

# Package ‘matsindf’

August 18, 2023

**Type** Package

**Title** Matrices in Data Frames

**Version** 0.4.4

**Date** 2023-08-18

**Maintainer** Matthew Heun <matthew.heun@me.com>

**Description** Provides functions to collapse a tidy data frame into matrices in a data frame and expand a data frame of matrices into a tidy data frame.

**License** MIT + file LICENSE

**Language** en-US

**Encoding** UTF-8

**LazyData** true

**RoxygenNote** 7.2.3

**Depends** R (>= 2.10)

**Imports** assertthat, dplyr, lifecycle, magrittr, matsbyname, purrr, rlang, tibble, tidyr

**Suggests** covr, ggplot2, Hmisc, knitr, RCLabels, rmarkdown, spelling, testthat

**Config/testthat/edition** 3

**VignetteBuilder** knitr

**URL** <https://github.com/MatthewHeun/matsindf>

**BugReports** <https://github.com/MatthewHeun/matsindf/issues>

**NeedsCompilation** no

**Author** Matthew Heun [aut, cre] (<<https://orcid.org/0000-0002-7438-214X>>)

**Repository** CRAN

**Date/Publication** 2023-08-18 16:42:32 UTC

**R topics documented:**

add_UKEnergy2000_matnames	2
add_UKEnergy2000_row_col_meta	3
build_keep_args	5
build_matsindf_apply_data_frame	5
collapse_to_matrices	6
df_to_msg	8
everything_except	9
expand_to_tidy	10
get_useable_default_args	11
group_by_everything_except	12
handle_empty_data	13
handle_null_args	14
index_column	14
matrix_cols	16
matsindf_apply	17
matsindf_apply_types	19
mat_to_rowcolval	20
rowcolval_to_mat	22
should_unlist	23
UKEnergy2000	24
verify_cols_missing	25
where_to_get_args	26
<b>Index</b>	<b>28</b>

---

add\_UKEnergy2000\_matnames

*Add a column of matrix names to tidy data frame*

---

**Description**

Add a column of matrix names to tidy data frame

**Usage**

```
add_UKEnergy2000_matnames(
  .DF,
  ledger_side_colname = "Ledger.side",
  energy_colname = "E.ktoe",
  supply_side = "Supply",
  consumption_side = "Consumption",
  matname_colname = "matname",
  U_name = "U",
  V_name = "V",
  Y_name = "Y"
)
```

**Arguments**

.DF	a data frame with ledger_side_colname and energy_colname.
ledger_side_colname	the name of the column in .DF that contains ledger side (a string). Default is "Ledger.side".
energy_colname	the name of the column in .DF that contains energy values (a string). Default is "E.ktoe".
supply_side	the identifier for items on the supply side of the ledger (a string). Default is "Supply".
consumption_side	the identifier for items on the consumption side of the ledger (a string). Default is "Consumption".
matname_colname	the name of the output column containing the name of the matrix in which this row belongs (a string). Default is "UVY".
U_name	the name for the use matrix (a string). Default is "U".
V_name	the name for the make matrix (a string). Default is "V".
Y_name	the name for the final demand matrix (a string). Default is "Y".

**Value**

.DF with an added column, UVY\_colname.

**Examples**

```
matsindf:::add_UKEnergy2000_matnames(UKEnergy2000)
```

---

```
add_UKEnergy2000_row_col_meta
```

*Add row, column, row type, and column type metadata*

---

**Description**

Add row, column, row type, and column type metadata

**Usage**

```
add_UKEnergy2000_row_col_meta(
  .DF,
  matname_colname = "matname",
  U_name = "U",
  V_name = "V",
  Y_name = "Y",
  product_colname = "Product",
  flow_colname = "Flow",
```

```

industry_type = "Industry",
product_type = "Product",
sector_type = "Sector",
rowname_colname = "rowname",
colname_colname = "colname",
rowtype_colname = "rowtype",
coltype_colname = "coltype"
)

```

### Arguments

.DF	a data frame containing matname_colname.
matname_colname	the name of the column in .DF that contains names of matrices (a string). Default is "matname".
U_name	the name for use matrices (a string). Default is "U".
V_name	the name for make matrices (a string). Default is "V".
Y_name	the name for final demand matrices (a string). Default is "Y".
product_colname	the name of the column in .DF where Product names is found (a string). Default is "Product".
flow_colname	the name of the column in .DF where Flow information is found (a string). The Flow column usually contains the industries involved in this flow. Default is "Flow".
industry_type	the name that identifies production industries and and transformation processes (a string). Default is "Industry".
product_type	the name that identifies energy carriers (a string). Default is "Product".
sector_type	the name that identifies final demand sectors (a string). Default is "Sector".
rowname_colname	the name of the output column that contains row names for matrices (a string). Default is "rowname".
colname_colname	the name of the output column that contains column names for matrices (a string). Default is "colname".
rowtype_colname	the name of the output column that contains row types for matrices (a string). Default is "rowtype".
coltype_colname	the name of the output column that contains column types for matrices (a string). Default is "coltype".

### Value

.DF with additional columns named rowname\_colname, colname\_colname, rowtype\_colname, and coltype\_colname.

**Examples**

```
UKEnergy2000 %>%
  matsindf:::add_UKEnergy2000_matnames(.) %>%
  matsindf:::add_UKEnergy2000_row_col_meta(.)
```

---

build_keep_args	<i>Build a list of arguments to keep</i>
-----------------	--

---

**Description**

In the process of building data frames of arguments to FUN, we need to decide which arguments to keep from each source, . . . , .dat, and defaults to FUN. This function does that work in one place.

**Usage**

```
build_keep_args(where_to_find_args)
```

**Arguments**

```
where_to_find_args
      A list created by where_to_get_args().
```

**Value**

A list with names .dat, dots, and FUN which gives items to keep from each source.

---

build_matsindf_apply_data_frame	<i>Create a data frame consisting of the input data for matsindf_apply()</i>
---------------------------------	--

---

**Description**

This is an internal helper function that takes the types list and creates a data frame from which calculations can proceed.

