no difference between the two versions for length-1 vector results.
# hese will differ, however
(rg_l <- apply_column_dfl(student_results, rg=range))
(rg_w <- apply_column_dfw(student_results, rg=range))

# More complex examples can be used, by using pronouns and data annotation
(vals <- apply_column(student_results, avr=mean, avr_trim=mean(.j, trim=.05),
                                      reg=lm(.j ~ teacher)))

# You can wrap complex function results, such as for lm, into a list, to use
# the dfl/dfr version
(vals_tidy <- apply_column_dfw(student_results, avr=mean, avr_trim=mean(.j, trim=.05),
                                               reg=list(lm(.j ~ teacher))))

# You can provide complex expressions by using formulas
```

```
(r <- apply_column(student_results,
                                  res= ~ {
                                    log_score <- log(.j)
                                    p <- predict(lm(log_score ~ teacher + class))
                                    .j - exp(p)
                                  }))

# the .data pronoun can be useful to use names stored in variables
fn <- function(nm) {
  if (!is.character(nm) && length(nm) != 1) stop("this example won't work")
  apply_column(student_results, lm(.j ~ .data[[nm]]))
}
fn("teacher")

# You can use variables that are outside the scope of the matrixset object.
# You don't need to do anything special if that variable is not named as an
# annotation
pass_grade <- 0.5
(passed <- apply_row_dfw(student_results, pass = ~ .i >= pass_grade))

# use .env if shares an annotation name
previous_year_score <- 0.5
(passed <- apply_row_dfw(student_results, pass = ~ .i >= .env$previous_year_score))

# Grouping structure makes looping easy. Look at the output format
cl_prof_gr <- row_group_by(student_results, class, teacher)
(gr_summ <- apply_column(cl_prof_gr, avr=mean, med=median))
(gr_summ_tidy <- apply_column_dfw(cl_prof_gr, avr=mean, med=median))
# to showcase how we can play with format
(gr_summ_tidy_long <- apply_column_dfl(cl_prof_gr, summ = ~ c(avr=mean(.j), med=median(.j))))

# It is even possible to combine groupings
cl_prof_program_gr <- column_group_by(cl_prof_gr, program)
(mat_summ <- apply_matrix(cl_prof_program_gr, avr = mean, med = median, rg = range))
# it doesn' make much sense, but this is to showcase format
(summ_gr <- apply_matrix(cl_prof_program_gr, avr = mean, med = median, rg = range))
(summ_gr_long <- apply_column_dfl(cl_prof_program_gr,
                                  ct = ~ c(avr = mean(.j), med = median(.j)),
                                  rg = range))
(summ_gr_wide <- apply_column_dfw(cl_prof_program_gr,
                                  ct = c(avr = mean(.j), med = median(.j)),
                                  rg = range))


# This is an example where you may want to use the .force_name argument
(apply_matrix_dfl(column_group_by(student_results, program), FC = colMeans(.m)))
(apply_matrix_dfl(column_group_by(student_results, program), FC = colMeans(.m),
                  .force_name = TRUE))
```

---

matrixset *Matrix Set*

Description

Creates a matrix set, possibly annotated for rows and/or columns. These annotations are referred as traits.

Usage

