Package 'collapse'

May 27, 2020

Title Advanced and Fast Data Transformation

Version 1.2.1

Date 2020-05-22

BugReports https://github.com/SebKrantz/collapse/issues

Description A C/C++ based package for advanced data transformation in R that is extremely fast, flexible and parsimonious to code with and programmer friendly. It is well integrated with 'dplyr', 'plm' and 'data.table'.
--- Key Features: ---
(1) Advanced data programming: A full set of fast statistical functions supporting grouped and weighted computations on vectors, matrices and data frames. Fast (ordered) and programmable grouping, factor generation, manipulation of data frames and data object conversions.
(2) Advanced aggregation: Fast and easy multi-data-type, multi-function, weighted, parallelized and fully customized data aggregation.
(3) Advanced transformations: Fast (grouped, weighted) replacing and sweeping out of statistics, scaling / standardizing, centering (i.e. between and within transformations), higher-dimensional centering (i.e. multiple fixed effects transformations), linear prediction and partialling-out.
(4) Advanced time-computations: Fast (sequences of) lags / leads, and (lagged / leaded, iterated, quasi-, log-) differences and growth rates on (unordered) time-series and panel data. Multivariate auto, partial and cross-correlation functions for panel data.
Panel data to (ts-)array conversions.
(5) List processing: (Recursive) list search / identification, extraction / subsetting, data-apply, and generalized row-binding / unlisting in 2D.
(6) Advanced data exploration: Fast (grouped, weighted, panel-decomposed) summary statistics for complex multilevel / panel data.

License GPL (>= 2)

Encoding UTF-8

LazyData true

Depends R (>= 3.5.0)

Imports Rcpp (>= 1.0.1), lfe (>= 2.7)

LinkingTo Rcpp

Suggests dplyr, plm, data.table, ggplot2, scales, vars, knitr, rmarkdown, testthat, microbenchmark

SystemRequirements C++11

VignetteBuilder knitr

NeedsCompilation yes

Author Sebastian Krantz [aut, cre],
Matt Dowle [ctb],
Arun Srinivasan [ctb],
Simen Gaure [ctb],
Dirk Eddelbuettel [ctb],
R Core Team and contributors worldwide [ctb],
Martyn Plummer [cph],
1999-2016 The R Core Team [cph]

Maintainer Sebastian Krantz <sebastian.krantz@graduateinstitute.ch>

Repository CRAN

Date/Publication 2020-05-26 22:10:10 UTC

R topics documented:

`collapse-package` *Advanced and Fast Data Transformation*

Description

collapse is a C/C++ based package for data manipulation in R. It's aims are

- to facilitate complex data transformation and exploration tasks in R
- to help make R code fast, flexible, parsimonious and programmer friendly.

It is compatible with *dplyr*, *data.table* and the *plm* approach to panel-data.

Key Features:

1. *Advanced data programming*: A full set of fast statistical functions supporting grouped and weighted computations on vectors, matrices and data frames. Fast (ordered) and programmable grouping, factor generation, manipulation of data frames and data object conversions.
2. *Advanced aggregation*: Fast and easy multi-data-type, multi-function, weighted, parallelized and fully customized data aggregation.
3. *Advanced transformations*: Fast (grouped, weighted) replacing and sweeping out of statistics, scaling / standardizing, centering (i.e. between and within transformations), higher-dimensional centering (i.e. multiple fixed effects transformations), linear prediction and partialling-out.
4. *Advanced time-computations*: Fast (sequences of) lags / leads, and (lagged / leaded, iterated, quasi-, log-) differences and growth rates on (unordered) time-series and panel data. Multivariate auto, partial and cross-correlation functions for panel data. Panel data to (ts-)array conversions.
5. *List processing*: (Recursive) list search / identification, extraction / subsetting, data-apply, and generalized row-binding / unlisting in 2D.
6. *Advanced data exploration*: Fast (grouped, weighted, panel-decomposed) summary statistics for complex multilevel / panel data.

Getting Started

Please see Collapse Documentation & Overview, or the introductory vignette. A compact but non-exhaustive set of examples is also provided below.

Details

collapse provides an integrated set of functions organized into several topics (see Collapse Overview). Many functions are S3 generic with core methods for vectors, matrices and data.frames. Inputs are quickly passed to compiled C/C++ code, enabling flexible and parsimonious programming in R at extreme speeds.

Broad areas of use are fast grouped programming and data manipulation to implement complex statistical techniques, and fast data transformation and exploration code (i.e. for `shiny` apps). Applications include fast panel data estimators and techniques, fast weighted programming (i.e. for survey techniques), fast programming with and aggregation of categorical data, fast programming with time-series and panel-series data, and programming with lists of data objects.

The package largely avoids non-standard evaluation and exports core methods for maximum programmability. Smart attribute handling and additional (not-exported) methods ensure compatibility and support for *dplyr*, *data.table* and the *plm* approach to panel-data. *collapse* comes with a built-in hierarchical documentation facilitating the use of the package.

collapse is mainly coded in C++ and built with `Rcpp`, but also uses C functions from *data.table* (grouping, subsetting, row-binding), *lfe* (centering on multiple factors) and *stats* (ACF and PACF).

Author(s)

Maintainer: Sebastian Krantz `<sebastian.krantz@graduateinstitute.ch>`

Other contributors from packages *collapse* utilizes:

- Matt Dowle, Arun Srinivasan and contributors worldwide (data.table)

- Simen Gaure (lfe)
- Dirk Eddelbuettel and contributors worldwide (Rcpp)
- R Core Team and contributors worldwide (stats)

I also thank Ralf Stubner, Joseph Wood and Dirk Eddelbuettel for helpful answers on Stackoverflow, and Joris Meys on R-Devel for encouraging me and helping to set up the github repository for *collapse*.

Developing / Feature Requests / Bug Reporting

- If you are interested in extending or optimizing this package, see the source code at https://github.com/SebKrantz/collapse/tree/master, fork and send pull-requests, or e-mail me.
- Please send feature requests via e-mail.
- Please report issues at https://github.com/SebKrantz/collapse/issues or e-mail me.

Examples

```
# World Bank World Development Data: 216 countries, 59 years, 4 series (columns 9-12)
head(wlddev)

# Describe data
descr(wlddev)

# Pairwise correlations with p-value
pwcor(num_vars(wlddev), P = TRUE)

# Panel-summarize columns 9 though 12 of this data (within and between countries)
qsu(wlddev, pid = ~ country, cols = 9:12, vlabels = TRUE)

# Do all of that by region and also compute higher moments -> returns a 4D array
qsu(wlddev, ~ region, ~ country, cols = 9:12, higher = TRUE)

# Return as nested list of statistics-matrices instead
suml <- qsu(wlddev, ~ region, ~ country,
            cols = 9:12, higher = TRUE, array = FALSE)
str(suml)

# Create data.frame from this list with 3 identifier columns
head(unlist2d(suml, idcols = c("Variable","Trans"), row.names = "Region"))

# Select columns from wlddev
series <- get_vars(wlddev, 9:12)    # same as wlddev[9:12] but 2x faster and works with data.tables
series <- fselect(wlddev, PCGDP:ODA) # Same thing: > 100x faster t. dplyr::select(wlddev, PCGDP:ODA)

# Replace columns, 8x faster than wlddev[9:12] <- series and also replaces names
get_vars(wlddev, 9:12) <- series

# Fast subsetting
head(fsubset(wlddev, country == "Ireland", -country, -iso3c))
head(fsubset(wlddev, country == "Ireland" & year > 1990, year, PCGDP:ODA))
```

```
ss(wlddev, 1:10, 1:10) # This is an order of magnitude faster than wlddev[1:10, 1:10]

# Fast transforming
head(ftransform(wlddev, ODA_GDP = ODA / PCGDP, ODA_LIFEEX = sqrt(ODA) / LIFEEX))
head(ftransform(wlddev, ODA_GDP = ODA / PCGDP, PCGDP = NULL, ODA = NULL, GINI_sum = fsum(GINI)))

# Calculating fast colum-wise statistics
fNobs(series)                   # Number of non-missing values
fmean(series)                   # means of series
fmedian(series)                 # medians of series
fmin(series)                    # mins of series

# Fast grouped statistics
fNobs(series, wlddev$region)    # regional number of obs
fmean(series, wlddev$region)    # regional means
fmedian(series, wlddev$region)  # regional medians
fsd(series, wlddev$region)      # regional standard-deviations

# Means by region and income
fmean(series, fselect(wlddev, region, income))

# Same using GRP objects:
g <- GRP(wlddev, ~ region + income)
print(g)
plot(g)

# GRP objects are extremely efficient inputs to fast functions
fmean(series, g)
fmedian(series, g)
fsd(series, g)

# Another option is creating a grouped_df, using dplyr::group_by or the faster fgroup_by
gseries <- fgroup_by(fselect(wlddev, region, income, PCGDP:ODA), region, income)
str(gseries)
fmean(gseries)          # grouped mean
fmean(gseries, w = ODA) # weighted grouped mean, weighted by ODA
fsd(gseries, w = ODA)   # Weighted group standard deviation

# Faster aggregations with dplyr:
library(dplyr) # This is already a lot faster than summarize_all(mean)
wlddev %>% group_by(region,income) %>% select(PCGDP,LIFEEX) %>% fmean
# Now this is getting fast, apart from the pipe which still slows things down...
wlddev %>% fgroup_by(region,income) %>% fselect(PCGDP,LIFEEX) %>% fmean

# Data-Apply to columns
head(dapply(series, log))
dapply(series, quantile, na.rm = TRUE)

# Data-Apply to rows (for sum use rowSums(qM(series), na.rm = TRUE), same for rowMeans ...)
head(dapply(mtcars, max, MARGIN = 1, na.rm = TRUE))
head(dapply(mtcars, quantile, MARGIN = 1))

# qM -> quickly convert data to matrix, qDF/qDT do the reverse
```

```
fmean(rowSums(qM(series), na.rm = TRUE))

# Split-apply combine computing on columns
BY(series, wlddev$region, sum, na.rm = TRUE)  # Please use: fsum(series, wlddev$region) -> faster
BY(series, wlddev$region, quantile, na.rm = TRUE)
BY(series, wlddev$region, quantile, na.rm = TRUE, expand.wide = TRUE)

# Convert panel-data to array
psar <- psmat(wlddev, ~country, ~year, cols = 9:12)
str(psar)
psar["Ireland",,]                                    # Fast data access
psar["Ireland",,"PCGDP"]
psar[,"2016",]
qDF(psar[,"2016",], row.names.col = "Country")          # Convert to data.frame
plot(psar, colour = TRUE, labs = vlabels(wlddev)[9:12]) # Visualize
plot(psar[c("Brazil","India","South Africa","Russian Federation","China"),,
          c("PCGDP","LIFEEX","ODA")], legend = TRUE, labs = vlabels(wlddev)[c(9:10,12)])
plot(ts(psar["Brazil",,], 1960, 2018), main = "Brazil, 1960-2018")

# Aggregate this data by country and decade: Numeric columns with mean, categorical with mode
head(collap(wlddev, ~ country + decade, fmean, fmode))

# Multi-function aggregation of certain columns
head(collap(wlddev, ~ country + decade,
       list(fmean, fmedian, fsd),
       list(ffirst, flast), cols = c(3,9:12)))

# Customized Aggregation: Assign columns to functions
head(collap(wlddev, ~ country + decade,
       custom = list(fmean = 9:10, fsd = 9:12, flast = 3, ffirst = 6:8)))

# Fast functions can also do grouped transformations:
head(fsd(series, g, TRA = "/"))     # Scale series by region and income
head(fsum(series, g, TRA = "%"))    # Percentages by region and income
head(fmean(series, g, TRA = "-"))   # Demean / center by region and income
head(fmedian(series, g, TRA = "-")) # De-median by region and income
gmeds <- fmedian(series, g)         # Same thing in 2 steps
head(TRA(series, gmeds, "-", g))

# Faster transformations with dplyr:
wlddev %>% fgroup_by(region,income) %>% fselect(PCGDP,LIFEEX,ODA) %>%
fwithin(ODA) # Centering using weighted means, weighted by ODA

## But there are also tidy transformation operators for common jobs:
# Centering (within-transforming) the 4 series by country
head(W(wlddev, ~ country, cols = 9:12))

# Same but adding overall mean back after subtracting out group means
head(W(wlddev, ~ country, cols = 9:12, mean = "overall.mean"))

# Partialling out country and year fixed effects from 2 series (qF = quick-factor)
head(HDW(wlddev, PCGDP + LIFEEX ~ qF(country) + qF(year)))
```

```
# Same, adding ODA as continuous regressor
head(HDW(wlddev, PCGDP + LIFEEX ~ qF(country) + qF(year) + ODA))

# Standardizing (scaling and centering) by country
head(STD(wlddev, ~ country, cols = 9:12))

# Computing 1 lead and 3 lags of the 4 series: Panel-computations efficient and exactly identified
head(L(wlddev, -1:3, ~ country, ~year, cols = 9:12))

# Computing the 1- and 10-year first differences of the 4 series
head(D(wlddev, c(1,10), 1, ~ country, ~year, cols = 9:12))
head(D(wlddev, c(1,10), 1:2, ~ country, ~year, cols = 9:12))  # first and second differences
head(D(wlddev, -1:1, 1, ~ country, ~year, cols = 9:12))      # 1-year lagged and leaded FD

# Computing the 1- and 10-year growth rates of the 4 series (also keeping the level series)
head(G(wlddev, c(0,1,10), 1, ~ country, ~year, cols = 9:12))

# Adding exactly identified growth rates using data.table
library(data.table)
setDT(wlddev)[, paste0("G.", names(wlddev)[9:12]) := fgrowth(.SD,1,1,iso3c,year), .SDcols = 9:12]

# Deleting again and doing the dame thing with add_vars
get_vars(wlddev, "G1.", regex = TRUE) <- NULL
add_vars(wlddev) <- fgrowth(gv(wlddev, 9:12), 1, 1, wlddev$iso3c, wlddev$year)
get_vars(wlddev, "G1.", regex = TRUE) <- NULL

# Computing the 1- and 10-year log-differences of GDP per capita and Life-Expectancy
head(G(wlddev, c(0,1,10), 1, PCGDP + LIFEEX ~ country, ~year, logdiff = TRUE))

# Same transformations using plm package:

library(plm)
pwlddev <- pdata.frame(wlddev, index = c("country","year"))
head(W(pwlddev$PCGDP))                      # Country-demeaning
head(W(pwlddev, cols = 9:12))
head(W(pwlddev$PCGDP, effect = 2))          # Time-demeaning
head(W(pwlddev, effect = 2, cols = 9:12))
head(HDW(pwlddev$PCGDP))                    # Country- and time-demeaning
head(HDW(pwlddev, cols = 9:12))
head(STD(pwlddev$PCGDP))                    # Standardizing by country
head(STD(pwlddev, cols = 9:12))
head(L(pwlddev$PCGDP, -1:3))                # Panel-lags
head(L(pwlddev, -1:3, 9:12))
head(G(pwlddev$PCGDP))                      # Panel-Growth rates
head(G(pwlddev, 1, 1, 9:12))
```