**Usage**

```
build_matsindf_apply_data_frame(
  .dat = NULL,
  FUN,
  ...,
  types = matsindf_apply_types(.dat, FUN = FUN, ... = ...)
)
```

**Arguments**

<code>.dat</code>	The value of the <code>.dat</code> argument to <code>matsindf_apply()</code> , as a list or a data frame.
<code>FUN</code>	The function supplied to <code>matsindf_apply()</code> .
<code>...</code>	The <code>...</code> argument supplied to <code>matsindf_apply()</code> .
<code>types</code>	The types for <code>matsindf_apply()</code> . Supply if already calculated externally. Default is <code>types = matsindf_apply_types(.dat, FUN = FUN, ... = ...)</code> .

**Details**

This function enforces the precedence rules for `matsindf_apply()`, namely that variables found in `...` take priority over variables found in `.dat`, which take priority over variables found in the default values of `FUN`.

**Value**

A data frame (actually, a tibble) with columns from `dots`, `.dat`, and the default values to `FUN`, according to precedence rules for `matsindf_apply()`.

---

`collapse_to_matrices` *Collapse a "tidy" data frame to matrices in a data frame* `matsindf`)

---

**Description**

A "tidy" data frame contains information that can be collapsed into matrices, including columns for matrix names, row names, column names, row types, column types, and values (entries in matrices). These column names are specified as strings by the `matnames`, `rownames`, `colnames`, `rowtypes`, `coltypes`, and `values` arguments to `collapse_to_matrices()`, respectively. A `matsindf`-style matrix has named rows and columns. In addition, `matsindf`-style matrices have "types" for row and column information, such as "Commodities", "Industries", "Products", or "Machines". The row and column types for the `matsindf`-style matrices are stored as attributes on the matrix (`rowtype` and `coltype`), which can be accessed with the functions `matsbyname::rowtype()` and `matsbyname::coltype()`. Row and column types are both respected and propagated by the various `*_byname` functions of the `matsbyname` package. Use the `*_byname` functions when you do operations on the `matsindf`-style matrices. The `matsindf`-style matrices will be stored in a column with same name as the incoming values column. This function is similar to `tidyr::nest()`, which stores data frames into a cell of a data frame. With `collapse_to_matrices`, matrices are created. This function respects groups, like `dplyr::summarise()`. (In fact, calls to this function may not work properly unless grouping is provided. Errors of the form "Error: Duplicate identifiers for rows ..." are usually fixed by grouping `.DF` prior to calling this function.) The usual approach is to `dplyr::group_by()` the `matnames` column and any other columns to be preserved in the output. Note that execution is halted if any of `rownames`, `colnames`, `rowtypes`, `coltypes`, or `values` is a grouping variable in `.DF`; `rowtypes` and `coltypes` should be the same for all rows of the same matrix in `.DF`; execution is halted if that is not the case. `tidyr::pivot_wider()`ing the output by `matnames` may be necessary before calculations are done on the collapsed matrices. See the example.

**Usage**

```
collapse_to_matrices(
  .DF,
  matnames = "matnames",
  matvals = "matvals",
  rownames = "rownames",
  colnames = "colnames",
  rowtypes = if ("rowtypes" %in% names(.DF)) "rowtypes" else NULL,
  coltypes = if ("coltypes" %in% names(.DF)) "coltypes" else NULL,
  matrix.class = lifecycle::deprecated(),
  matrix_class = c("matrix", "Matrix")
)
```

**Arguments**

.DF	the "tidy" data frame
matnames	A string identifying the column in .DF containing matrix names for matrices to be created. Default is "matnames".
matvals	A string identifying the column in .DF containing values to be inserted into the matrices to be created. This will also be the name of the column in the output containing matrices formed from the data in the matvals column. Default is "matvals".
rownames	A string identifying the column in .DF containing row names for matrices to be created. Default is "rownames".
colnames	A string identifying the column in .DF containing column names for matrices to be created. Default is "colnames".
rowtypes	An optional string identifying the column in .DF containing the type of values in rows of the matrices to be created. Default is if ("rowtypes" %in% names(.DF)) "rowtypes" else NULL, so that failure to set the rowtypes argument will give NULL, as appropriate.
coltypes	An optional string identifying the column in .DF containing the type of values in columns of the matrices to be created. Default is if ("coltypes" %in% names(.DF)) "coltypes" else NULL, so that failure to set the coltypes argument will give NULL, as appropriate.
matrix.class	<b>[Deprecated]</b> Use matrix_class instead.
matrix_class	One of "matrix" or "Matrix". "matrix" creates a base::matrix object with the matrix() function. "Matrix" creates a Matrix::Matrix object using the matsbyname::Matrix() function. This could be a sparse matrix. Default is "matrix".

**Details**

Groups are not preserved on output.

Note that two types of matrices can be created, a matrix or a Matrix. Matrix has the advantage of representing sparse matrices with less memory (and disk space). Matrix objects are created by matsbyname::Matrix().

**Value**

A data frame with matrices in the `matvals` column.

**See Also**

`tidyr::nest()` and `dplyr::summarise()`.