```
matrixset(
  ...,
  expand = NULL,
  row_info = NULL,
  column_info = NULL,
  row_key = "rowname",
  column_key = "colname",
  row_tag = ".rowname",
  column_tag = ".colname"
)
```

Arguments

...	A single list of matrices (must be a named list), or individual matrices, e.g. mat1 = m1, mat2 = m2, etc. NULL elements are accepted. This allows to create a placeholder that can be filled later on.
expand	By default (NULL), input matrix expansion is disabled. Setting this parameter to TRUE will enable the expansion feature. See the section 'Matrix Expansion' for more details of what it is, as well as other possible options for expand. The section will also detail how the default expansion value is dependent on the matrix types.
row_info	a data frame, used to annotate matrix rows. The link between the matrix row names and the data frame is given in column "rowname". A different column can be used if one provides a different row_key.
column_info	a data frame, used to annotate matrix columns. The link between the matrix column names and the data frame is given in column "colname". A different column can be used if one provides a different column_key.
row_key	column name in 'row_info" data frame that will link the row names with the row information. A string is expected.
column_key	column name in col_info data frame that will link the column names with the row information. A string is expected.
row_tag	A string, giving the row annotation data frame column that will link the row names to the data frame. While row_key specifies the column name of the data frame at input, row_tag specifies the column name that will be used throughout in the matrixset object.
column_tag	A string, giving the column annotation data frame column that will link the row names to the data frame. While column_key specifies the column name of the data frame at input, column_tag specifies the column name that will be used throughout in the matrixset object.

Details

A `matrixset` is a collection of matrices that share the same dimensions and, if applicable, dimnames. It is designed to hold different measures for the same rows/columns. For example, each matrix could be a different time point for the same subjects.

Traits, which are annotations, can be provided in the form of data frames for rows and/or columns. If traits are provided, the `data.frame` must contain only one entry per row/column (see examples).

Row or column names are not mandatory to create a proper `matrixset`. The only way for this to work however is to leave traits (annotations) empty. If provided, each matrices must have the same dimnames as well.

If dimnames are missing, note that most of the operations for matrixsets won't be available. For instance, operations that use traits will not work, e.g., `filter_row()`.

It is allowed for matrix elements of a `matrixset` to be `NULL` - see examples.

Value

Returns a `matrixset`, a collection of matrices (see 'Details').

Matrix Expansion

The concept of matrix expansion allows to provide input matrices that do not share the same dimensions.

This works by taking the union of the dimnames and padding, if necessary, each matrix with a special value for the missing rows/columns.

Because the dimnames are used, they must necessarily be non-`NULL` in the provided matrices.

An interesting side-effect is that one can use this option to match the dimnames and provide a common row/column order among the matrices.

For base matrices, the padding special value is, by default (`expand = TRUE`), `NA`. For the special matrices (Matrix package), the default value is `0`. For these special matrices, padding with 0 forces conversion to sparse matrix.

The default value can be changed by providing any value (e.g, `-1`) to `expand`, in which case the same padding value is used for all matrices.

If different padding values are needed for each matrices, a list can be provided to `expand`. If the list is unnamed, it must match the number of input matrices in length and the padding values are assigned to the matrices in order.

A named list can be provided as well. In that case, `expand` names and matrix names are matched. All matrices must have a match in the `expand` list (more `expand` values can be provided, though).

See Also

`as_matrixset()`

Examples

```
# A single NULL element will create an empty matrixset (it doesn't hold
# any matrices)
lst <- NULL
matrixset(lst)

# This will hold to empty matrices
lst <- list(a = NULL, b = NULL)
matrixset(lst)
# this is equivalent
matrixset(a = NULL, b = NULL)

# A basic example
lst <- list(a = matrix(0, 2, 3))
matrixset(lst)
# equivalent
matrixset(a = matrix(0, 2, 3))

# can mix with NULL too
lst <- list(a = NULL, b = matrix(0, 2, 3), c = matrix(0, 2, 3))
matset <- matrixset(lst)

# dimnames are also considered to be traits
lst <- list(a = NULL, b = matrix(0, 2, 3), c = matrix(0, 2, 3))
rownames(lst$b) <- c("r1", "r2")
rownames(lst$c) <- c("r1", "r2")
matrixset(lst)

# You don't have to annotate both rows and columns. But you need to provide
# the appropriate dimnames when you provide traits
lst <- list(a = matrix(0, 2, 3), b = matrix(0, 2, 3), c = NULL)
rownames(lst$a) <- c("r1", "r2")
rownames(lst$b) <- c("r1", "r2")
colnames(lst$a) <- c("c1", "c2", "c3")
colnames(lst$b) <- c("c1", "c2", "c3")
ri <- data.frame(rowname = c("r1", "r2"), g = 1:2)
matset <- matrixset(lst, row_info = ri)