A0-collapse-documentation

Collapse Documentation & Overview

Description

The following table fully summarizes the contents of `collapse`. The documentation follows a hierarchical structure: This is the main overview page, linking to topical overview pages and associated function pages (unless functions are documented on the topic page).

Topics and Functions

Topic	*Main Features / Keywords*
Fast Statistical Functions	Fast (grouped and weighted) statistical functions for vector, matrix, data.frame and groupe
Fast (Ordered) Grouping	Fast (ordered or unordered) groupings from vectors, data.frames, lists. 'GRP' objects are
Fast Data Frame Manipulation	Fast and flexible select, subset and transform data, including modifying columns by refere
Quick Data Conversion	Quick conversions: data.frame <> data.table | matrix <> list, data.frame, data.table | array
Advanced Data Aggregation	Fast and easy (weighted and parallelized) aggregation of multi-type data, with (multiple) f
Data Transformations	Efficient row- and column- data-apply and Split-Apply-Combine computing. Fast (groupe
Time-Series and Panel-Series	Fast (sequences of) lags / leads and (lagged / leaded and iterated) differences, quasi-differ
List Processing	(Recursive) list search and identification, search and extract list-elements / list-subsetting,
Summary Statistics	Extremely fast (one-pass, grouped and weighted), summary statistics for cross-sectional a
Recode and Replace Values	Recode multiple values (exact or regex matching) and replace `NaN/Inf/-Inf` and outliers
Small (Helper) Functions	Set and extract variable labels, extract variable classes and C storage types, display variab
Data and Global Macros	Groningen Growth and Development Centre 10-Sector Database, World Bank World Dev
Global Options	Set the action taken by generic functions encountering unknown arguments. The default is

Details

The added top-level documentation infrastructure in *collapse* allows you to effectively navigate the package (as in other commercial software documentations like Mathematica). Calling `?FUN` brings up the documentation page documenting the function as in other R packages, with links

to associated topic pages and closely related functions. You can also call topical documentation pages directly from the console. The links to these pages are contained in the global macro `.COLLAPSE_TOPICS` (i.e. calling `help(.COLLAPSE_TOPICS[1])` brings up this page).

Author(s)

Maintainer: Sebastian Krantz <sebastian.krantz@graduateinstitute.ch>

See Also

collapse-package

`A1-fast-statistical-functions`
Fast (Grouped, Weighted) Statistical Functions for Matrix-Like Objects

---

Description

With `fsum`, `fprod`, `fmean`, `fmedian`, `fmode`, `fvar`, `fsd`, `fmin`, `fmax`, `ffirst`, `flast`, `fNobs` and `fNdistinct`, *collapse* presents a coherent set of extremely fast and flexible statistical functions (S3 generics) to perform column-wise, grouped and weighted computations on atomic vectors, matrices and data.frames, with special support for *dplyr* grouped tibbles and data.table's.

Notes: (1) Panel-decomposed (i.e. between and within) statistics as well as grouped and weighted skewness and kurtosis are implemented in `qsu`. (2) The vector-valued functions and operators `fscale/STD`, `fbetween/B`, `fHDbetween/HDB`, `fwithin/W`, `fHDwithin/HDW`, `flag/L/F`, `fdiff/D/Dlog` and `fgrowth/G` are documented under Data Transformations and Time-Series and Panel-Series. These functions also support `plm::pseries` and `plm::pdata.frame`'s.

Usage

```
## All functions (FUN) follow a common syntax in 4 methods:
FUN(x, ...)

## Default S3 method:
FUN(x, g = NULL, [w = NULL,] TRA = NULL, [na.rm = TRUE,]
    use.g.names = TRUE, ...)

## S3 method for class 'matrix'
FUN(x, g = NULL, [w = NULL,] TRA = NULL, [na.rm = TRUE,]
    use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'data.frame'
FUN(x, g = NULL, [w = NULL,] TRA = NULL, [na.rm = TRUE,]
    use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'grouped_df'
```

```
FUN(x, [w = NULL,] TRA = NULL, [na.rm = TRUE,]
    use.g.names = FALSE, keep.group_vars = TRUE, [keep.w = TRUE,] ...)
```

Arguments

x	a vector, matrix, data.frame or grouped tibble (dplyr::grouped_df).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internal
w	a numeric vector of (non-negative) weights, may contain missing values. Supported by fsum, fprod,
TRA	an integer or quoted operator indicating the transformation to perform: 1 - "replace_fill" | 2 - "replace
na.rm	logical. Skip missing values in x. Defaults to TRUE in all functions and implemented at very little com
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.fra
drop	*matrix and data.frame methods:* Drop dimensions and return an atomic vector if g = NULL and TRA = N
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation. By default groupi
keep.w	*grouped_df method:* Logical. TRUE (default) also aggregates weights and saves them in a column, FAL
...	arguments to be passed to or from other methods, and extra arguments to some functions, i.e. the algo

Details

Please see the documentation of individual functions.

Value

x aggregated. data.frame column-attributes and overall attributes are preserved.

See Also

Collapse Overview, Data Transformations, Time-Series and Panel-Series

Examples

```
## default vector method
mpg <- mtcars$mpg
fsum(mpg)                         # Simple sum
fsum(mpg, TRA = "/")              # Simple transformation: divide all values by the sum
fsum(mpg, mtcars$cyl)             # Grouped sum
fmean(mpg, mtcars$cyl)            # Grouped mean
fmean(mpg, w = mtcars$hp)         # Weighted mean, weighted by hp
fmean(mpg, mtcars$cyl, mtcars$hp) # Grouped mean, weighted by hp
fsum(mpg, mtcars$cyl, TRA = "/")  # Proportions / division by group sums
fmean(mpg, mtcars$cyl, mtcars$hp, # Subtract weighted group means, see also ?fwithin
      TRA = "-")

## data.frame method
fsum(mtcars)
fsum(mtcars, TRA = "%")                 # This computes percentages
fsum(mtcars, mtcars[c(2,8:9)])           # Grouped column sum
g <- GRP(mtcars, ~ cyl + vs + am)        # Here precomputing the groups!
fsum(mtcars, g)                          # Faster !!
fmean(mtcars, g, mtcars$hp)
fmean(mtcars, g, mtcars$hp, "-")         # demeaning by weighted group means...
fmean(fgroup_by(mtcars, cyl, vs, am), hp, "-")  # another way of doing it...


fmode(wlddev, drop = FALSE)           # Compute statistical modes of variables in this data
fmode(wlddev, wlddev$income)              # grouped statistical modes ..

## matrix method
m <- qM(mtcars)
fsum(m)
fsum(m, g) # ...

## method for grouped tibbles - for use with dplyr
library(dplyr)
mtcars %>% group_by(cyl,vs,am) %>% select(mpg,carb) %>% fsum
mtcars %>% fgroup_by(cyl,vs,am) %>% fselect(mpg,carb) %>% fsum # equivalent and faster !!
mtcars %>% fgroup_by(cyl,vs,am) %>% fsum(TRA = "%")
mtcars %>% fgroup_by(cyl,vs,am) %>% fmean(hp)      # weighted grouped mean, save sum of weights
mtcars %>% fgroup_by(cyl,vs,am) %>% fmean(hp, keep.group_vars = FALSE)
```

`A2-fast-grouping` *Fast (Ordered) Grouping*

Description

collapse provides the following functions to efficiently group (and order) data:

- `radixorder`, provides fast radix-ordering (+ grouping information) through direct access to the method `base::order(...,method = "radix")`. The source code for both `radixorder` and `base::order(...,method = "radix")`, comes from `data.table:::forder`. `radixorder`

was modified to optionally return either a vector of group starts, a vector of group sizes, or both as an attribute, and also an attribute providing the size of the largest group and a logical statement on whether the input was already ordered. The function radixorderv exists as a programmers alternative.

- GRP creates *collapse* grouping objects of class 'GRP' based on radixorderv. 'GRP' objects form the central building block for grouped operations and programming in *collapse* and are very efficient inputs to all *collapse* functions supporting grouped operations. A 'GRP' object provides information about (1) the number of groups, (2) which rows belong to which group, (3) the group sizes, (4) the unique groups, (5) the variables used for grouping, (6) whether the grouping and initial inputs were ordered and (7) (optionally) the output from radixorder containing the ordering vector with group starts and maximum group size attributes.
- fgroup_by provides a fast replacement for dplyr::group_by, creating a grouped tibble with a 'GRP' object attached. This grouped tibble can however only be used for grouped operations using *collapse* fast functions. dplyr functions will treat this tibble like an ordinary (non-grouped) one.
- qF, shorthand for 'quick-factor' implements very fast (ordered) factor generation from atomic vectors using either radix ordering method = "radix" or index hashing method = "hash". Factors can also be used for efficient grouped programming with collapse functions, especially if they are generated using qF(x,na.exclude = FALSE) which assigns a level to missing values and attaches a class 'na.included' ensuring that no additional missing value checks are executed by *collapse* functions.
- qG, shorthand for 'quick-group', generates a kind of factor-light without the levels attribute but instead an attribute providing the number of levels. Optionally the levels / groups can be attached, but without converting them to character. Objects have a class 'qG', which is also recognized in the collapse ecosystem.
- finteraction is a fast alternative to base::interaction implemented as a wrapper around as.factor.GRP(GRP(...)). It can ge used to generate a factor from multiple vectors, factors or a list of vectors / factors. Unused factor levels are always dropped.
- groupid is a generalization of data.table::rleid providing a run-length type group-id from atomic vectors. It is generalization as it also supports passing an ordering vector and skipping missing values. For example qF and qG with method = "radix" are essentially implemented using groupid(x,radixorder(x)).
- seqid is a specialized function which creates a group-id from sequences of integer values. For any ordinary panel-dataset groupid(id,order(id,time)) and seqid(time,order(id,time)) provide the same id variable. seqid is especially useful for identifying discontinuities in time-sequences and helps to perform operations such as lags or differences on irregularly spaced time-series and panels.