**Examples**

```
library(dplyr)
library(tidyr)
library(tibble)
ptype <- "Products"
itype <- "Industries"
tidy <- data.frame(Country = c( "GH", "GH", "GH", "GH", "GH", "GH", "GH",
                               "US", "US", "US", "US", "GH", "US"),
                  Year    = c( 1971, 1971, 1971, 1971, 1971, 1971, 1971,
                               1980, 1980, 1980, 1980, 1971, 1980),
                  matrix  = c( "U", "U", "E", "E", "E", "V", "V",
                               "U", "U", "E", "E", "eta", "eta"),
                  row     = c( "c 1", "c 2", "c 1", "c 2", "c 2", "i 1", "i 2",
                               "c 1", "c 1", "c 1", "c 2", NA, NA),
                  col     = c( "i 1", "i 2", "i 1", "i 2", "i 3", "c 1", "c 2",
                               "i 1", "i 2", "i 1", "i 2", NA, NA),
                  rowtypes = c( ptype, ptype, ptype, ptype, ptype, itype, itype,
                               ptype, ptype, ptype, NA, NA),
                  coltypes = c( itype, itype, itype, itype, itype, ptype, ptype,
                               itype, itype, itype, NA, NA),
                  vals     = c( 11 , 22, 11 , 22 , 23 , 11 , 22 ,
                               11 , 12 , 11 , 22, 0.2, 0.3)
                ) %>% group_by(Country, Year, matrix)
mats <- collapse_to_matrices(tidy, matnames = "matrix", matvals = "vals",
                             rownames = "row", colnames = "col",
                             rowtypes = "rowtypes", coltypes = "coltypes")
mats %>% pivot_wider(names_from = matrix, values_from = vals)
```

---

df\_to\_msg

*Create a message from a data frame*

---

**Description**

This function is especially helpful for cases when a data frame of missing or unset values is at hand. Trim unneeded columns, then call this function to create a string with rows separated by semicolons and entries separated by commas.

**Usage**

```
df_to_msg(df)
```



**Arguments**

df                    The data frame to be converted to a message

**Value**

A string with rows separated by semicolons and entries separated by commas.

**Examples**

```
data.frame(a = c(1, 2, 3), b = c("a", "b", "c")) |>
  df_to_msg()
```

---

everything\_except            *Get symbols for all columns except ...*

---

**Description**

This convenience function performs a set difference between the columns of `.DF` and the variable names (or symbols) given in `...`

**Usage**

```
everything_except(.DF, ..., .symbols = TRUE)
```

**Arguments**

`.DF`                    A data frame whose variable names are to be differenced.

`...`                    A string, strings, vector of strings, or list of strings representing column names to be subtracted from the names of `.DF`/

`.symbols`                A boolean that defines the return type: TRUE for symbols, FALSE for strings.

**Value**

A vector of symbols (when `.symbols = TRUE`) or strings (when `symbol = FALSE`) containing all variables names except those given in `...`

**Examples**

```
DF <- data.frame(a = c(1, 2), b = c(3, 4), c = c(5, 6))
everything_except(DF, "a", "b")
everything_except(DF, "a", "b", symbols = FALSE)
everything_except(DF, c("a", "b"))
everything_except(DF, list("a", "b"))
```

---

 expand\_to\_tidy

*Expand a matsindf data frame*


---

## Description

Any tidy data frame of matrices (in which each row represents one matrix observation) can also be represented as a tidy data frame with each non-zero matrix entry as an observation on its own row. This function (and `collapse_to_matrices()`) convert between the two representations.

## Usage

```
expand_to_tidy(
  .DF,
  matnames = "matnames",
  matvals = "matvals",
  rownames = "rownames",
  colnames = "colnames",
  rowtypes = "rowtypes",
  coltypes = "coltypes",
  drop = NA
)
```

## Arguments

<code>.DF</code>	The data frame containing <b>matsindf</b> -style matrices. ( <code>.DF</code> may also be a named list of matrices, in which case names of the matrices are taken from the names of items in the list and list items are expected to be matrices.)
<code>matnames</code>	The name of the column in <code>.DF</code> containing matrix names (a string). Default is "matnames".
<code>matvals</code>	The name of the column in <code>.DF</code> containing IO-style matrices or constants (a string), This will also be the name of the column containing matrix entries in the output data frame. Default is "matvals".
<code>rownames</code>	The name for the output column of row names (a string). Default is "rownames".
<code>colnames</code>	The name for the output column of column names (a string). Default is "colnames".
<code>rowtypes</code>	An optional name for the output column of row types (a string). Default is "rowtypes".
<code>coltypes</code>	The optional name for the output column of column types (a string) Default is "coltypes".
<code>drop</code>	If specified, the value to be dropped from output, For example, <code>drop = 0</code> will cause <code>0</code> entries in the matrices to be deleted from output. If <code>NA</code> , no values are dropped from output. Default is <code>NA</code> .

## Details

Names for output columns are specified in the `rownames`, `colnames`, `rowtypes`, and `coltypes`, arguments. The entries of the **matsindf**-style matrices are stored in an output column named `vals`.

## Value

A tidy data frame containing expanded **matsindf**-style matrices

## Examples

```
library(dplyr)
library(matsbyname)
ptype <- "Products"
itype <- "Industries"
tidy <- data.frame(Country = c( "GH", "GH", "GH", "GH", "GH", "GH", "GH",
                               "US", "US", "US", "US", "GH", "US"),
                  Year     = c( 1971, 1971, 1971, 1971, 1971, 1971, 1971,
                               1980, 1980, 1980, 1980, 1971, 1980),
                  matrix   = c( "U", "U", "Y", "Y", "Y", "V", "V",
                               "U", "U", "Y", "Y", "eta", "eta"),
                  row      = c( "c1", "c2", "c1", "c2", "c2", "i1", "i2",
                               "c1", "c1", "c1", "c2", NA, NA),
                  col      = c( "i1", "i2", "i1", "i2", "i3", "c1", "c2",
                               "i1", "i2", "i1", "i2", NA, NA),
                  rowtypes = c( ptype, ptype, ptype, ptype, ptype, itype, itype,
                               ptype, ptype, ptype, ptype, NA, NA),
                  coltypes = c( itype, itype, itype, itype, itype, ptype, ptype,
                               itype, itype, itype, itype, NA, NA),
                  vals     = c(11 , 22, 11 , 22 , 23 , 11 , 22 ,
                               11 , 12 , 11 , 22, 0.2, 0.3)) %>%
  group_by(Country, Year, matrix)
mats <- collapse_to_matrices(tidy, matnames = "matrix", rownames = "row", colnames = "col",
                             rowtypes = "rowtypes", coltypes = "coltypes",
                             matvals = "vals") %>%
  ungroup()
expand_to_tidy(mats, matnames = "matrix", matvals = "vals",
               rownames = "rows", colnames = "cols",
               rowtypes = "rt", coltypes = "ct")
expand_to_tidy(mats, matnames = "matrix", matvals = "vals",
               rownames = "rows", colnames = "cols",
               rowtypes = "rt", coltypes = "ct", drop = 0)
```

**Description**

`formals(FUN)` does not handle arguments without a default well, returning a name vector of length 1, which when converted to character is `""`. This function detects that condition and replaces the no-default argument with the value of `.no_default`, by default `NULL`.