# You can provide a column name that contains the keys
ri <- data.frame(foo = c("r1", "r2"), g = 1:2)
matset <- matrixset(lst, row_info = ri, row_key = "foo")

lst <- list(a = matrix(0, 2, 3), b = matrix(0, 2, 3), c = NULL)
rownames(lst$a) <- c("r1", "r2")
rownames(lst$b) <- c("r1", "r2")
colnames(lst$a) <- c("c1", "c2", "c3")
colnames(lst$b) <- c("c1", "c2", "c3")
ri <- data.frame(rowname = c("r1", "r2"), g = 1:2)
ci <- data.frame(colname = c("c1", "c2", "c3"), h = 1:3)
matset <- matrixset(lst, row_info = ri, column_info = ci)

# This is not allowed, because the row trait data frame has more than one
```

```
# entry for "r1"
lst <- list(a = matrix(0, 2, 3), b = matrix(0, 2, 3), c = NULL)
rownames(lst$a) <- c("r1", "r2")
rownames(lst$b) <- c("r1", "r2")
colnames(lst$a) <- c("c1", "c2", "c3")
colnames(lst$b) <- c("c1", "c2", "c3")
ri <- data.frame(rowname = c("r1", "r2", "r1"), g = 1:3)
ci <- data.frame(colname = c("c1", "c2", "c3"), h = 1:3)
ans <- tryCatch(matrixset(lst, row_info = ri, column_info = ci),
                error = function(e) e)
is(ans, "error")
```

`meta` *Matrixset group metadata*

Description

- `row_group_meta()` and `column_group_meta()` returns the grouping structure, in a data frame format. See `dplyr`'s `dplyr::group_data()`, from which the functions are based. Returns `NULL` for ungrouped `matrixsets`.
- `row_group_keys()` and `column_group_keys()` retrieve the grouping data, while the locations (row or column indices) are retrieved with `row_group_where()` and `column_group_where()`.
- `row_group_indices()` and `column_group_indices()` each return an integer vector the same length as the number of rows or columns of `.ms`, and gives the group that each row or column belongs to.
- `row_group_vars()` and `column_group_vars()` give names of grouping variables as character vector; `row_groups()` and `column_groups()` give the names as a list of symbols.

Usage

```
row_group_meta(.ms)

row_group_vars(.ms)

row_group_keys(.ms)

row_group_where(.ms)

row_group_indices(.ms)

row_groups(.ms)

column_group_meta(.ms)

column_group_vars(.ms)
```

```
column_group_keys(.ms)

column_group_where(.ms)

column_group_indices(.ms)

column_groups(.ms)
```

Arguments

.ms	a matrixset object

---

mrm_plus2015 *Table S1 and S2 of MRMPlus Paper in* matrixset *Format*

Description

Table S1 and S2 of MRMPlus Paper in matrixset Format

Usage

```
mrm_plus2015
```

Format

A matrixset of 30 rows and 45 columns The object contains four matrices:

light_area Peak area of light peptides.

heavy_area Peak area of heavy peptides.

light_rt Retention time of light peptides.

heavy_rt Retention time of heavy peptides.

The column names, analytes, are a combination of peptide sequence and fragment ion. Rownames are the replicate names.

Source

Aiyetan P, Thomas SN, Zhang Z, Zhang H. MRMPlus: an open source quality control and assessment tool for SRM/MRM assay development. BMC Bioinformatics. 2015 Dec 12;16:411. doi: 10.1186/s12859-015-0838-z. PMID: 26652794; PMCID: PMC4676880.

ms_to_df *Convert matrixset to data frame*

Description

Converts a `matrixset` to a `data.frame` (a `tibble`, more specifically), in a long format.

When `as_list` is `TRUE`, each matrix is converted separately. Row/column annotation is included if requested.

Usage