Table of Functions

Function / S3 Generic	*Methods*	*Description*
radixorder, radixorderv	No methods, for data.frame's and vectors	radix based ordering + g
GRP	default, factor, qG, grouped_df, pseries, pdata.frame	fast (ordered) grouping
fgroup_by	No methods, for data.frame's	fast grouped tibbles
qF	No methods, for vectors	quick factor generation
qG	No methods, for vectors	quick grouping

finteraction	No methods, for data.frame's and vectors	faster interactions
groupid	No methods, for vectors	run-length type group-i
seqid	No methods, for vectors	run-length type integer

See Also

Fast Statistical Functions, Collapse Overview

`A3-data-frame-manipulation`
Fast Data Frame Manipulation

Description

collapse provides the following functions for fast manipulation of (mostly) data.frames.

- `fselect` is a much faster alternative to `dplyr::select` to select columns using expressions involving column names. `get_vars` is a more versatile and programmer friendly function to efficiently select and replace columns by names, indices, logical vectors, regular expressions or using functions to identify columns.
- The functions `num_vars`, `cat_vars`, `char_vars`, `fact_vars`, `logi_vars` and `Date_vars` are convenience functions to efficiently select and replace columns by data type.
- `add_vars` efficiently adds new columns at any position within a data.frame (default at the end). This can be done vie replacement (i.e. `add_vars(data) <-newdata`) or returning the appended data (i.e. `add_vars(data,newdata1,newdata2,...)`). Because of the latter, `add_vars` is also a more efficient alternative to `cbind.data.frame`.
- `fsubset` is a much faster version of `base::subset` for efficiently subset vectors, matrices and data.frames. If the non-standard evaluation offered by `fsubset` is not needed, the function `ss` is a much faster and also more secure alternative to `[.data.frame`.
- `ftransform` is a much faster version of `base::transform`, to modify and delete existing columns or append a data frame with new computed columns. `settransform` does all of that by reference, i.e. it modifies the data frame in the global environment. `fcompute` is similar to `ftransform` but only returns modified and computed columns in a new data frame.

Table of Functions

Function / S3 Generic	*Methods*
`fselect`	No methods, for data.frame's
`get_vars`, `num_vars`, `cat_vars`, `char_vars`, `fact_vars`, `logi_vars`, `Date_vars`	No methods, for data.frame's
`add_vars`	No methods, for data.frame's
`fsubset`	`default, matrix, data.frame`
`ss`	No methods, for data.frame's

ftransform	No methods, for data.frame's
settransform	No methods, for data.frame's
fcompute	No methods, for data.frame's

See Also

Quick Data Conversion, Collapse Overview

A4-quick-conversion *Quick Data Conversion*

Description

Convert common data objects quickly, without method dispatch and extensive checks:

- qDF and qDT convert vectors, matrices, higher-dimensional arrays and suitable lists to data.frame and *data.table* respectively.
- qM converts vectors, higher-dimensional arrays, data.frames and suitable lists to matrix.
- mctl and mrtl column- or row-wise convert a matrix to list, data.frame or *data.table*. They are used internally by qDF and qDT, dapply, BY, etc...
- qF converts atomic vectors to factor (documented on a separate page).
- as.numeric_factor and as.character_factor convert factors, or all factor columns in a list, to numeric or character (by converting the levels).

Usage

```
qDF(X, row.names.col = FALSE)
qDT(X, row.names.col = FALSE)
qM(X)
mctl(X, names = FALSE, return = "list")
mrtl(X, names = FALSE, return = "list")
as.numeric_factor(X, keep.attr = TRUE)
as.character_factor(X, keep.attr = TRUE)
```

Arguments

X	a vector, factor, matrix, higher-dimensional array, data.frame or list. mctl and mrtl only take matrices.
row.names.col	should a column capturing names or row.names be added? i.e. when converting atomic objects to data.frame or data.frame to data.table. Can be logical TRUE, which will add a column "row.names" in front, or can supply a name for the column i.e. "column1".
names	logical. Should the list be named?
return	an integer or string specifying what to return. The options are:

Int.	*String*	*Description*
1	"list"	returns a plain list
2	"data.frame"	returns a data.frame
3	"data.table"	returns a data.table

`keep.attr` logical. `TRUE` keeps all attributes of factor variables apart from the levels and class attributes (such as variable labels etc.).

Value

`qDF` - returns a data.frame
`qDT` - returns a data.table
`qM` - returns a matrix
`mctl`, `mrtl` - return a list, data.frame or data.table
`qF` - returns a factor
`as.numeric_factor` - returns X with factors converted to numeric variables
`as.character_factor` - returns X with factors converted to character variables

See Also

`GRP`, Collapse Overview

Examples

```
mtcarsM <- qM(mtcars)                # Matrix from data.frame
mtcarsDT <- qDT(mtcarsM)             # data.table from matrix columns
mrtl(mtcarsM, TRUE, "data.frame")    # data.frame from matrix rows, etc...
qDF(mtcarsM, "cars")              # Adding a row.names column when converting from matrix
qDT(mtcars, "cars")            # Saving row.names when converting data.frame to data.table

cylF <- qF(mtcars$cyl)               # Factor from atomic vector
cylF

## Factor to numeric conversions
identical(mtcars,  as.numeric_factor(dapply(mtcars, qF)))
```

`A6-data-transformations`

Data Transformations

Description

collapse provides an ensemble of functions to perform common data transformations efficiently and user friendly:

- `dapply` applies functions to **rows or columns** of matrices and data.frame's.

- `BY` is an S3 generic for **Split-Apply-Combine computing** and can perform aggregation as well as grouped transformations. (for aggregation please also see `collap` and Fast Statistical Functions).
- `TRA` is an S3 generic to efficiently perform (groupwise) **replacement and sweeping out of statistics**. Supported operations are:

Integer-id	*String-id*	*Description*
1	"replace_fill"	replace and overwrite missing values
2	"replace"	replace but preserve missing values
3	"-"	subtract
4	"-+"	subtract group-statistics but add group-frequency weighted average of group statistics
5	"/"	divide
6	"%"	compute percentages
7	"+"	add
8	"*"	multiply
9	"%%"	modulus
10	"-%%"	subtract modulus

  All of *collapse*'s Fast Statistical Functions have a built-in `TRA` argument for faster access (i.e. you can compute (groupwise) statistics and use them to transform your data with a single function call).
- `fscale/STD` is an S3 generic to perform (groupwise and / or weighted) **scaling / standardizing** of data and is orders of magnitude faster than `base::scale`.
- `fwithin/W` is an S3 generic to efficiently perform (groupwise and / or weighted) **within-transformations / demeaning / centering** of data. Similarly `fbetween/B` computes (groupwise and / or weighted) **between-transformations / averages**.
- `fHDwithin/HDW`, shorthand for 'higher-dimensional within transform', is an S3 generic to efficiently **center data on multiple groups** and partial-out linear models (possibly involving many levels of fixed effects and interactions). In other words, `fHDwithin/HDW` efficiently computes **residuals** from (potentially complex) linear models. Similarly `fHDbetween/HDB`, shorthand for 'higher-dimensional between transformation', computes the corresponding means or **fitted values**.
- `fFtest` is a fast implementation of the R-Squared based F-test, to test **exclusion restrictions** on linear models potentially involving multiple large factors (fixed effects). It internally utilizes `fHDwithin` to project out factors while counting the degrees of freedom.
- `flag/L/F`, `fdiff/D/Dlog` and `fgrowth/G` are S3 generics to compute sequences of **lags / leads** and suitably lagged and iterated (quasi-, log-) **differences** and **growth rates** on time-series and panel data. More in Time-Series and Panel-Series.
- `STD,W,B,HDW,HDB,L,D,Dlog` and `G` are parsimonious wrappers around the `f-` functions above representing the corresponding transformation 'operators'. They have additional capabilities when applied to data-frames (i.e. variable selection, formula input, auto-renaming and id-variable preservation), and are easier to employ in regression formulas, but are otherwise identical in functionality.

Table of Functions

Function / S3 Generic	*Methods*	*Description*
dapply	No methods, works with matrices and data frames	apply functions to
BY	default, matrix, data.frame, grouped_df	Split-Apply-Comb
TRA	default, matrix, data.frame, grouped_df	replace and sweep
fscale/STD	default, matrix, data.frame, pseries, pdata.frame, grouped_df	scale / standardize
fwithin/W	default, matrix, data.frame, pseries, pdata.frame, grouped_df	demean / center da
fbetween/B	default, matrix, data.frame, pseries, pdata.frame, grouped_df	compute means / a
fHDwithin/HDW	default, matrix, data.frame, pseries, pdata.frame	high-dimensional
fHDbetween/HDB	default, matrix, data.frame, pseries, pdata.frame	high-dimensional a
fFtest	No methods, it's a standalone test to which data needs to be supplied.	fast F-test of exclu
flag/L/F	default, matrix, data.frame, pseries, pdata.frame, grouped_df	(sequences of) lag
fdiff/D/Dlog	default, matrix, data.frame, pseries, pdata.frame, grouped_df	(sequences of lagg
fgrowth/G	default, matrix, data.frame, pseries, pdata.frame, grouped_df	(sequences of lagg

See Also

Collapse Overview, Fast Statistical Functions, collap, Time-Series and Panel-Series

---

A7-time-series-panel-series

Time-Series and Panel-Series

Description

collapse provides the following functions to work with time-dependent data:

- flag, and the lag- and lead- operators L and F are S3 generics to efficiently compute sequences of **lags and leads** on ordered or unordered time-series and panel data.
- fdiff, fgrowth, and the operators D, Dlog and G are S3 generics to efficiently compute sequences of suitably lagged / leaded and iterated **differences, log-differences and growth rates** on ordered or unordered time-series and panel data. fdiff/D/Dlog can also compute **quasi-differences** of the form $x_t - \rho x_{t-1}$ or $log(x_t) - \rho log(x_{t-1})$ for log-differences.
- psmat is an S3 generic to efficiently convert panel-vectors or plm::pseries and data.frame's or plm::pdata.frame's to **panel-series matrices and 3D arrays**, respectively.
- psacf, pspacf and psccf are S3 generics to compute estimates of the **auto-, partial auto- and cross- correlation or covariance functions** for panel-vectors or plm::pseries, and multivariate versions for data.frame's or plm::pdata.frame's.