**Usage**

```
get_useable_default_args(FUN, which = c("values", "names"), no_default = NULL)
```

**Arguments**

<code>FUN</code>	A function from which values of default arguments are to be extracted.
<code>which</code>	Tells whether to get "names" of arguments or "values" of arguments. Default is "values".
<code>no_default</code>	The placeholder value for arguments with no default.

**Value**

A named list of default arguments to `FUN`. Names are the argument names. Values are the default argument values.

**Examples**

```
f <- function(a = 42, b) {
  return(a + b)
}
matsindf:::get_useable_default_args(f)
matsindf:::get_useable_default_args(f, no_default = logical())
```

---

`group_by_everything_except`

*Group by all variables except some*

---

**Description**

This is a convenience function that allows grouping of a data frame by all variables (columns) except those variables specified in `...`

**Usage**

```
group_by_everything_except(.DF, ..., .add = FALSE, .drop = FALSE)
```

**Arguments**

<code>.DF</code>	A data frame to be grouped.
<code>...</code>	A string, strings, vector of strings, or list of strings representing column names to be excluded from grouping.
<code>.add</code>	When <code>.add = FALSE</code> , the default, <code>dplyr::group_by()</code> will override existing groups. To add to the existing groups, use <code>.add = TRUE</code> .
<code>.drop</code>	When <code>.drop = TRUE</code> , empty groups are dropped. Default is <code>FALSE</code> .

**Value**

A grouped version of `.DF`.

**Examples**

```
library(dplyr)
DF <- data.frame(a = c(1, 2), b = c(3, 4), c = c(5, 6))
group_by_everything_except(DF) %>% group_vars()
group_by_everything_except(DF, NULL) %>% group_vars()
group_by_everything_except(DF, c()) %>% group_vars()
group_by_everything_except(DF, list()) %>% group_vars()
group_by_everything_except(DF, c) %>% group_vars()
group_by_everything_except(DF, "a") %>% group_vars()
group_by_everything_except(DF, "c") %>% group_vars()
group_by_everything_except(DF, c("a", "c")) %>% group_vars()
group_by_everything_except(DF, c("a")) %>% group_vars()
group_by_everything_except(DF, list("a")) %>% group_vars()
```

---

handle_empty_data	<i>Gracefully handle empty data</i>
-------------------	-------------------------------------

---

**Description**

When empty data are provided to `matsindf_apply()`, care must be taken with the return value. This function assembles the correct zero-row data frame or zero-length lists.

**Usage**

```
handle_empty_data(.dat = NULL, FUN, DF, types)
```

**Arguments**

<code>.dat</code>	The <code>.dat</code> argument to <code>matsindf_apply()</code> .
<code>FUN</code>	The <code>FUN</code> argument to <code>matsindf_apply()</code> .
<code>DF</code>	The assembled <code>DF</code> inside <code>matsindf_apply()</code> .
<code>types</code>	The <code>types</code> object assembled inside <code>matsindf_apply()</code> .

**Value**

The appropriate return value from `matsindf_apply()`, either a zero-length list or a zero-row data frame.

---

<code>handle_null_args</code>	<i>Gracefully handle NULL arguments</i>
-------------------------------	---

---

**Description**

When NULL is passed as an element of the `.dat` or `...` arguments to `matsindf_apply()`, special care must be taken. This function helps in those situations.

**Usage**

```
handle_null_args(.arg)
```

**Arguments**

`.arg` One of `.dat` or `...` (as a list) arguments to `matsindf_apply()`.

**Value**

A list representation of `.arg` with NULL values handled appropriately.

---

<code>index_column</code>	<i>Index a column in a data frame by groups relative to an initial year</i>
---------------------------	---

---

**Description**

This function indexes (by ratio) variables in `vars_to_index` to the first time in `time_var` or to `index_time` (if specified). Groups in `.DF` are both respected and required. Neither `var_to_index` nor `time_var` can be in the grouping variables.

**Usage**

```
index_column(
  .DF,
  var_to_index,
  time_var = "Year",
  index_time = NULL,
  indexed_var = paste0(var_to_index, suffix),
  suffix = "_indexed"
)
```

**Arguments**

.DF	the data frame in which the variables are contained
var_to_index	the column name representing the variable to be indexed (a string)
time_var	the name of the column containing time information. Default is "Year".
index_time	the time to which data in var_to_index are indexed. If NULL (the default), index_time is set to the first time of each group.
indexed_var	the name of the indexed variable. Default is "<<var_to_index>>_<<suffix>>".
suffix	the suffix to be appended to the indexed variable. Default is "_indexed".

**Details**

Note that this function works when the variable to index is a column of numbers or a column of matrices.

**Value**

a data frame with same number of rows as .DF and the following columns: grouping variables of .DF, var\_to\_index, time\_var, and one additional column containing indexed var\_to\_index named with the value of indexed\_var.

**Examples**

```
library(dplyr)
library(tidyr)
DF <- data.frame(Year = c(2000, 2005, 2010), a = c(10, 15, 20), b = c(5, 5.5, 6)) %>%
  gather(key = name, value = var, a, b) %>%
  group_by(name)
index_column(DF, var_to_index = "var", time_var = "Year", suffix = "_ratioed")
index_column(DF, var_to_index = "var", time_var = "Year", indexed_var = "now.indexed")
index_column(DF, var_to_index = "var", time_var = "Year", index_time = 2005,
             indexed_var = "now.indexed")
## Not run:
DF %>%
  ungroup() %>%
  group_by(name, var) %>%
  index_column(var_to_index = "var", time_var = "Year") # Fails! Do not group on var_to_index.
DF %>%
  ungroup() %>%
  group_by(name, Year) %>%
  index_column(var_to_index = "var", time_var = "Year") # Fails! Do not group on time_var.
## End(Not run)
```

matrix\_cols

*Find columns that contain matrices***Description**

It is often helpful to find the columns of a `matsindf` data frame that contain exclusively or some matrices. This function helps with that task.