```
ms_to_df(
  .ms,
  add_row_info = TRUE,
  add_column_info = TRUE,
  as_list = FALSE,
  .matrix = NULL
)
```

Arguments

`.ms`	`matrixset` object to convert to `data.frame`
`add_row_info`	`logical`, to include row annotation or not
`add_column_info`	`logical`, to include column annotation or not
`as_list`	`logical`. By default (`FALSE`), a single tibble is returned with matrices as columns. When `TRUE`, the list structure, an element by converted matrix, is kept.
`.matrix`	matrix indices of which matrix to include in the conversion. The default, `NULL`, means all the matrices are used. If not `NULL`, index is numeric or character vectors. Numeric values are coerced to integer as by `as.integer()` (and hence truncated towards zero). Character vectors will be matched to the matrix names of the object. Can also be logical vectors, indicating elements/slices to replace. Such vectors are *NOT* recycled, which is an important difference with usual matrix replacement. It means that the `logical` vector must match the number of matrices in length. Can also be negative integers, indicating elements/slices to leave out of the replacement.

Value

A tibble, or if `as_list` is `TRUE`, A `list` of data frames, an element per converted matrix

Examples

```
# includes both annotation
ms_to_df(student_results)

# includes only row annotation
ms_to_df(student_results, add_column_info = FALSE)
```

`mutate_matrix` *Create/modify/delete matrices from a* `matrixset` *object*

Description

Applies functions that takes matrices as input and return similar matrices. The definition of similar is that the new matrix has the same dimension and dimnames as `.ms`.

If the returned matrix is assigned to a new matrix, this matrix is added to the `matrixset` object. If it is assigned to an already existing matrix, it overwrites the matrix of the same name.

Setting a matrix value to `NULL` will **not** delete the matrix, but will create an empty slot (`NULL`) for the matrix.

To delete a matrix, use the function `remove_matrix()`. See examples below.

Note that matrices are created sequentially and can be used by other name-value pairs. There is an example that showcases this.

Usage

```
mutate_matrix(.ms, ...)
```

Arguments

`.ms`	A `matrixset` object.
`...`	Name-value pairs, ala `dplyr`'s `dplyr::mutate()`. The value can be one of: • a `matrix`, with same dimension and dimnames as `.ms`. • `NULL`, which will turn the matrix as an empty placeholder. • `remove_matrix()`, to remove the matrix

Value

A `matrixset` with updated matrices.

Examples

```
# Notice how FC can be used as soon as created
ms <- mutate_matrix(student_results,
                    FC = remedial/failure,
                    foo = NULL,
                    logFC = log2(FC),
                    FC = remove_matrix())
# this is NULL
matrix_elm(ms, "foo")

# running this would return an error, since FC was deleted
# matrix_elm(ms, "FC")
```

`print.matrixset` *Print a matrixset*

Description

When printing a `matrixset`:

- The number of matrices and their dimension is shown
- Prints each matrix of the object, showing its type and dimension. Full matrices are shown only for those with 3 rows or less. Otherwise, only the first and last row is shown. The same also applies for the columns.
- An exception to the point above: if the number of matrices is greater than `n_matrices`, the first `n_matrices` are displayed, while the others will be named only.
- The row and column annotations (`row_info`/`column_info`) are displayed as `tibble` objects.

Usage

```
## S3 method for class 'matrixset'
print(x, ..., n_matrices = 2)
```

Arguments

`x`	`matrixset` object to print
`...`	currently not used
`n_matrices`	Number of matrices to display

Value

Invisibly, the `matrixset` object.

Examples

```
print(student_results)
print(mrm_plus2015)
```

properties *Matrixset properties*

Description

Utility functions to extract relevant information from a `matrixset` object.

Usage

```
## S3 method for class 'matrixset'
dim(x)