Table of Functions

S3 Generic	*Methods*	*Description*
flag/L/F	default, matrix, data.frame, pseries, pdata.frame, grouped_df	compute (sequences of) la
fdiff/D/Dlog	default, matrix, data.frame, pseries, pdata.frame, grouped_df	compute (sequences of lag
fgrowth/G	default, matrix, data.frame, pseries, pdata.frame, grouped_df	compute (sequences of lag

psmat	default, pseries, data.frame, pdata.frame	convert panel-data to matr
psacf	default, pseries, data.frame, pdata.frame	compute ACF on panel-da
pspacf	default, pseries, data.frame, pdata.frame	compute PACF on panel-d
psccf	default, pseries, data.frame, pdata.frame	compute CCF on panel-da

See Also

Collapse Overview, Data Transformations

`A8-list-processing` *List Processing*

Description

collapse provides the following set of functions to work with lists of R objects:

- **Search and Identification**
  - `is.regular` checks whether an R object is either atomic or a list. A (nested) list composed of regular objects at each level of the list-tree is unlistable to an atomic vector, checked by `is.unlistable`.
  - `ldepth` determines the level of nesting of the list (i.e. the maximum number of nodes of the list-tree).
  - `has_elem` searches elements in a list using element names, regular expressions applied to element names, or a function applied to the elements, and returns `TRUE` if any matches were found.
- **Subsetting**
  - `atomic_elem` examines the top-level of a list and returns a sublist with the atomic elements. Conversely `list_elem` returns the sublist of elements which are themselves lists or list-like objects.
  - `reg_elem` and `irreg_elem` are recursive versions of the former. `reg_elem` extracts the regular part of the list-tree (leading to atomic elements in the final nodes), while `irreg_elem` extracts the 'irregular' part of the list tree leading to non-atomic elements in the final nodes. (*Tipp*: try calling both on an `lm` object). Naturally for all lists `l`, `is.unlistable(reg_elem(l))` evaluates to `TRUE`...
  - `get_elem` extracts elements from a list using element names, regular expressions applied to element names, a function applied to the elements, or element-indices used to subset the lowest-level sub-lists. by default the result is presented as a simplified list containing all matching elements. With the `keep.tree` option however `get_elem` can also be used to subset lists i.e. maintain the full tree but cut off non-matching branches.
- **Apply Functions**
  - `rapply2d` is a recursive version of `base::lapply` with two key differences to `base::rapply`: (1) Data frames are considered as atomic objects, not as (sub-)lists, and (2) the result is not simplified.

- **Unlisting / Row-Binding**
  - unlist2d efficiently unlists unlistable lists in 2-dimensions and creates a data.frame (or *data.table*) representation of the list (unlike base::unlist which returns an atomic vector). This is done by recursively flattening and row-binding R objects in the list (using data.table::rbindlist) while creating identifier columns for each level of the list-tree and (optionally) saving the row-names of the objects in a separate column. unlist2d can thus also be understood as a recursive generalization of do.call(rbind,l), for lists of vectors, data.frames, arrays or heterogeneous objects.

Table of Functions

Function	*Description*
is.regular	function(x) is.atomic(x) || is.list(x)
is.unlistable	checks if list is unlistable
ldepth	level of nesting / maximum depth of list-tree
has_elem	checks if list contains a certain element
get_elem	subset list / extract certain elements
get_elem	subset list / extract certain elements
reg_elem	subset / extract regular part of list
irreg_elem	subset / extract non-regular part of list
atomic_elem	top-level subset atomic elements
list_elem	top-level subset list/list-like elements
rapply2d	recursively apply functions to lists of data objects
unlist2d	recursively unlist/row-bind lists of data objects in 2D, to data.frame or *data.table*

See Also

Collapse Overview

A9-summary-statistics *Summary Statistics*

Description

collapse provides the following functions to efficiently summarize and examine data:

- qsu, shorthand for quick-summary, is an extremely fast summary command inspired by the (xt)summarize command in the STATA statistical software. It computes a set of 7 statistics (nobs, mean, sd, min, max, skewness and kurtosis) using a numerically stable one-pass method. Statistics can be computed weighted, by groups, and also within-and between entities (for multilevel / panel-data).
- descr computes a concise and detailed description of a data.frame, including frequency tables for categorical variables and various statistics and quantiles for numeric variables. It is inspired by Hmisc::describe, but about 10x faster.

- pwcor, pwcov and pwNobs compute pairwise correlations, covariances and observation counts on matrices and data frame's. Pairwise correlations and covariances can be computed together with observation counts and p-values, and output as 3D array (default) or list of matrices. A major feature of pwcor and pwcov is the print method displaying all of these statistics in a single correlation table.
- varying very efficiently checks for the presence of any variation in data (optionally) within groups (such as panel-identifiers).

Table of Functions

Function / S3 Generic	*Methods*	*Description*
qsu	default, matrix, data.frame, pseries, pdata.frame	fast (grouped, wei
descr	No methods, for data.frame's or lists of vectors	detailed statistical
pwcor	No methods, for matrices or data.frame's	pairwise correlatio
pwcov	No methods, for matrices or data.frame's	pairwise covarianc
pwNobs	No methods, for matrices or data.frame's	pairwise observati
varying	default, matrix, data.frame, pseries, pdata.frame, grouped_df	fast variation chec

See Also

Fast Statistical Functions, Collapse Overview

AA1-recode-replace *Recode and Replace Values in Matrix-Like Objects*

Description

A small suite of functions to efficiently perform common recoding and replacing tasks in matrix-like objects (vectors, matrices, arrays, data.frames, lists of atomic objects):

- recode_num and recode_char can be used to efficiently recode multiple numeric or character values, respectively. The syntax is inspired by dplyr::recode, but the functionality is enhanced in the following respects: (1) they are faster than dplyr::recode, (2) when passed a data.frame/list, all appropriately typed columns will be recoded. (3) They preserve the attributes of the data object and of columns in a data.frame/list, and (4) recode_char also supports regular expression matching using grepl.
- replace_NA efficiently replaces NA/NaN with a value. data.frame's can be multi-typed.
- replace_Inf replaces Inf/-Inf (or optionally NaN/Inf/-Inf) with a value (default is NA). replace_Inf skips non-numeric columns in a data.frame.
- replace_outliers replaces values falling outside a 1- or 2-sided numeric threshold or outside a certain number of column- standard deviations with a value (default is NA). replace_outliers skips non-numeric columns in a data.frame.

Usage

```
recode_num(X, ..., default = NULL, missing = NULL)

recode_char(X, ..., default = NULL, missing = NULL, regex = FALSE)

replace_NA(X, value)

replace_Inf(X, value = NA, replace.nan = FALSE)

replace_outliers(X, limits, value = NA,
                 single.limit = c("SDs", "min", "max", "overall_SDs"))
```

Arguments

X	a vector, matrix, array, data.frame or list of atomic objects.
...	comma-separated recode arguments of the form: value = replacement, `2` = 0, Secondary = "SEC" etc.. recode_char with regex = TRUE also supports regular expressions i.e. `^S|D$` = "STD" etc.
default	optional argument to specify a scalar value to replace non-matched elements with.
missing	optional argument to specify a scalar value to replace missing elements with. *Note* that to increase efficiency this is done before the rest of the recoding i.e. the recoding is performed on data where missing values are filled!
regex	logical. If TRUE, all recode-argument names are (sequentially) passed to grepl as a pattern to search X. All matches are replaced. *Note* that NA's are also matched as strings by grepl.
value	a single (scalar) value to replace matching elements with.
replace.nan	logical. TRUE replaces NaN/Inf/-Inf. FALSE (default) replaces only Inf/-Inf.
limits	either a vector of two-numeric values c(minval,maxval) constituting a two-sided outlier threshold, or a single numeric value constituting either a factor of standard deviations (default), or the minimum or maximum of a one-sided outlier threshold. See also single.limit.
single.limit	a character or integer (argument only applies if length(limits) == 1): • 1 -"SDs" specifies that limits will be interpreted as a (two-sided) threshold in column standard-deviations. The underlying code is equivalent to X[abs(fscale(X)) > limits] <-value but faster. Since fscale is S3 generic with methods for grouped_df, pseries and pdata.frame, the standardizing will be grouped if such objects are passed (i.e. the outlier threshold is then measured in within-group standard deviations). • 2 -"min" specifies that limits will be interpreted as a (one-sided) minimum threshold. The underlying code is equivalent to X[X < limits] <-value. • 3 -"max" specifies that limits will be interpreted as a (one-sided) maximum threshold. The underlying code is equivalent to X[X > limits] <-value. • 4 -"overall_SDs" is equivalent to "SDs" but ignores groups when a grouped_df, pseries or pdata.frame is passed (i.e. standardizing and determination of outliers is by the overall column standard deviation).

Note

These functions are not generic and do not offer support for factors or date(-time) objects. see `dplyr::recode_factor`, *forcats* and other appropriate packages for dealing with these classes.

See Also

Small (Helper) Functions, Collapse Overview

Examples

```
recode_char(c("a","b","c"), a = "b", b = "c")
recode_char(month.name, ber = NA, regex = TRUE)
mtcr <- recode_num(mtcars, `0` = 2, `4` = Inf, `1` = NaN)
replace_Inf(mtcr)
replace_Inf(mtcr, replace.nan = TRUE)
replace_outliers(mtcars, c(2, 100))           # replace all values below 2 and above 100 w. NA
replace_outliers(mtcars, 2, single.limit = "min")  # replace all value smaller than 2 with NA
replace_outliers(mtcars, 100, single.limit = "max") # replace all value larger than 100 with NA
replace_outliers(mtcars, 2)                  # replace all values above or below 2 column-
                                          # standard-deviations from the column-mean w. NA
replace_outliers(fgroup_by(iris, Species), 2)     # Passing a grouped_df, pseries or pdata.frame
                                            # allows to remove outliers according to
                                           # in-group standard-deviation. see ?fscale
```

`AA2-small-helpers` *Small (Helper) Functions*

Description

Convenience functions in the *collapse* package that help to deal with variable names, labels, missing values, matching and object checking etc.. Some functions are performance improved replacements for base R functions.

Usage

```
vlabels(X, attrn = "label")  # Get labels of variables in X, in attr(X[[i]], attrn)
vlabels(X, attrn = "label") <- value      # Set labels of variables in X
vclasses(X)                    # Get classes of variables in X
vtypes(X)              # Get data storage types of variables in X (calling typeof)
namlab(X, class = FALSE,      # Return data.frame of names, labels and classes
          attrn = "label")
add_stub(X, stub, pre = TRUE) # Add a stub (i.e. prefix or postfix) to column names
rm_stub(X, stub, pre = TRUE)  # Remove stub from column names
x %!in% table                 # The opposite of %in%
ckmatch(x, table,         # Check-match: throws an informative error if non-matched
     e = "Unknown columns:")
fnlevels(x)                   # Faster version of nlevels(x) (for factors)
funique(x, ordered = TRUE)    # Faster unique(x) and sort(unique(x)) for vectors
```

```
fnrow(X)                  # Faster nrow for data.frames (not faster for matrices)
fncol(X)                  # Faster ncol for data.frames (not faster for matrices)
fdim(X)                    # Faster dim for data.frames (not faster for matrices)
na_rm(x)                    # Remove missing values from vector and return vector
na_omit(X, cols = NULL,       # Faster na.omit for matrices and data.frames
        na.attr = FALSE)
na_insert(X, prop = 0.1)    # Insert missing values at random in vectors, matrices DF's
all_identical(...)         # Check exact equality of multiple objects or list-elements
all_obj_equal(...)         # Check near equality of multiple objects or list-elements
seq_row(X)                    # Fast integer sequences along rows of X
seq_col(X)                    # Fast integer sequences along columns of X
setRownames(object = nm,      # Set rownames of object and return object
        nm = seq_row(object))
setColnames(object = nm, nm)  # Set colnames of object and return object
setDimnames(object = dn, dn)  # Set dimension names of object and return object
unattrib(object)              # Remove all attributes from object
is.categorical(x)             # The opposite of is.numeric
is.Date(x)             # Check if object is of class "Date", "POSIXlt" or "POSIXct"
```

Arguments

X	a matrix or data.frame (some functions also support vectors and arrays although that is less common).
object	a suitable R object.
x, table	a atomic vector.
attrn	character. Name of attribute to store labels or retrieve labels from.
value	a matching character vector of variable labels.
class	logical. Also show the classes of variables in X in a column?
stub	a single character stub, i.e. "log.", which by default will be pre-applied to all variables or column names in X.
pre	logical. FALSE will post-apply stub.
cols	only removes rows with missing values on these columns. Columns can be selected using column names, indices or a selector function (i.e. is.numeric).
na.attr	logical. TRUE adds an attribute containing the removed cases. For compatibility reasons this is exactly the same format as na.omit i.e. the attribute is called "na.action" and of class "omit".
nm	a suitable vector of row- or column-names.
dn	a suitable list of dimension names.
ordered	logical. TRUE (default) sorts the output, FALSE is slightly faster.
prop	specify the proportion of observations randomly replaced with NA.
e	The error message thrown by ckmatch for non-matched elements.
...	for all_identical / all_obj_equal: either multiple comma-separated objects or a single list of objects in which all elements will be checked for exact or numeric equality by all_identical and all_obj_equal, respectively.