**Usage**

```
matrix_cols(df, .drop_names = FALSE, .any = FALSE)
```

**Arguments**

<code>.df</code>	The data frame to be queried for matrix columns.
<code>.drop_names</code>	A boolean that tells whether to remove the names from the returned integer vector. Default is <code>FALSE</code> .
<code>.any</code>	A boolean that tells whether a column is reported when <code>any()</code> of the rows contain matrices (instead of <code>all()</code> rows contain matrices). Default is <code>FALSE</code> , in which case all entries in a column must be a matrix for the column to be reported.

**Details**

By default, a column is considered a matrix column if `all()` of the rows contain matrices. Use the `.test_any` argument to modify this behavior.

By default, the vector of integers returned from this function is named by the columns. Use the `.drop_names` function to modify this behavior.

**Value**

A vector of integers saying which columns contain matrices.

**Examples**

```
tidy <- tibble::tibble(matrix = c("V1", "V1", "V1", "V2", "V2"),
  row = c("i1", "i1", "i2", "i1", "i2"),
  col = c("p1", "p2", "p2", "p1", "p2"),
  vals = c(1, 2, 3, 4, 5)) %>%
  dplyr::mutate(
    rowtypes = "Industries",
    coltypes = "Products"
  ) %>%
  dplyr::group_by(matrix)
matsdf <- tidy %>%
  collapse_to_matrices(matnames = "matrix", matvals = "vals",
    rownames = "row", colnames = "col",
    rowtypes = "rowtypes", coltypes = "coltypes")
```



```

matsdf
matrix_cols(matsdf)
matrix_cols(matsdf, .drop_names = TRUE)

```

---

matsindf\_apply      *Apply a function to a matsindf data frame (and more)*

---

## Description

Applies FUN to .dat or performs the calculation specified by FUN on numbers or matrices. FUN must return a named list. The values of the list returned FUN become entries in columns in a returned data frame or entries in the sub-lists of a returned list. The names of the items in the list returned by FUN become names of the columns in a returned data frame or names of the list items in the returned list.

## Usage

```
matsindf_apply(.dat = NULL, FUN, ..., .warn_missing_FUN_args = TRUE)
```

## Arguments

.dat	A list of named items or a data frame.
FUN	The function to be applied to .dat.
...	Named arguments to be passed by name to FUN.
.warn_missing_FUN_args	A boolean that tells whether to warn of missing arguments to FUN. Default is TRUE.

## Details

If `is.null(.dat)` and `...` are all named numbers or matrices of the form `argname = m`, `ms` are passed to FUN by `argnames`. The return value is a named list provided by FUN. The arguments in `...` are not included in the output.

If `is.null(.dat)` and `...` are all lists of numbers or matrices of the form `argname = l`, FUN is Mapped across the various `ls` to obtain a list of named lists returned from FUN. The return value is a list whose top-level names are the names of the returned items from FUN. `.dat` is not included in the return value.

If `!is.null(.dat)` and `...` are all named, `length == 1` character strings of the form `argname = string`, `argnames` are expected to be names of arguments to FUN, and `strings` are expected to be column names in `.dat`. The return value is `.dat` with additional columns (at right) whose names are the names of list items returned from FUN. When `.dat` contains columns whose names are same as columns added at the right, a warning is emitted.

`.dat` can be a list of named items in which case a list will be returned instead of a data frame.

If items in `.dat` have same names as arguments to FUN, it is not necessary to specify any arguments in `...`. `matsindf_apply` assumes that the appropriately-named items in `.dat` are intended to be arguments to FUN. When an item name appears in both `...` and `.dat`, `...` takes precedence.

if `.dat` is a data frame, the items in its columns (possibly matrices) are `unname()`d before calling FUN.

NULL arguments in `...` are ignored for the purposes of deciding whether all arguments are numbers, matrices, lists of numbers or matrices, or named character strings. However, all NULL arguments are passed to FUN, so FUN should be able to deal with NULL arguments appropriately.

If `.dat` is present, `...` contains `length == 1` strings, and one of the `...` strings is not the name of a column in `.dat`, FUN is called WITHOUT the argument whose column is missing. I.e., that argument is treated as missing. If FUN works despite the missing argument, execution proceeds. If FUN cannot handle the missing argument, an error will occur in FUN.

It is suggested that FUN is able to handle empty data gracefully, returning an empty result with the same names as when non-empty data are fed to FUN. Attempts are made to handle zero-row data (in `.dat` or `...`) gracefully. First, FUN is called with the empty (but named) data. If FUN can handle empty data without error, the result is returned. If FUN errors when fed empty data, FUN is called with an empty argument list in the hopes that FUN has reasonable default values. If that fails, `.dat` is returned unmodified (if not NULL) or the data in `...` is returned.

## Value

A named list or a data frame. (See details.)