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

## S3 replacement method for class 'matrixset'
dimnames(x) <- value

matrixnames(x)

matrixnames(x) <- value

matrix_elm(x, matrix)

matrix_elm(x, matrix) <- value

nmatrix(x)

row_traits(x)

column_traits(x)

row_traits(x) <- value

column_traits(x)

column_traits(x) <- value

row_tag(x)

column_tag(x)

row_info(x)

row_info(x) <- value

column_info(x)
```

```
column_info(x) <- value

is_matrixset(x)
```

Arguments

x	matrixset object from which to retrieve information, or object to test for being a matrixset.
value	valid value for replacement
matrix	index specifying matrix or matrices to extract. Index is numeric or character vectors or empty (NULL). Note that treating NULL as empty differs from the usual extraction, where it is treated as integer(0). Here a NULL (empty) results in selecting all matrices.

Details

is_matrixset tests if its argument is a proper matrixset object.

dim retrieves the dimension of the matrixset matrices (which are the same for reach). Similarly,nrow returns the number of rows for each matrices, and ncol returns the number of columns.

dimnames retrieves the dimnames of the matrixset matrices (which are the same for reach). Similarly, rownames (colnames) will retrieve row (column) names.

matrixnames retrieves the matrix names, or NULL if the matrices are not named.

nmatrix returns the number of matrices of a matrixset.

row_traits returns the object's row traits; these are the column names of the row annotation data frame.

column_traits returns the object's column traits; these are the column names of the column annotation data frame.

row_info extracts the row annotation data frame. column_info does the same thing for column annotation.

row_tag returns the column name of row_info that stores the matrixset's row names. column_tag returns the column name of column_info that stores the matrixset's column names.

The replacement methods for row_traits/row_info and column_traits/column_info can potentially change meta variables that were used for grouping. There is always an attempt to keep the original groups, but they will be updated if it is possible - a message is issued when that happens - and otherwise removed altogether, with a warning.

matrix_elm extracts a single matrix. It's a wrapper to x[,,matrix], but returns the matrix element. The replacement method matrix_elm is also a wrapper to x[,,matrix] <-.

Value

is_matrixset returns a logical.

dim returns a length-2 vector; nrow and ncol return length-1 vector.

dimnames returns a length-2 list; one component for each dimnames (row and column). rownames and colnames each returns a character vector of names.

`matrixnames` a`character` vector of matrix names, or `NULL`.

`nmatrix` returns an `integer`.

`row_traits` and `column_traits` returns a `character` vector.

`row_tag` and `column_tag` returns a `character` vector.

`row_info` extracts the row annotation data frame. `column_info` does the same thing for column annotation.

Examples

```
is_matrixset(student_results)
dim(student_results)
c(nrow(student_results), ncol(student_results))
dimnames(student_results)
list(rownames(student_results), colnames(student_results))
matrixnames(student_results)
nmatrix(student_results)
list(row_traits(student_results), column_traits(student_results))
row_info(student_results)
column_info(student_results)
```

---

remove_anno *Remove meta info of a* `matrixset`

Description

Deletes row or column annotation (i.e., trait).

The tag is a special trait that can't be removed. The tag is the column name of the meta data frame that holds the row or column names. The tag identity of the' object can be obtained via `row_tag()` or `column_tag()`.

Usage

```
remove_row_annotation(.ms, ...)

remove_column_annotation(.ms, ...)
```

Arguments

`.ms`	A `matrixset` object
`...`	Name of traits to remove. Tidy selection is supported.

Value

A `matrixset` with updated row or column meta info.

Groups

Removing a trait that is used for grouping is not allowed and will not work.

Examples

```
ms1 <- remove_row_annotation(student_results, class, teacher)

# this doesn't work because "class" is used for grouping
ms2 <- tryCatch(remove_row_annotation(row_group_by(student_results, class), class),
                error = function(e) e)
is(ms2, "error") #TRUE
ms2$message
```

---

remove_matrix *Remove one or more matrices of the* matrixset *object*

Description

This is a special case of the [method, with the benefit of being explicit about what action is taken.

Usage

```
remove_matrix(.ms, matrix)
```

Arguments

.ms	A matrixset object. Leave empty only if remove_matrix() is used inside mutate_matrix().
matrix	index specifying matrix or matrices to remove. Index is *posivie* numeric or character vectors. Tidy select is also supported .Leave empty only if remove_matrix() is used inside mutate_matrix().

Value

A matrixset with updated matrices.

Usage inside mutate_matrix()

In most cases, both arguments of the function are mandatory. However, if you want to declare that a matrix should be removed via the mutate_matrix() function, the remove_matrix() must be called without arguments. There is an example that illustrates that.

Examples