See Also

Collapse Overview

Examples

```
## Variable labels
namlab(wlddev, class = TRUE)
vlabels(wlddev)
vlabels(wlddev) <- vlabels(wlddev)

## Stub-renaming
log_mtc <- add_stub(log(mtcars), "log.")
rm_stub(log_mtc, "log.")
rm(log_mtc)

## Checking exact equality of multiple objects
all_identical(iris, iris, iris, iris)
l <- replicate(100, fmean(num_vars(iris), iris$Species), simplify = FALSE)
all_identical(l)
rm(l)

## Missing values
mtc_na <- na_insert(mtcars, 0.15)    # Set 15% of values missing at random
fNobs(mtc_na)                        # See observation count
na_omit(mtc_na)                      # 12x faster than na.omit(airquality)
na_omit(mtc_na, na.attr = TRUE)      # Adds attribute with removed cases, like na.omit
na_omit(mtc_na, cols = c("vs","am")) # Removes only cases missing vs or am
na_omit(qM(mtc_na))                  # Also works for matrices
na_omit(mtc_na$vs, na.attr = TRUE)   # Also works with vectors
na_rm(mtc_na$vs)                     # For vectors na_rm is faster ...
rm(mtc_na)
```

BY *Split-Apply-Combine Computing*

Description

BY is an S3 generic that efficiently applies functions over vectors or matrix- and data.frame columns by groups, and returns various output formats. A simple parallelism is also available.

Usage

```
BY(X, ...)

## Default S3 method:
BY(X, g, FUN, ..., use.g.names = TRUE, sort = TRUE,
   expand.wide = FALSE, parallel = FALSE, mc.cores = 1L,
   return = c("same","list"))
```

```
## S3 method for class 'matrix'
BY(X, g, FUN, ..., use.g.names = TRUE, sort = TRUE,
   expand.wide = FALSE, parallel = FALSE, mc.cores = 1L,
   return = c("same","matrix","data.frame","list"))

## S3 method for class 'data.frame'
BY(X, g, FUN, ..., use.g.names = TRUE, sort = TRUE,
   expand.wide = FALSE, parallel = FALSE, mc.cores = 1L,
   return = c("same","matrix","data.frame","list"))

## S3 method for class 'grouped_df'
BY(X, FUN, ..., use.g.names = FALSE, keep.group_vars = TRUE,
   expand.wide = FALSE, parallel = FALSE, mc.cores = 1L,
   return = c("same","matrix","data.frame","list"))
```

Arguments

X	a atomic vector, matrix or data frame.
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
FUN	a function, can be scalar- or vector-valued.
...	further arguments to FUN.
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.frame method). No row-names are generated for data.tables and grouped tibbles.
sort	logical. Sort the groups? Internally passed to GRP or qF, and only effective if g is not already a factor or GRP object.
expand.wide	logical. If FUN is a vector-valued function returning a vector of fixed length > 1 (such as the quantile function), expand.wide can be used to return the result in a wider format (instead of stacking the resulting vectors of fixed length above each other in each output column).
parallel	logical. TRUE implements simple parallel execution by internally calling parallel::mclapply instead of base::lapply.
mc.cores	integer. Argument to parallel::mclapply indicating the number of cores to use for parallel execution. Can use parallel::detectCores() to select all available cores. See also ?parallel::mclapply.
return	an integer or string indicating the type of object to return. The default 1 -"same" returns the same object type (i.e. passing a matrix returns a matrix and passing a data frame returns a data frame). 2 -"matrix" always returns the output as matrix, 3 -"data.frame" always returns a data frame and 4 -"list" returns the raw (uncombined) output. *Note*: 4 -"list" works together with expand.wide to return a list of matrices.
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation.

Details

BY is a frugal reimplementation of the Split-Apply-Combine computing paradigm. It is faster than base::tapply, base::by, base::aggregate and plyr, and preserves data attributes just like dapply.

I note at this point that the philosophy of *collapse* is to move beyond this rather slow computing paradigm, which is why the Fast Statistical Functions were implemented. However sometimes tasks need to be performed that involve more complex and customized operations on data, and for these cases BY is a good solution.

BY is built principally as a wrapper around lapply(split(x,g),FUN,...), but strongly optimizes on attribute checking compared to *base* R. For more details examine the code yourself or look at the documentation for dapply which works very similar (the only difference really is the splitting performed in BY).

BY is used internally in collap (*collapse*'s main aggregation command) for functions that are not Fast Statistical Functions.

Value

X where FUN was applied to every column split by g.

See Also

dapply, collap, Fast Statistical Functions, Data Transformations, Collapse Overview

Examples

```
v <- iris$Sepal.Length   # A numeric vector
f <- iris$Species       # A factor. Vectors/lists will internally be converted to factor

## default vector method
BY(v, f, sum)                          # Sum by species
BY(v, f, scale)                        # Scale by species (please use fscale instead)
BY(v, f, scale, use.g.names = FALSE)   # Omitting auto-generated names
BY(v, f, quantile)                     # Species quantiles: by default stacked
BY(v, f, quantile, expand.wide = TRUE) # Wide format

## matrix method
m <- qM(num_vars(iris))
BY(m, f, sum)                          # Also return as matrix
BY(m, f, sum, return = "data.frame")  # Return as data.frame ... also works for computations below
BY(m, f, scale)
BY(m, f, scale, use.g.names = FALSE)
BY(m, f, quantile)
BY(m, f, quantile, expand.wide = TRUE)
BY(m, f, quantile, expand.wide = TRUE, # Return as list of matrices
   return = "list")

## data.frame method
BY(num_vars(iris), f, sum)             # Also returns a data.fram
BY(num_vars(iris), f, sum, return = 2) # Return as matrix ... also works for computations below
BY(num_vars(iris), f, scale)
```

```
BY(num_vars(iris), f, scale, use.g.names = FALSE)
BY(num_vars(iris), f, quantile)
BY(num_vars(iris), f, quantile, expand.wide = TRUE)
BY(num_vars(iris), f, quantile,        # Return as list of matrices
   expand.wide = TRUE, return = "list")

## grouped tibble method
library(dplyr)
giris <- group_by(iris, Species)
giris %>% BY(sum)                      # Compute sum
giris %>% BY(sum, use.g.names = TRUE, # Use row.names and
             keep.group_vars = FALSE) # remove 'Species' and groups attribute
giris %>% BY(sum, return = "matrix")  # Return matrix
giris %>% BY(sum, return = "matrix",  # Matrix with row.names
                    use.g.names = TRUE)
giris %>% BY(log)                      # Take logs
giris %>% BY(log, use.g.names = TRUE, # Use row.names and
             keep.group_vars = FALSE) # remove 'Species' and groups attribute
giris %>% BY(quantile)                 # Compute quantiles (output is stacked)
giris %>% BY(quantile,                 # Much better, also keeps 'Species'
             expand.wide = TRUE)
```

---

collap *Advanced Data Aggregation*

Description

collap is a fast and easy to use multi-purpose data aggregation command.

It performs simple aggregations, multi-type data aggregations applying different functions to numeric and categorical data, weighted aggregations (including weighted multi-type aggregations), aggregations applying multiple functions to each column (which can be performed in parallel), and fully customized aggregations where the user passes a list mapping functions to columns.

collap works with *collapse*'s Fast Statistical Functions, providing extremely fast conventional and weighted aggregation. It also works with other functions but this does not deliver high speeds on large data and does not support weighted aggregations.

Usage

```
# Main function: allows formula and data input to `by` and `w` arguments
collap(X, by, FUN = fmean, catFUN = fmode, cols = NULL, w = NULL, wFUN = fsum,
       custom = NULL, keep.by = TRUE, keep.w = TRUE, keep.col.order = TRUE,
       sort.row = TRUE, parallel = FALSE, mc.cores = 1L,
       return = c("wide","list","long","long_dupl"), give.names = "auto", ...)

# Programmer function: allows column names and indices input to `by` and `w` arguments
collapv(X, by, FUN = fmean, catFUN = fmode, cols = NULL, w = NULL, wFUN = fsum,
        custom = NULL, keep.by = TRUE, keep.w = TRUE, keep.col.order = TRUE,
        sort.row = TRUE, parallel = FALSE, mc.cores = 1L,
```

```
        return = c("wide","list","long","long_dupl"), give.names = "auto", ...)

# Auxiliary function: for grouped tibble ('grouped_df') input + non-standard evaluation
collapg(X, FUN = fmean, catFUN = fmode, cols = NULL, w = NULL, wFUN = fsum, custom = NULL,
     keep.group_vars = TRUE, keep.w = TRUE, keep.col.order = TRUE, sort.row = TRUE,
        parallel = FALSE, mc.cores = 1L,
        return = c("wide","list","long","long_dupl"), give.names = "auto", ...)
```

Arguments

X	a data.frame, or an object coercible to data.frame using qDF.
by	for collap: a one-or two sided formula, i.e. ~ group1 or var1 + var2 ~ group1 + group2, or a atomic vector, list of vectors or GRP object used to group X. For collapv: names or indices of grouping columns, or a logical vector or selector function such as is.categorical selecting grouping columns.
FUN	a function, list of functions (i.e. list(fsum,fmean,fsd) or list(myfun1 = function(x)..,sd = sd)), or a character vector of function names, which are automatically applied only to numeric variables.
catFUN	same as FUN, but applied only to categorical (non-numeric) typed columns (is.categorical).
cols	select columns to aggregate using a function, column names, indices or logical vector. *Note*: cols is ignored if a two-sided formula is passed to by.
w	weights. Can be passed as numeric vector or alternatively as formula i.e. ~ weightvar in collap or column name / index etc. i.e. "weightvar" in collapv. collapg supports non-standard evaluations so weightvar can be indicate without quotes if found in X.
wFUN	same as FUN: Function(s) to aggregate weight variable if keep.w = TRUE. By default the sum of the weights is computed in each group.
custom	a named list specifying a fully customized aggregation task. The names of the list are function names and the content columns to aggregate using this function (same input as cols). For example custom = list(fmean = 1:6,fsd = 7:9,fmode = 10:11) tells collap to aggregate columns 1-6 of X using the mean, columns 7-9 using the standard deviation etc. *Note*: custom lets collap ignore any inputs passed to FUN, catFUN or cols.
keep.by, keep.group_vars	logical. FALSE will omit grouping variables from the output. TRUE keeps the variables, even if passed externally in a list or vector (unlike other *collapse* functions).
keep.w	logical. FALSE will omit weight variable from the output i.e. no aggregation of the weights. TRUE aggregates and adds weights, even if passed externally as a vector (unlike other *collapse* functions).
keep.col.order	logical. Retain original column order post-aggregation.
sort.row	logical. Sort rows by the groups. From *collapse 1.2.0* this only applies to character grouping variables.
parallel	logical. Use parallel::mclapply instead of lapply for multi-function or custom aggregation.

mc.cores	integer. Argument to parallel::mclapply setting the number of cores to use.
return	character. Control the output format when aggregating with multiple functions or performing custom aggregation. "wide" (default) returns a wider data frame with added columns for each additional function. "list" returns a list of data frame's - one for each function. "long" adds a column "Function" and row-binds the results from different functions using data.table::rbindlist. "long.dupl" is a special option for aggregating multi-type data using multiple FUN but only one catFUN or vice-versa. In that case the format is long and data aggregated using only one function is duplicated. See Examples.
give.names	logical. Create unique names of aggregated columns by adding a prefix 'FUN.'. 'auto' will automatically create such prefixes whenever multiple functions are applied to a column or custom is used.
...	additional arguments passed to all functions supplied to FUN, catFUN, wFUN or custom. The behavior of Fast Statistical Functions is regulated by option("collapse_unused_arg_acti and defaults to "warning".