## Examples

```
library(matsbyname)
example_fun <- function(a, b){
  return(list(c = sum_byname(a, b),
             d = difference_byname(a, b)))
}
# Single values for arguments
matsindf_apply(FUN = example_fun, a = 2, b = 2)
# Matrices for arguments
a <- 2 * matrix(c(1,2,3,4), nrow = 2, ncol = 2, byrow = TRUE,
               dimnames = list(c("r1", "r2"), c("c1", "c2")))
b <- 0.5 * a
matsindf_apply(FUN = example_fun, a = a, b = b)
# Single values in lists are treated like columns of a data frame
matsindf_apply(FUN = example_fun, a = list(2, 2), b = list(1, 2))
# Matrices in lists are treated like columns of a data frame
matsindf_apply(FUN = example_fun, a = list(a, a), b = list(b, b))
# Single numbers in a data frame
DF <- data.frame(a = c(4, 4, 5), b = c(4, 4, 4))
matsindf_apply(DF, FUN = example_fun, a = "a", b = "b")
# By default, arguments to FUN come from DF
matsindf_apply(DF, FUN = example_fun)
# Now put some matrices in a data frame.
DF2 <- data.frame(a = I(list(a, a)), b = I(list(b,b)))
matsindf_apply(DF2, FUN = example_fun, a = "a", b = "b")
# All arguments to FUN are supplied by named items in .dat
matsindf_apply(list(a = 1, b = 2), FUN = example_fun)
# All arguments are supplied by named arguments in ..., but mix them up.
# Note that the named arguments override the items in .dat
```

```

matsindf_apply(list(a = 1, b = 2, z = 10), FUN = example_fun, a = "z", b = "b")
# A warning is issued when an output item has same name as an input item.
matsindf_apply(list(a = 1, b = 2, c = 10), FUN = example_fun, a = "c", b = "b")
# When a zero-row data frame supplied to .dat,
# .dat is returned unmodified, unless FUN can handle empty data.
DF3 <- DF2[0, ]
DF3
matsindf_apply(DF3, FUN = example_fun, a = "a", b = "b")

```

---

matsindf\_apply\_types *Determine types of .dat and ... arguments for matsindf\_apply()*

---

## Description

This is a convenience function that returns a list for the types of `.dat` and `...` as well as names in `.dat` and `...`, with components named `.dat_null`, `.dat_df`, `.dat_list`, `.dat_names`, `FUN_arg_all_names`, `FUN_arg_default_names`, `FUN_arg_default_values`, `dots_present`, `all_dots_num`, `all_dots_mats`, `all_dots_list`, `all_dots_vect`, `all_dots_char`, `all_dots_longer_than_1`, `dots_names`, and `keep_args`.

## Usage

```
matsindf_apply_types(.dat = NULL, FUN, ..., .warn_missing_FUN_args = TRUE)
```

## Arguments

<code>.dat</code>	The <code>.dat</code> argument to be checked.
<code>FUN</code>	The function sent to <code>matsindf_apply()</code> .
<code>...</code>	The list of arguments to <code>matsindf_apply()</code> to be checked.
<code>.warn_missing_FUN_args</code>	A boolean that tells whether to warn of missing arguments to <code>FUN</code> . Default is <code>TRUE</code> .

## Details

When `.dat` is a `data.frame`, both `.dat_list` and `.dat_df` are `TRUE`.

When arguments are present in `...`, `dots_present` is `TRUE` but `FALSE` otherwise. When all items in `...` are single numbers, `all_dots_num` is `TRUE` and all other list members are `FALSE`. When all items in `...` are matrices, `all_dots_mats` is `TRUE` and all other list members are `FALSE`. When all items in `...` are lists, `all_dots_list` is `TRUE` and all other list members are `FALSE`. When all items in `...` are vectors (including lists), `all_dots_vect` is `TRUE`. When all items in `...` have length  $> 1$ , `all_dots_longer_than_1` is `TRUE`. When all items in `...` are character strings, `all_dots_char` is `TRUE` and all other list members are `FALSE`.

The various `FUN_arg_*` components give information about the arguments to `FUN`. `FUN_arg_all_names` gives the names of all arguments to `FUN`, regardless of whether they have default values. `FUN_arg_default_names` gives the names of only those arguments with default values. `FUN_arg_default_values` gives the

values of the default arguments, already eval()ed in the global environment. When there are no values in a category, NULL is returned. thus, if FUN has no arguments with default values assigned in the signature of the function, both FUN\_arg\_default\_names and FUN\_arg\_default\_values will be NULL. If FUN has no arguments, all of FUN\_arg\_all\_names, FUN\_arg\_default\_names and FUN\_arg\_default\_values will be NULL.

keep\_args is a named list() of arguments, which indicates which arguments to keep from which source (... , .dat, or default args to FUN) by order of preference, ... over .dat over default arguments to FUN. Arguments not used by FUN are kept, again according to the rules of preference.

## Value

A logical list with components named .dat\_null, .dat\_df, .dat\_list, .dat\_names, FUN\_arg\_all\_names, FUN\_arg\_default\_names, FUN\_arg\_default\_values, dots\_present, all\_dots\_num, all\_dots\_mats, all\_dots\_list, all\_dots\_vect, all\_dots\_char, all\_dots\_longer\_than\_1, dots\_names, and keep\_args.

## Examples

```
identity_fun <- function(a, b) {list(a = a, b = b)}
matsindf_apply_types(.dat = NULL, FUN = identity_fun, a = 1, b = 2)
matsindf_apply_types(.dat = data.frame(), FUN = identity_fun,
  a = matrix(c(1, 2)), b = matrix(c(2, 3)))
matsindf_apply_types(.dat = list(), FUN = identity_fun,
  a = c(1, 2), b = c(3, 4))
matsindf_apply_types(.dat = NULL, FUN = identity_fun,
  a = list(1, 2), b = list(3, 4))
```

---

mat\_to\_rowcolval

*Convert a matrix to a data frame with rows, columns, and values.*

---

## Description

This function "expands" a matrix into a tidy data frame with a values column and factors for row names, column names, row types, and column types. Optionally, values can be dropped.

## Usage

```
mat_to_rowcolval(
  .matrix,
  matvals = "matvals",
  rownames = "rownames",
  colnames = "colnames",
  rowtypes = "rowtypes",
  coltypes = "coltypes",
  drop = NA
)
```

**Arguments**

.matrix	The IO-style matrix to be converted to a data frame with rows, columns, and values.
matvals	A string for the name of the output column containing values. Default is "matvals".
rownames	A string for the name of the output column containing row names. Default is "rownames".
colnames	A string for the name of the output column containing column names. Default is "colnames".
rowtypes	A string for the name of the output column containing row types. Default is "rowtypes".
coltypes	A string for the name of the output column containing column types. Default is "coltypes".
drop	If specified, the value to be dropped from output. Default is NA. For example, drop = 0 will cause 0 entries in the matrices to be deleted from output. If NA, no values are dropped from output.