```
ms1 <- remove_matrix(student_results, "remedial")
ms2 <- remove_matrix(student_results, 2)
ms3 <- mutate_matrix(student_results, remedial = remove_matrix())
```

row_group_by_drop_default *Default value for .drop argument of function row_group_by()*

Description

Default value for `.drop` argument of function `row_group_by()`

Usage

```
row_group_by_drop_default(.ms)
```

Arguments

`.ms`	a `matrixset` object

Value

Returns `TRUE` for row-ungrouped `matrixsets`. For row-grouped objects, the default is also `TRUE` unless `.ms` has been previously grouped with `.drop = FALSE`.

Examples

```
student_results |>
    row_group_by(class, .drop = FALSE) |>
    row_group_by_drop_default()
```

student_results *Fake Final Exam Results of School Students Before and After Remedial Courses*

Description

Fake Final Exam Results of School Students Before and After Remedial Courses

Usage

```
student_results
```

Format

A `matrixset` of 20 rows and 3 columns The object contains two matrices, one for the failure results (matrix named `failure`) and one for the results after remedial classes (matrix named `remedial`). Each matrix has results for 20 students and 3 classes:

- Mathematics
- English
- Science

The object has been annotated both for rows (students) and columns (courses). Each students has been annotated for the following information:

class Group, or class, in which the student was part of

teacher Professor that gave the remedial course

previous_year_score Score the student had in the previous level of the same class

Each course has been annotated for the following information:

national_average National average of all students for the course

school_average Average of the school's students for the course

program Program in which the course is given

`subsetting` *Subsetting matrixsets*

Description

Extract parts of a matrixset, where indexes refers to rows and columns.

Usage

```
## S3 method for class 'matrixset'

  x[
  i = NULL,
  j = NULL,
  matrix = NULL,
  drop = FALSE,
  keep_annotation = TRUE,
  warn_class_change = getOption("matrixset.warn_class_change")
]

## S3 method for class 'row_grouped_ms'

  x[
  i = NULL,
```

```
  j = NULL,
  matrix = NULL,
  drop = FALSE,
  keep_annotation = TRUE,
  warn_class_change = getOption("matrixset.warn_class_change")
]

## S3 method for class 'col_grouped_ms'

  x[
  i = NULL,
  j = NULL,
  matrix = NULL,
  drop = FALSE,
  keep_annotation = TRUE,
  warn_class_change = getOption("matrixset.warn_class_change")
]

## S3 method for class 'dual_grouped_ms'

  x[
  i = NULL,
  j = NULL,
  matrix = NULL,
  drop = FALSE,
  keep_annotation = TRUE,
  warn_class_change = getOption("matrixset.warn_class_change")
]