Details

collap automatically checks each function passed to it whether it is a Fast Statistical Function (i.e. whether the function name is contained in .FAST_STAT_FUN). If the function is a fast function, collap only does the grouping and then calls the function to carry out the grouped computations. If the function is not one of .FAST_STAT_FUN, BY is called internally to perform the computation. The resulting computations from each function are put into a list and recombined to produce the desired output format as controlled by the return argument. When multiple functions are used with collap, setting parallel = TRUE and the number of cores with mc.cores will instruct collap to execute these function calls in parallel using parallel::mclapply. If only a single function is used which is not a .FAST_STAT_FUN, the parallel and mc.cores arguments are handed down to BY. See Examples.

Value

X aggregated by groups supplied to the by argument.

Note

Since BY does not check and split additional arguments passed to it, it is presently not possible to create a weighted function in R and apply it to data by groups with collap. Weighted aggregations only work with Fast Statistical Functions supporting weights. User written weighted functions can be applied using the *data.table* package.

collap by default (keep.by = TRUE,keep.w = TRUE) preserves all arguments passed to the by or w arguments, whether passed in a formula or externally. The names of externally passed vectors and lists are intelligently extracted. So it is possible to write collap(iris,iris$Species), and obtain an aggregated data frame with two Species columns, whereas collap(iris,~ Species) only has one Species column. Similarly for weight vectors passed to w. In this regard collap is more sophisticated than other *collapse* functions where preservation of grouping and weight variables is restricted to formula use. For example STD(iris,iris$Species) does not preserve Species in the output, whereas STD(iris,~ Species) does. This collap feature is there simply for convenience, for example sometimes a survey is disaggregated into several datasets, and this now allows easy

pulling of identifiers or weights from other datasets for aggregations. If all information is available in one dataset, just using formulas is highly recommended.

See Also

BY, Fast Statistical Functions, Collapse Overview

Examples

```
## A Simple Introduction -------------------------------------
head(iris)
collap(iris, ~ Species)                                  # Default: FUN = fmean for numeric
collapv(iris, 5)                                                 # Same using collapv
collap(iris, ~ Species, fmedian)                                 # Using the median
collap(iris, ~ Species, fmedian, keep.col.order = FALSE)         # Groups in-front
collap(iris, Sepal.Width + Petal.Width ~ Species, fmedian)       # Only '.Width' columns
collapv(iris, 5, cols = c(2, 4))                                 # Same using collapv
collap(iris, ~ Species, list(fmean, fmedian))                    # Two functions
collap(iris, ~ Species, list(fmean, fmedian), return = "long") # Long format
collapv(iris, 5, custom = list(fmean = 1:2, fmedian = 3:4))    # Custom aggregation
collapv(iris, 5, custom = list(fmean = 1:2, fmedian = 3:4),   # Raw output, no column reordering
        return = "list")
collapv(iris, 5, custom = list(fmean = 1:2, fmedian = 3:4),    # A strange choice...
        return = "long")
collap(iris, ~ Species, w = ~ Sepal.Length)              # Using Sepal.Length as weights, ..
weights <- abs(rnorm(fnrow(iris)))
collap(iris, ~ Species, w = weights)                             # Some random weights..
collap(iris, iris$Species, w = weights)                          # Note this behavior...
collap(iris, iris$Species, w = weights,
       keep.by = FALSE, keep.w = FALSE)
library(dplyr) # Needed for "%>%"
iris %>% fgroup_by(Species) %>% collapg                          # dplyr style, but faster

## Multi-Type Aggregation ------------------------------------
head(wlddev)                                               # World Development Panel Data
head(collap(wlddev, ~ country + decade))                  # Aggregate by country and decade
head(collap(wlddev, ~ country + decade, fmedian, ffirst))       # Different functions
head(collap(wlddev, ~ country + decade, cols = is.numeric))    # Aggregate only numeric columns
head(collap(wlddev, ~ country + decade, cols = 9:12))            # Only the 4 series
head(collap(wlddev, PCGDP + LIFEEX ~ country + decade))      # Only GDP and life-expactancy
head(collap(wlddev, PCGDP + LIFEEX ~ country + decade, fsum))   # Using the sum instead
head(collap(wlddev, PCGDP + LIFEEX ~ country + decade, sum,    # Same using base::sum -> slower!!
            na.rm = TRUE))
head(collap(wlddev, wlddev[c("country","decade")], fsum,      # same, exploring different inputs
            cols = 9:10))
head(collap(wlddev[9:10], wlddev[c("country","decade")], fsum))
head(collapv(wlddev, c("country","decade"), fsum))          # ... names/indices with collapv
head(collapv(wlddev, c(1,5), fsum))

g <- GRP(wlddev, ~ country + decade)                          # Precomputing the grouping
head(collap(wlddev, g, keep.by = FALSE))                     # This is slightly faster now
# Aggregate categorical data using not the mode but the last element
head(collap(wlddev, ~ country + decade, fmean, flast))
```

```
head(collap(wlddev, ~ country + decade, catFUN = flast,      # Aggregate only categorical data
            cols = is.categorical))


## Weighted aggregation ----------------------------------------
weights <- abs(rnorm(fnrow(wlddev)))                           # Random weight vector
head(collap(wlddev, ~ country + decade, w = weights))       # Takes weighted mean for numeric..
# ..and weighted mode for categorical data. The weight vector is aggregated using fsum
wlddev$weights <- weights                                      # Adding to data
head(collap(wlddev, ~ country + decade, w = ~ weights))        # Keeps column order
head(collap(wlddev, ~ country + decade, w = ~ weights,       # Aggregating weights using sum
          wFUN = list(fsum, fmax)))                        # and max (corresponding to mode)
wlddev$weights <- NULL


## Multi-Function Aggregation ----------------------------------
head(collap(wlddev, ~ country + decade, list(fmean, fNobs),     # Saving mean and Nobs
            cols = 9:12))

head(collap(wlddev, ~ country + decade,                      # same using base R -> slower
            list(mean = mean,
                 Nobs = function(x,...) sum(!is.na(x))),
            cols = 9:12, na.rm = TRUE))

head(collap(wlddev, ~ country + decade,                          # list output format
            list(fmean, fNobs), cols = 9:12, return = "list"))

head(collap(wlddev, ~ country + decade,                          # long output format
            list(fmean, fNobs), cols = 9:12, return = "long"))

head(collap(wlddev, ~ country + decade,                 # also aggregating categorical data,
           list(fmean, fNobs), return = "long_dupl"))         # and duplicating it 2 times

head(collap(wlddev, ~ country + decade,                     # now also using 2 functions on
            list(fmean, fNobs), list(fmode, flast),              # categorical data
            keep.col.order = FALSE))

head(collap(wlddev, ~ country + decade,                    # more functions, string input,
            c("fmean","fsum","fNobs","fsd","fvar"),              # parallelized execution
           c("fmode","ffirst","flast","fNdistinct"),          # (choose more than 1 cores,
            parallel = TRUE, mc.cores = 1L,                   # depending on your machine)
            keep.col.order = FALSE))


## Custom Aggregation ------------------------------------------
head(collap(wlddev, ~ country + decade,                          # custom aggregation
            custom = list(fmean = 9:12, fsd = 9:10, fmode = 7:8)))

head(collap(wlddev, ~ country + decade,                          # using column names
            custom = list(fmean = "PCGDP", fsd = c("LIFEEX","GINI"),
                          flast = "date")))

head(collap(wlddev, ~ country + decade,                     # weighted parallelized custom
```

```
            custom = list(fmean = 9:12, fsd = 9:10,             # aggregation
                          fmode = 7:8), w = weights,
            wFUN = list(fsum, fmax),
            parallel = TRUE, mc.cores = 1L))

head(collap(wlddev, ~ country + decade,                         # No column reordering
            custom = list(fmean = 9:12, fsd = 9:10,
                          fmode = 7:8), w = weights,
            wFUN = list(fsum, fmax),
            parallel = TRUE, mc.cores = 1L, keep.col.order = FALSE))


## Piped use ------------------------------------------------
wlddev %>% fgroup_by(country, decade) %>% collapg
wlddev %>% fgroup_by(country, decade) %>% collapg(w = ODA)
wlddev %>% fgroup_by(country, decade) %>% collapg(fmedian, flast)
wlddev %>% fgroup_by(country, decade) %>%
  collapg(custom = list(fmean = 9:12, fmode = 5:7, flast = 3))
```

---

collapse-depreciated *Depreciated* collapse *Functions*

Description

The functions Recode and replace_non_finite available until *collapse* v1.1.0 will be removed in the next update of *collapse*. Since v1.2.0, Recode is replaced by recode_num and recode_char and replace_non_finite is replaced by replace_Inf.

Usage

```
Recode(X, ..., copy = FALSE, reserve.na.nan = TRUE, regex = FALSE)

replace_non_finite(X, value = NA, replace.nan = TRUE)
```

Arguments

X	a vector, matrix or data.frame.
...	comma-separated recode arguments of the form: name = newname, `2` = 0, `NaN` = 0, `NA` = 0, `Inf` = NA, `-Inf` = NA, etc...
value	a single (scalar) value to replace matching elements with. Default is NA.
copy	logical. For reciprocal or sequential replacements of the form a = b, b = c make a copy of X to prevent a being replaced with b and then all b-values being replaced with c again. In general Recode does the replacements one-after the other, starting with the first.
reserve.na.nan	logical. TRUE identifies NA and NaN as special numeric values and does the correct replacement. FALSE will treat NA/NaN as strings, and thus not match numeric NA/NaN. *Note*: This is not an issue for Inf/-Inf, which are matched in both numeric and character variables.

regex	logical. If TRUE, all recode-argument names are (sequentially) passed to grepl as a pattern to search X. All matches are replaced.
replace.nan	logical. TRUE (default) replaces NaN/Inf/-Inf. FALSE replaces only Inf/-Inf.

Note

Recode is not suitable for recoding factors or other classed objects / columns, it simply does X[X == value] <-replacement in a more efficient way. For classed objects, see for example dplyr::recode.

See Also

Recode Replace, Collapse Overview

Examples

```
Recode(c("a","b","c"), a = "b", b = "c")
Recode(c("a","b","c"), a = "b", b = "c", copy = TRUE)
Recode(c("a","b","c"), a = "b", b = "a", copy = TRUE)
Recode(month.name, ber = NA, regex = TRUE)
mtcr <- Recode(mtcars, `0` = 2, `4` = Inf, `1` = NaN)
replace_non_finite(mtcr)
replace_non_finite(mtcr, replace.nan = FALSE)
```

---

collapse-options collapse *Global Options*

Description

currently *collapse* only provides option("collapse_unused_arg_action"), which regulates how generic functions (such as the Fast Statistical Functions) in the package react when an unknown argument is passed to a method. The default action is "warning" which issues a warning. Other options are "error", "message" or "none", whereby the latter enables silent swallowing of such arguments.

---

dapply *Data Apply*

Description

dapply efficiently applies functions to columns or rows of matrices and data frame's and (default) returns an object of the same type and with the same attributes, or converts to the other type. A simple parallelism is also available.

Usage

```
dapply(X, FUN, ..., MARGIN = 2, parallel = FALSE, mc.cores = 1L,
       return = c("same","matrix","data.frame"), drop = TRUE)
```

Arguments

X	a matrix or data frame.
FUN	a function, can be scalar- or vector-valued.
...	further arguments to FUN.
MARGIN	integer. The margin which FUN will be applied over. Default 2 indicates columns while 1 indicates rows. See also Details.
parallel	logical. TRUE implements simple parallel execution by internally calling parallel::mclapply instead of base::lapply.
mc.cores	integer. Argument to parallel::mclapply indicating the number of cores to use for parallel execution. Can use parallel::detectCores() to select all available cores. See also ?parallel::mclapply.
return	an integer or string indicating the type of object to return. The default 1 -"same" returns the same object type (i.e. passing a matrix returns a matrix and passing a data frame returns a data frame). 2 -"matrix" always returns the output as matrix and 3 -"data.frame" always returns a data frame.
drop	logical. If the result has only one row or one column, drop = TRUE will drop dimensions and return a (named) atomic vector.