**Value**

A data frame with rows, columns, and values.

**Examples**

```
library(matsbyname)
data <- data.frame(Country = c("GH", "GH", "GH"),
                  rows = c("c1", "c1", "c2"),
                  cols = c("i1", "i2", "i2"),
                  rt = c("Commodities", "Commodities", "Commodities"),
                  ct = c("Industries", "Industries", "Industries"),
                  vals = c(11, 12, 22))

data
A <- data %>%
  rowcolval_to_mat(rownames = "rows", colnames = "cols",
                  rowtypes = "rt", coltypes = "ct", matvals = "vals")

A
mat_to_rowcolval(A, rownames = "rows", colnames = "cols",
                rowtypes = "rt", coltypes = "ct", matvals = "vals")
mat_to_rowcolval(A, rownames = "rows", colnames = "cols",
                rowtypes = "rt", coltypes = "ct", matvals = "vals", drop = 0)

# This also works for single values
mat_to_rowcolval(2, matvals = "vals",
                rownames = "rows", colnames = "cols",
                rowtypes = "rt", coltypes = "ct")
mat_to_rowcolval(0, matvals = "vals",
                rownames = "rows", colnames = "cols",
                rowtypes = "rt", coltypes = "ct", drop = 0)
```

---

rowcolval\_to\_mat      *Collapse a tidy data frame into a matrix with named rows and columns*

---

## Description

Columns not specified in one of `rownames`, `colnames`, `rowtype`, `coltype`, or `values` are silently dropped. `rowtypes` and `coltypes` are added as attributes to the resulting matrix (via `matsbyname::setrowtype()` and `matsbyname::setcoltype()`). The resulting matrix is a (under the hood) a data frame. If both `rownames` and `colnames` columns of `.DF` contain `NA`, it is assumed that this is a single value, not a matrix, in which case the value in the `values` column is returned.

## Usage

```
rowcolval_to_mat(
  .DF,
  matvals = "matvals",
  rownames = "rownames",
  colnames = "colnames",
  rowtypes = "rowtypes",
  coltypes = "coltypes",
  fill = 0,
  matrix.class = lifecycle::deprecated(),
  matrix_class = c("matrix", "Matrix")
)
```

## Arguments

<code>.DF</code>	A tidy data frame containing columns for row names, column names, and values.
<code>matvals</code>	The name of the column in <code>.DF</code> containing values with which to fill the matrix (a string). Default is "matvals".
<code>rownames</code>	The name of the column in <code>.DF</code> containing row names (a string). Default is "rownames".
<code>colnames</code>	The name of the column in <code>.DF</code> containing column names (a string). Default is "colnames".
<code>rowtypes</code>	An optional string identifying the types of information found in rows of the matrix to be constructed. Default is "rowtypes".
<code>coltypes</code>	An optional string identifying the types of information found in columns of the matrix to be constructed. Default is "coltypes".
<code>fill</code>	The value for missing entries in the resulting matrix. default is 0.
<code>matrix.class</code>	<b>[Deprecated]</b> Use <code>matrix_class</code> instead.
<code>matrix_class</code>	One of "matrix" or "Matrix". "matrix" creates a <code>base::matrix</code> object with the <code>matrix()</code> function. "Matrix" creates a <code>Matrix::Matrix</code> object using the <code>matsbyname::Matrix()</code> function. This could be a sparse matrix. Default is "matrix".

## Details

Note that two types of matrices can be created, a `matrix` or a `Matrix`. `Matrix` has the advantage of representing sparse matrices with less memory (and disk space). `Matrix` objects are created by `matsbyname::Matrix()`.

## Value

A matrix with named rows and columns and, optionally, row and column types.

## Examples

```
library(matsbyname)
library(dplyr)
data <- data.frame(Country = c("GH", "GH", "GH"),
                  rows = c("c 1", "c 1", "c 2"),
                  cols = c("i 1", "i 2", "i 2"),
                  vals = c( 11 , 12, 22 ))
A <- rowcolval_to_mat(data, rownames = "rows", colnames = "cols", matvals = "vals")
A
rowtype(A) # NULL, because types not set
coltype(A) # NULL, because types not set
B <- rowcolval_to_mat(data, rownames = "rows", colnames = "cols", matvals = "vals",
                    rowtypes = "Commodities", coltypes = "Industries")
B
C <- data %>% bind_cols(data.frame(rt = c("Commodities", "Commodities", "Commodities"),
                                ct = c("Industries", "Industries", "Industries"))) %>%
  rowcolval_to_mat(rownames = "rows", colnames = "cols", matvals = "vals",
                  rowtypes = "rt", coltypes = "ct")
C
# Also works for single values if both the rownames and colnames columns contain NA
data2 <- data.frame(Country = c("GH"), rows = c(NA), cols = c(NA),
                  rowtypes = c(NA), coltypes = c(NA), vals = c(2))
data2 %>% rowcolval_to_mat(rownames = "rows", colnames = "cols", matvals = "vals",
                        rowtypes = "rowtypes", coltypes = "coltypes")
data3 <- data.frame(Country = c("GH"), rows = c(NA), cols = c(NA), vals = c(2))
data3 %>% rowcolval_to_mat(rownames = "rows", colnames = "cols", matvals = "vals")
# Fails when rowtypes or coltypes not all same. In data3, column rt is not all same.
data4 <- data %>% bind_cols(data.frame(rt = c("Commodities", "Industries", "Commodities"),
                                ct = c("Industries", "Industries", "Industries")))
## Not run: rowcolval_to_mat(data4, rownames = "rows", colnames = "cols",
                          matvals = "vals", rowtypes = "rt", coltypes = "ct")
## End(Not run)
```

---

should\_unlist

*Tell whether a column can be unlisted*

---

## Description

When evaluating each row of a data frame in `matsindf_apply()`, the result will be a tibble with list columns. This function tells whether a column can be unlisted. This is internal helper function and should not be called externally.