## S3 method for class 'matrixset'
x$matrix

## S3 method for class 'matrixset'
x[[matrix]]
```

Arguments

x	`matrixset` object from which to extract element(s)
i, j	rows (`i`) and columns (`j`) to extract from matrices of `x`, as indices. These are either `numeric` or `character` values. To extract every rows or columns, use `i = NULL` or `j = NULL`, which is the default for both. Note that treating `NULL` as empty differs from the usual extraction, where it is treated as `integer(0)`. Numeric values are coerced to integer through `as.integer()`, which means they are truncated towards zero. Character vectors will be matched to the dimnames of the object. Indices an also be logical vectors, stating for each element if it is extracted (`TRUE`) or rejected (`FALSE`). Logical vectors are *NOT* recycled, which is an im-

portant difference with usual matrix extraction. It means that the `logical` vector must match the object dimension in length.

Can also be negative integers, in which case they are indices of elements to leave out of the selection.

When indexing, a single argument `i` can be a matrix with two columns. This is treated as if the first column was the `i` index and the second column the `j` index.

`matrix` index specifying matrix or matrices to extract. Index is numeric or character vectors or empty (`NULL`). Note that treating `NULL` as empty differs from the usual extraction, where it is treated as `integer(0)`. Here a `NULL` (empty) results in selecting all matrices.

See arguments `i,j`, as the same rules are followed.

`drop` If `TRUE`, the `drop` option of matrix extraction is used. See [[()]. Note that the default for `matrixset` objects is `FALSE`.

`keep_annotation`

`logical` specifying if the resulting object should keep the annotations (meta info, or traits, as per `matrixset` notation) as part of the object. The default (`TRUE`), combined with the default `drop = FALSE`, guarantees that the resulting object is a `matrixset` object. If `keep_annotation` is `FALSE`, the resulting object will be a list of matrix, and a warning will be issued, unless `warn_class_change` is `FALSE`.

`warn_class_change`

`logical` that determines if a warning should be issued when the extraction result is not a `matrixset`. The default is to use the global option `"matrixset.warn_class_change"`, which is `FALSE` by default. If one wants to silence permanently this warning, this is the option to change.

Details

Indexes `i` and `j` are given as for a regular `matrix()` (note however that factors are currently not allowed for indexing). Which matrices are extracted (all or a subset) is specified via argument `"matrix"`.

Missing values (`NA`) are not allowed for indexing, as it results in unknown selection. Character indexes use exact matching, not partial.

The default arguments for `"drop"` and `"keep_annotation"` are chosen so that the object resulting from the extraction is still a `matrixset`.

Setting `"keep_annotation"` to `FALSE` automatically results in a class change (a list of matrix) and a warning is issued (see argument `warn_class_change`, however).

Setting `drop` to `TRUE` may also result to a change of class, depending on the provided indices (the same way matrix may result to a vector when `drop` is `TRUE`).

The subsetting operator `[[` is a convenient wrapper for `[(,,matrix)`.

There is no `$` subsetting operator for the `matrixset` object.

Value

The resulting object type depends on the subsetting options. By default, a `matrixset` object will be returned. This object will have the following properties:

- Rows and/or columns are a subset of the input (based on what has been subsetted), but appear in the same order.
- Annotations, or traits, are subsetted appropriately.
- The number of groups may be reduced.
- Currently, attributes are *not* preserved.

If `keep_annotation` is `FALSE`, the resulting object will be a list. Typically, it will be a list of `matrix`, but if `drop` is `TRUE`, some list elements could be vectors.

Grouped matrixset

When subsetting a grouped `matrixset` (by rows and/or columns), when the resulting object is still a `matrixset`, the grouping structure will be updated based on the resulting data.

Examples

```
lst <- list(a = matrix(1:6, 2, 3), b = matrix(101:106, 2, 3), c = NULL)
rownames(lst$a) <- rownames(lst$b) <- c("r1", "r2")
colnames(lst$a) <- colnames(lst$b) <- c("c1", "c2", "c3")
ri <- data.frame(rowname = c("r1", "r2"), g = 1:2)
ci <- data.frame(colname = c("c1", "c2", "c3"), h = 1:3)
matset <- matrixset(lst, row_info = ri, column_info = ci, row_tag = "foo", column_tag = "bar")

# this doesn't subset anything, just returns matset again
matset[]

# this extracts the first row of every matrix. Note how each matrices is
# still a matrix, so you still end up with a matrixset object. Note also
# that you need placeholder for j and matrix index, even when not provided
matset[1, , ]

# similar idea
matset[,2, ]
matset[1,2,]

# it obviously works with vector indexes
matset[1:2, c(1,3),]

# you can extract the matrices this - even without the 'annoying' warning
matset[, , , keep_annotation = FALSE]
matset[, , , keep_annotation = FALSE, warn_class_change = FALSE]