Details

dapply is an efficient command to apply functions to rows or columns of data without loosing information (attributes) about the data or changing the classes or format of the data. It is principally an efficient wrapper around base::lapply and works as follows:

- Save the attributes of X.
- If MARGIN = 2 (columns), convert matrices to plain lists of columns using mctl and remove all attributes from data frames.
- If MARGIN = 1 (rows), convert matrices to plain lists of rows using mrtl. For data frames remove all attributes, efficiently convert to matrix using do.call(rbind,X) and also convert to list of rows using mrtl.
- Call base::lapply or parallel::mclapply on these plain lists (which is faster than calling lapply on an object with attributes).
- depending on the requested output type, use base::matrix, base::unlist or do.call(cbind,...) to convert the result back to a matrix or list of columns.
- modify the relevant attributes accordingly and efficiently attach to the object again (no further checks).

This performance gain from working with plain lists makes dapply not much slower than calling lapply itself on a data frame. Because of the conversions involved, row-operations require some memory, but are still faster than base::apply.

Value

X where FUN was applied to every row or column.

See Also

BY, collap, Fast Statistical Functions, Data Transformations, Collapse Overview

Examples

```
dapply(mtcars, log)                       # Take natural log of each variable
dapply(mtcars, log, return = "matrix")    # Return as matrix
m <- as.matrix(mtcars)
dapply(m, log)                            # Same thing
dapply(m, log, return = "data.frame")     # Return data frame from matrix
dapply(mtcars, sum); dapply(m, sum)      # Computing sum of each column, return as vector
dapply(mtcars, sum, drop = FALSE)         # This returns a data.frame of 1 row
dapply(mtcars, sum, MARGIN = 1)         # Compute row-sum of each column, return as vector
dapply(m, sum, MARGIN = 1)           # Same thing for matrices, faster than apply(m, 1, sum)
dapply(m, sum, MARGIN = 1, drop = FALSE) # Gives matrix with one column
dapply(m, quantile, MARGIN = 1)           # Compute row-quantiles
dapply(m, quantile)                       # Column-quantiles
dapply(mtcars, quantile, MARGIN = 1)    # Same for data frames, output is also a data.frame
dapply(mtcars, quantile)

# Let's now take a more complex classed object, like a dplyr grouped tibble
library(dplyr)
gmtcars <- group_by(mtcars,cyl,vs,am)
dapply(gmtcars, log)                      # Still gives a grouped tibble back
dapply(gmtcars, log, MARGIN = 1)
dapply(gmtcars, quantile, MARGIN = 1)     # Also works for quantiles
dapply(gmtcars, log, return = "matrix")  # Output as matrix
```

---

descr *Detailed Statistical Description of Data Frame*

Description

descr offers concise description of each variable in a data frame. It is built as a wrapper around qsu, but by default also computes frequency tables with percentages for categorical variables, and quantiles and the number of distinct values for numeric variables (next to the mean, sd, min, max, skewness and kurtosis computed by qsu).

Usage

```
descr(X, Ndistinct = TRUE, higher = TRUE, table = TRUE,
      Qprobs = c(0.01, 0.05, 0.25, 0.5, 0.75, 0.95, 0.99),
      cols = NULL, label.attr = "label", ...)

## S3 method for class 'descr'
```

```
print(x, n = 6, perc = TRUE, summary = TRUE, ...)

## S3 method for class 'descr'
as.data.frame(x, ...)
```

Arguments

X	a data.frame or list of atomic vectors. Atomic vectors, matrices or arrays can be passed but will first be coerced to data.frame using qDF.
Ndistinct	logical. TRUE (default) computes the number of distinct values on all variables using fNdistinct.
higher	logical. Argument is passed down to qsu: TRUE (default) computes the skewness and the kurtosis.
table	logical. TRUE (default) calls table on all categorical variables (excluding Date variables).
Qprobs	probabilities for quantiles to compute on numeric variables, passed down to quantile. If something non-numeric is passed (i.e. NULL, FALSE, NA, "" etc.), no quantiles are computed.
cols	select columns to describe using column names, indices or a function (i.e. is.numeric).
label.attr	character. The name of a label attribute to display for each variable (if variables are labeled).
...	other arguments passed to qsu.default.
x	an object of class 'descr'.
n	integer. The number of first and last entries to display of the table computed for categorical variables.
perc	logical. TRUE (default) adds percentages in brackets behind the frequencies in the table for categorical variables.
summary	logical. TRUE (default) computes and displays a summary of the frequencies if the size of the table for a categorical variables exceeds 2*n.

Details

descr was heavily inspired by Hmisc::describe, but computes about 10x faster. The performance is comparable to base::summary. descr was built as a wrapper around qsu, to enrich the set of statistics computed by qsu for both numeric and categorical variables.

qsu itself is yet about 10x faster than descr, and is optimized for grouped, panel-data and weighted statistics. It is possible to also compute grouped, panel-data and/or weighted statistics with desc by passing group-ids to g, panel-ids to pid or a weight vector to w. These arguments are handed down to qsu.default and only affect the statistics natively computed by qsu, i.e. passing a weight vector produces a weighted mean, sd, skewness and kurtosis but not weighted quantiles.

The list-object returned from descr can be converted to a tidy data.frame using as.data.frame. This representation will not include frequency tables computed for categorical variables, and the method cannot handle arrays of statistics (applicable when g or pid arguments are passed to descr, in that case as.data.frame.descr will throw an appropriate error).

Value

A 2-level nested list, the top-level containing the statistics computed for each variable, which are themselves stored in a list containing the class, the label, the basic statistics and quantiles / tables computed for the variable. The object is given a class 'descr' and also has the number of observations in the dataset attached as an 'N' attribute, as well as an attribute 'arstat' indicating whether the object contains arrays of statistics.

See Also

qsu, pwcor, Fast Statistical Functions, Collapse Overview

Examples

```
## Standard Use
descr(iris)
descr(wlddev)
descr(GGDC10S)

as.data.frame(descr(wlddev))

## Passing Arguments down to qsu: For Panel-Data Statistics
descr(iris, pid = iris$Species)
descr(wlddev, pid = wlddev$iso3c)

## Grouped Statistics
descr(iris, g = iris$Species)
descr(GGDC10S, g = GGDC10S$Region)
```

---

extract-list *Find and Extract / Subset List Elements*

Description

A suite of functions to subset or extract from (potentially complex) lists and list-like structures. Subsetting may occur according to certain data types, using identifier functions, element names or regular expressions to search the list for certain objects.

- `atomic_elem` and `list_elem` are non-recursive functions to extract and replace the atomic and sub-list elements at the top-level of the list tree.
- `reg_elem` is the recursive equivalent of `atomic_elem` and returns the 'regular' part of the list - with atomic elements in the final nodes. See `is.regular` and `is.unlistable`. `irreg_elem` returns all the non-regular elements (i.e. call and terms objects, formulas, etc...). See Examples.
- `get_elem` returns the part of the list responding to either an identifier function, regular expression or exact element names, or indices applied to all final objects. `has_elem` checks for the existence of the searched element and returns `TRUE` if a match is found. See Examples.

Usage

```
## Non-recursive (top-level) subsetting and replacing
atomic_elem(l, return = "sublist", keep.class = FALSE)
atomic_elem(l) <- value
list_elem(l, return = "sublist", keep.class = FALSE)
list_elem(l) <- value

## Recursive separation of regular (atomic) and irregular (non-atomic) parts
reg_elem(l, recursive = TRUE, keep.tree = FALSE, keep.class = FALSE)
irreg_elem(l, recursive = TRUE, keep.tree = FALSE, keep.class = FALSE)

## Extract elements using a function or regular expression
get_elem(l, elem, recursive = TRUE, DF.as.list = TRUE, keep.tree = FALSE,
         keep.class = FALSE, regex = FALSE, ...)

## Check for the existence of elements
has_elem(l, elem, recursive = TRUE, DF.as.list = TRUE, regex = FALSE, ...)
```

Arguments

`l`	a list.
`value`	a list of the same length as the extracted subset of `l`.
`elem`	a function returning `TRUE` or `FALSE` when applied to elements of `l`, or a character vector of element names or regular expressions (if `regex = TRUE`). `get_elem` also supports a vector or indices which will be used to subset all final objects.
`return`	an integer or string specifying what the selector function should return. The options are:

Int.	*String*	*Description*
1	"sublist"	subset of data.frame (default)
2	"names"	column names
3	"indices"	column indices
4	"named_indices"	named column indices
5	"logical"	logical selection vector
6	"named_logical"	named logical vector

Note: replacement functions only replace data, However column names are replaced together with the data.

`recursive`	logical. should the list search be recursive (i.e. go though all the elements), or just at the top-level?
`DF.as.list`	logical. treat data.frame's like (sub-)lists or like atomic elements?
`keep.tree`	logical. `TRUE` always returns the entire list tree leading up to all matched results, while `FALSE` drops the top-level part of the tree if possible.
`keep.class`	logical. for classed objects: Should the class be retained?

regex	logical. should regular expression search be used on the list names, or only exact matches?
...	further arguments to grep (if regex = TRUE).

Details

A list is made up of regular and irregular elements. I defined regular elements as all elements that are either atomic or a list (see is.regular). reg_elem with recursive = TRUE therefore extracts the subset of the list tree leading up to atomic elements in the final nodes. This part of the list tree is unlistable - calling is.unlistable(reg_elem(l)) will be TRUE for all lists l. Conversely, all elements left behind by reg_elem will be picked up be irreg_elem (if available). Thus is.unlistable(irreg_elem(l)) is always FALSE for lists with irregular elements (otherwise irreg_elem returns an empty list).

If keep.tree = TRUE, reg_elem, irreg_elem and get_elem always return the entire list tree, but cut off all of the branches not leading to the desired result. If keep.tree = FALSE, top-level parts of the tree are omitted so far this is possible. For example in a nested list with three levels and one data-matrix in one of the final branches, get_elem(l,is.matrix,keep.tree = TRUE) will return a list (lres) of depth 3, from which the matrix can be accessed as lres[[1]][[1]][[1]]. This however does not make much sense. get_elem(l,is.matrix,keep.tree = FALSE) will therefore figgure out that it can drop the entire tree and return just the matrix. keep.tree = FALSE makes additional optimizations if matching elements are at far-apart corners in a nested structure, by only preserving the hierarchy if elements are above each other on the same branch. Thus for a list l <-list(list(2,list("a",1)),list(1,list("b",2))) calling get_elem(l,is.character) will just return list("a","b").

See Also

List Processing, Collapse Overview

Examples

```
l <- list(list(2,list("a",1)),list(1,list("b",2)))
has_elem(l, is.logical)
has_elem(l, is.character)
get_elem(l, is.character)
get_elem(l, is.character, keep.tree = TRUE)

l <- lm(mpg ~ cyl + vs, data = mtcars)
str(reg_elem(l))
str(irreg_elem(l))
get_elem(l, is.matrix)
get_elem(l, "residuals")
get_elem(l, "fit", regex = TRUE)
has_elem(l,"tol")
get_elem(l, "tol")
```

`fbetween, fwithin` *Fast Between (Averaging) and Within (Centering) Transformations*

Description

`fbetween` and `fwithin` are S3 generics to efficiently obtain between-transformed (averaged) or within-transformed (demeaned) data. These operations can be performed groupwise and/or weighted. `B` and `W` are wrappers around `fbetween` and `fwithin` representing the 'between-operator' and the 'within-operator'. `B` / `W` provide more flexibility than `fbetween` / `fwithin` when applied to data frames (i.e. column subsetting, formula input, auto-renaming and id-variable-preservation capabilities...), but are otherwise identical.

(`fbetween` and `fwithin` are simple programmers functions in style of the Fast Statistical Functions while `B` and `W` are more practical to use in regression formulas or for ad-hoc computations on data frames.)