**Usage**

```
should_unlist(this_col)
```

**Arguments**

`this_col` The column to be checked. Or a `data.frame`, in which case every column is checked.

**Value**

A boolean. TRUE if the column can be unlisted, FALSE otherwise. When `this_col` is a `data.frame`, a named boolean vector, one entry for each column.

---

 UKEnergy2000

---

*Energy consumption in the UK in 2000*


---

**Description**

A dataset containing approximations to some of the energy flows in the UK in the year 2000. These data first appeared as the example in Figures 3, 7, and 11 of M.K. Heun, A. Owen, and P.E. Brockway. 2018. A physical supply-use table framework for energy analysis on the energy conversion chain. Applied Energy, Vol. 226, pp. 1134-1162.

**Usage**

```
UKEnergy2000
```

**Format**

A data frame with 36 rows and 7 variables:

**Country** country, GB (Great Britain, only one country)

**Year** year, 2000 (only one year)

**Ledger.side** Supply or Consumption

**Flow.aggregation.point** tells where each row should be aggregated

**Flow** the Industry or Sector involved in this flow

**Product** the energy product involved in this flow

**E.ktoe** magnitude of the energy flow in ktoe

**Source**

[doi:10.1016/j.apenergy.2018.05.109](https://doi.org/10.1016/j.apenergy.2018.05.109)



---

verify\_cols\_missing    *Verify that column names in a data frame are not already present*

---

### Description

In the Recca package, many functions add columns to an existing data frame. If the incoming data frame already contains columns with the names of new columns to be added, a name collision could occur, deleting the existing column of data. This function provides a way to quickly check whether newcols are already present in .DF.

### Usage

```
verify_cols_missing(.DF, newcols)
```

### Arguments

.DF	the data frame to which newcols are to be added
newcols	a single string, a single name, a vector of strings representing the names of new columns to be added to .DF, or a vector of names of new columns to be added to .DF

### Details

This function terminates execution if a column of .DF will be overwritten by one of the newcols.

### Value

NULL. This function should be called for its side effect of checking the validity of the names of newcols to be added to .DF.

### Examples

```
df <- data.frame(a = c(1,2), b = c(3,4))
verify_cols_missing(df, "d") # Silent. There will be no problem adding column "d".
newcols <- c("c", "d", "a", "b")
## Not run: verify_cols_missing(df, newcols) # Error: a and b are already in df.
```

---

where\_to\_get\_args      *Decide where to get each argument to FUN*

---

## Description

The precedence rules for where to obtain values for the FUN argument to `matsindf_apply()` are codified here. The rules are:

- Precedence order: `...`, `.dat`, defaults arguments to FUN (highest priority to lowest priority).
- If an element of `...` is a character string of length 1, the element of `...` provides a mapping between an item in `.dat` (with same name as the value of the character string of length 1) to an argument of FUN (with the same name as the name of the character string of length 1).
- If the value of the character string of length 1 is not a name in `.dat`, the default arguments to FUN are checked in this order.
  - If the name of a default argument to FUN is the same as the value of the string of length 1 argument in `...`, a mapping occurs.
  - If a mapping is not possible, the default arg to FUN is used directly.

## Usage

```
where_to_get_args(.dat = NULL, FUN, ...)
```

## Arguments

<code>.dat</code>	The <code>.dat</code> argument to <code>matsindf_apply()</code> .
<code>FUN</code>	The FUN argument to <code>matsindf_apply()</code> .
<code>...</code>	The <code>...</code> argument to <code>matsindf_apply()</code> .

## Value

A named list wherein the names are the argument names to FUN. Values are character vectors with 2 elements. The first element is named `source` and provides the argument to `matsindf_apply()` from which the named argument should be found, one of `".dat"`, `"FUN"`, or `"..."`. The second element is named `arg_name` and provides the variable name or argument name in the source that contains the input data for the argument to FUN.

## Examples

```
example_fun <- function(a = 1, b) {
  list(c = a + b, d = a - b)
}
# b is not available anywhere, likely causing an error later
matsindf::where_to_get_args(FUN = example_fun)
# b is now available in ...
matsindf::where_to_get_args(FUN = example_fun, b = 2)
# b is now available in .dat
matsindf::where_to_get_args(list(b = 2), FUN = example_fun)
```

```
# b now comes from ..., because ... takes precedence over .dat
matsindf::where_to_get_args(list(b = 2), FUN = example_fun, b = 3)
# Mapping from c in .dat to b in FUN
matsindf::where_to_get_args(list(c = 2),
                             FUN = example_fun, b = "c")
# Redirect from an arg in ... to a different default to FUN
matsindf::where_to_get_args(FUN = example_fun, b = "a")
# b is found in FUN, not in .dat, because the mapping (b = "a")
# is not available in .dat
matsindf::where_to_get_args(list(b = 2), FUN = example_fun, b = "a")
```

# Index

## \* datasets

- UKEnergy2000, [24](#)
- add\_UKEnergy2000\_matnames, [2](#)
- add\_UKEnergy2000\_row\_col\_meta, [3](#)
- build\_keep\_args, [5](#)
- build\_matsindf\_apply\_data\_frame, [5](#)
- collapse\_to\_matrices, [6](#)
- df\_to\_msg, [8](#)
- everything\_except, [9](#)
- expand\_to\_tidy, [10](#)
- get\_useable\_default\_args, [11](#)
- group\_by\_everything\_except, [12](#)
- handle\_empty\_data, [13](#)
- handle\_null\_args, [14](#)
- index\_column, [14](#)
- mat\_to\_rowcolval, [20](#)
- matrix\_cols, [16](#)
- matsindf\_apply, [17](#)
- matsindf\_apply\_types, [19](#)
- rowcolval\_to\_mat, [22](#)
- should\_unlist, [23](#)
- UKEnergy2000, [24](#)
- verify\_cols\_missing, [25](#)
- where\_to\_get\_args, [26](#)