# extracts subsetted matrices (no annotations)
matset[1, , , keep_annotation = FALSE, warn_class_change = FALSE]

# a bit more in line with how R subsets matrices
matset[1, , , drop = TRUE, warn_class_change = FALSE]

# you can obviously get some of the matrices only
matset[,,1]
matset[c(1,2),,1:2]
```

```
# to showcase other kind of indexes. These are all equivalents
matset[1,,]
matset["r1", ,]
matset[c(TRUE, FALSE), ,]
matset[-2, ,] # equivalent because there are only 2 rows

# this is also equivalent
matset[,,1]
matset[[1]]
```

`[<-.matrixset` *Replace Parts of a matrixset*

Description

Replace whole or parts of some - or all - matrices of a `matrixset`.

Usage

```
## S3 replacement method for class 'matrixset'
x[i = NULL, j = NULL, matrix = NULL] <- value
```

Arguments

`x`	`matrixset` object from which to replace element(s)

`i, j`	Indices specifying elements to replace. Indices are numeric or character vectors or empty (`NULL`). Note that treating `NULL` as empty differs from the usual replacement, where it is treated as `integer(0)`. Here a `NULL` (empty) results in selecting all rows or columns.
	
	`Numeric values are coerced to integer as by [as.integer()] (and hence truncated towards zero).`
	
	`Character vectors will be matched to the dimnames of the object.`
	
	`Can also be logical vectors, indicating elements/slices to replace Such vectors are **NOT** recycled, which is an important difference with usual matrix replacement. It means that the logical vector must match the object dimension in length.`
	
	`Can also be negative integers, indicating elements/slices to leave out of the replacement.`
	
	`When indexing, a single argument `i` can be a matrix with two columns. This is treated as if the first column was the `i` index and the second column the `j` index.`

matrix	index specifying matrix or matrices to replace. Index is numeric or character vectors or empty (NULL). Note that treating NULL as empty differs from the usual replacement, where it is treated as integer(0). Here a NULL (empty) results in replacing all matrices. Numeric values are coerced to integer as by as.integer() (and hence truncated towards zero). Character vectors will be matched to the matrix names of the object. Can also be logical vectors, indicating elements/slices to replace. Such vectors are *NOT* recycled, which is an important difference with usual matrix replacement. It means that the logical vector must match the number of matrices in length. Can also be negative integers, indicating elements/slices to leave out of the replacement.
value	object to use as replacement value

Details

If matrix is left unspecified (or given as NULL), all matrices will be replaced by value. How replacement exactly occurs depends on value itself.

If value is a single atomic vector (this excludes lists) or matrix, relevant subscripts of all requested matrices will be replaced by the same value. This is conditional to the dimensions being compatible.

Alternatively, value can be a list of atomic vectors/matrices. If value has a single element, the same rules as above apply. Otherwise, the length of value must match the number of matrices for which subscripts have to be replaced.

If the list elements are named, the names are matched to the names of the matrices that need replacement - in which case value needs not to be the same length.

A final possibility for value is for it to be NULL. In this case, target matrices are turned to NULL.

Value

A matrixset, with proper elements replaced.

vector value

Contrarily to matrix replacement, when submitting an atomic vector value, dimensions must match exactly.

Replacing NULL matrices

Replacing subscripts of NULL matrices is not possible, unless value is itself NULL, or a matrix the same dimensions (number of rows and columns) as x. If x has dimnames, value must have the same dimnames.

Examples

```
# an hypothetical example of students that failed 3 courses and their results
# after remedial class

# you can replace a line for all matrices at once. In the example, the "wrong"
# tag refers to the fact that the 'failure' results do not make sense after
# replacement
student_results_wrong <- student_results
student_results_wrong["student 2",,] <- c(0.81, 0.88, 0.71) # obviously, integer index works too
# note how all matrices had the same replacement
student_results_wrong

# this already makes more sense in the context of the example
student_results[2,,] <- list(c(0,0.45,0.1), c(0.81, 0.88, 0.71))
student_results

# or even these two equivalent commands
student_results["student 2",,"remedial"] <- c(0.77, 0.83, 0.75)
student_results[2,,2] <- matrix(c(0.77, 0.83, 0.75), 1, 3)
```

Index