Usage

```
fbetween(x, ...)
 fwithin(x, ...)
       B(x, ...)
       W(x, ...)

## Default S3 method:
fbetween(x, g = NULL, w = NULL, na.rm = TRUE, fill = FALSE, ...)
## Default S3 method:
fwithin(x, g = NULL, w = NULL, na.rm = TRUE, mean = 0, ...)
## Default S3 method:
B(x, g = NULL, w = NULL, na.rm = TRUE, fill = FALSE, ...)
## Default S3 method:
W(x, g = NULL, w = NULL, na.rm = TRUE, mean = 0, ...)

## S3 method for class 'matrix'
fbetween(x, g = NULL, w = NULL, na.rm = TRUE, fill = FALSE, ...)
## S3 method for class 'matrix'
fwithin(x, g = NULL, w = NULL, na.rm = TRUE, mean = 0, ...)
## S3 method for class 'matrix'
B(x, g = NULL, w = NULL, na.rm = TRUE, fill = FALSE, stub = "B.", ...)
## S3 method for class 'matrix'
W(x, g = NULL, w = NULL, na.rm = TRUE, mean = 0, stub = "W.", ...)

## S3 method for class 'data.frame'
fbetween(x, g = NULL, w = NULL, na.rm = TRUE, fill = FALSE, ...)
## S3 method for class 'data.frame'
fwithin(x, g = NULL, w = NULL, na.rm = TRUE, mean = 0, ...)
## S3 method for class 'data.frame'
B(x, by = NULL, w = NULL, cols = is.numeric, na.rm = TRUE,
```

```
  fill = FALSE, stub = "B.", keep.by = TRUE, keep.w = TRUE, ...)
## S3 method for class 'data.frame'
W(x, by = NULL, w = NULL, cols = is.numeric, na.rm = TRUE,
  mean = 0, stub = "W.", keep.by = TRUE, keep.w = TRUE, ...)

# Methods for compatibility with plm:

## S3 method for class 'pseries'
fbetween(x, effect = 1L, w = NULL, na.rm = TRUE, fill = FALSE, ...)
## S3 method for class 'pseries'
fwithin(x, effect = 1L, w = NULL, na.rm = TRUE, mean = 0, ...)
## S3 method for class 'pseries'
B(x, effect = 1L, w = NULL, na.rm = TRUE, fill = FALSE, ...)
## S3 method for class 'pseries'
W(x, effect = 1L, w = NULL, na.rm = TRUE, mean = 0, ...)

## S3 method for class 'pdata.frame'
fbetween(x, effect = 1L, w = NULL, na.rm = TRUE, fill = FALSE, ...)
## S3 method for class 'pdata.frame'
fwithin(x, effect = 1L, w = NULL, na.rm = TRUE, mean = 0, ...)
## S3 method for class 'pdata.frame'
B(x, effect = 1L, w = NULL, cols = is.numeric, na.rm = TRUE,
  fill = FALSE, stub = "B.", keep.ids = TRUE, keep.w = TRUE, ...)
## S3 method for class 'pdata.frame'
W(x, effect = 1L, w = NULL, cols = is.numeric, na.rm = TRUE,
  mean = 0, stub = "W.", keep.ids = TRUE, keep.w = TRUE, ...)

# Methods for compatibility with dplyr:

## S3 method for class 'grouped_df'
fbetween(x, w = NULL, na.rm = TRUE, fill = FALSE,
         keep.group_vars = TRUE, keep.w = TRUE, ...)
## S3 method for class 'grouped_df'
fwithin(x, w = NULL, na.rm = TRUE, mean = 0,
        keep.group_vars = TRUE, keep.w = TRUE, ...)
## S3 method for class 'grouped_df'
B(x, w = NULL, na.rm = TRUE, fill = FALSE,
  stub = "B.", keep.group_vars = TRUE, keep.w = TRUE, ...)
## S3 method for class 'grouped_df'
W(x, w = NULL, na.rm = TRUE, mean = 0,
  stub = "W.", keep.group_vars = TRUE, keep.w = TRUE, ...)
```

Arguments

x	a numeric vector, matrix, data.frame, panel-series (`plm::pseries`), panel-data.frame (`plm::pdata.frame`) or grouped tibble (`dplyr::grouped_df`).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.

by — *B and W data.frame method*: Same as g, but also allows one- or two-sided formulas i.e. ~ group1 or var1 + var2 ~ group1 + group2. See Examples.

w — a numeric vector of (non-negative) weights. B/W data frame and pdata.frame methods also allow a one-sided formula i.e. ~ weightcol. The grouped_df (*dplyr*) method supports lazy-evaluation. See Examples.

cols — *data.frame method*: Select columns to center/average using a function, column names or indices. Default: All numeric variables. *Note*: cols is ignored if a two-sided formula is passed to by.

na.rm — logical. skip missing values in x when computing averages. If na.rm = FALSE and a NA or NaN is encountered, the average for that group will be NA, and all data points belonging to that group will also be NA.

effect — *plm* methods: Select which panel identifier should be used as grouping variable. 1L means first variable in the plm::index, 2L the second etc. if more than one integer is supplied, the corresponding index-variables are interacted.

stub — a prefix or stub to rename all transformed columns. FALSE will not rename columns.

fill — *option to* fbetween/B: Logical. TRUE will overwrite missing values in x with the respective average. By default missing values in x are preserved.

mean — *option to* fwithin/W: The mean to center on, default is 0, but a different mean can be supplied and will be added to the data after the centering is performed. A special option when performing grouped centering is mean = "overall.mean". In that case the overall mean of the data will be added after subtracting out group means.

keep.by, keep.ids, keep.group_vars — *B and W data.frame, pdata.frame and grouped_df methods*: Logical. Retain grouping / panel-identifier columns in the output. For data frames this only works if grouping variables were passed in a formula.

keep.w — *B and W data.frame, pdata.frame and grouped_df methods*: Logical. Retain column containing the weights in the output. Only works if w is passed as formula / lazy-expression.

... — arguments to be passed to or from other methods.

Details

Without groups, fbetween/B replaces all data points in x with their mean or weighted mean (if w is supplied). Similarly fwithin/W subtracts the mean from all data points i.e. centers the data on the mean.

With groups supplied to g, the replacement / centering performed by fbetween/B | fwithin/W becomes groupwise. I like to think of this in terms of panel data: If x is a vector in such a dataset, xit denotes a single data-point belonging to group i in time-period t (t need not be a time-period). Then xi. denotes x, averaged over t. fbetween/B now returns xi. and fwithin/W returns x -xi.. Thus for any data x and any grouping vector g: B(x,g) + W(x,g) = xi. + x -xi. = x. In terms of variance, fbetween/B only retains the variance between group averages, while fwithin/W, by subtracting out group means, only retains the variance within those groups.

The data replacement performed by fbetween/B can keep (default) or overwrite missing values (option fill = TRUE) in x. fwithin/W can center data simply (default), or add back a mean after centering (option mean = value), or add the overall mean in groupwise computations (option mean = "overall.mean"). Let x.. denote the overall mean of x, then fwithin/W with mean = "overall.mean" returns x -xi. + x.. instead of x -xi.. This is useful to get rid of group-differences but preserve the overall level of the data (as simple groupwise centering will set the overall mean of the data to 0, or any other arbitrary value passed to mean). In regression analysis, centering with mean = "overall.mean" will only change the constant term. See Examples.

Value

fbetween/B returns x with every element replaced by its (groupwise) mean (xi.). fwithin/W returns x where every element was subtracted its (groupwise) mean (x -xi. or x -xi. + mean or x -xi. + x..). See Details.

See Also

fHDbetween/HDB and fHDwithin/HDW, fscale/STD, TRA, Data Transformations, Collapse Overview

Examples

```
## Simple centering and averaging
fbetween(mtcars)
B(mtcars)
fwithin(mtcars)
W(mtcars)
fbetween(mtcars) + fwithin(mtcars) == mtcars # This should be true apart from rounding errors

## Groupwise centering and averaging
fbetween(mtcars, mtcars$cyl)
 fwithin(mtcars, mtcars$cyl)
fbetween(mtcars, mtcars$cyl) + fwithin(mtcars, mtcars$cyl) == mtcars

W(wlddev, ~ iso3c, cols = 9:12)    # Center the 4 series in this dataset by country
cbind(get_vars(wlddev,"iso3c"),    # Same thing done manually using fwithin...
      add_stub(fwithin(get_vars(wlddev,9:12), wlddev$iso3c), "W."))

## Using B() and W() in regressions:

# Several ways of running the same regression with cyl-fixed effects
lm(W(mpg,cyl) ~ W(carb,cyl), data = mtcars)                 # Centering each individually
lm(mpg ~ carb, data = W(mtcars, ~ cyl, stub = FALSE))         # Centering the entire data
lm(mpg ~ carb, data = W(mtcars, ~ cyl, stub = FALSE,        # Here only the intercept changes
                        mean = "overall.mean"))
lm(mpg ~ carb + B(carb,cyl), data = mtcars)                    # Procedure suggested by
# ...Mundlak (1978) - partialling out group averages amounts to the same as demeaning the data

# Now with cyl, vs and am fixed effects
lm(W(mpg,list(cyl,vs,am)) ~ W(carb,list(cyl,vs,am)), data = mtcars)
lm(mpg ~ carb, data = W(mtcars, ~ cyl + vs + am, stub = FALSE))
lm(mpg ~ carb + B(carb,list(cyl,vs,am)), data = mtcars)
```

```
# Now with cyl, vs and am fixed effects weighted by hp:
lm(W(mpg,list(cyl,vs,am),hp) ~ W(carb,list(cyl,vs,am),hp), data = mtcars)
lm(mpg ~ carb, data = W(mtcars, ~ cyl + vs + am, ~ hp, stub = FALSE))
lm(mpg ~ carb + B(carb,list(cyl,vs,am),hp), data = mtcars)     # Gives a different coefficient!!
```

`fdiff` *Fast (Quasi-, Log-) Differences for Time-Series and Panel Data*

Description

`fdiff` is a S3 generic to compute (sequences of) suitably lagged / leaded and iterated differences, quasi-differences, log-differences or quasi-log-differences. The difference and log-difference operators `D` and `Dlog` also exists as parsimonious wrappers around `fdiff`. Apart from being more parsimonious, they provide a bit more flexibility than `fdiff` when applied to data frames.

Usage

```
  fdiff(x, n = 1, diff = 1, ...)
      D(x, n = 1, diff = 1, ...)
   Dlog(x, n = 1, diff = 1, ...)

## Default S3 method:
fdiff(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA, logdiff = FALSE, rho = 1,
      stubs = TRUE, ...)
## Default S3 method:
D(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA, rho = 1,
  stubs = TRUE, ...)
## Default S3 method:
Dlog(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA, rho = 1, stubs = TRUE, ...)

## S3 method for class 'matrix'
fdiff(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA, logdiff = FALSE, rho = 1,
      stubs = TRUE, ...)
## S3 method for class 'matrix'
D(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA, rho = 1,
  stubs = TRUE, ...)
## S3 method for class 'matrix'
Dlog(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA, rho = 1, stubs = TRUE, ...)

## S3 method for class 'data.frame'
fdiff(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA, logdiff = FALSE, rho = 1,
      stubs = TRUE, ...)
## S3 method for class 'data.frame'
D(x, n = 1, diff = 1, by = NULL, t = NULL, cols = is.numeric,
  fill = NA, rho = 1, stubs = TRUE, keep.ids = TRUE, ...)
## S3 method for class 'data.frame'
```

```
Dlog(x, n = 1, diff = 1, by = NULL, t = NULL, cols = is.numeric,
     fill = NA, rho = 1, stubs = TRUE, keep.ids = TRUE, ...)

# Methods for compatibility with plm:

## S3 method for class 'pseries'
fdiff(x, n = 1, diff = 1, fill = NA, logdiff = FALSE, rho = 1, stubs = TRUE, ...)
## S3 method for class 'pseries'
D(x, n = 1, diff = 1, fill = NA, rho = 1, stubs = TRUE, ...)
## S3 method for class 'pseries'
Dlog(x, n = 1, diff = 1, fill = NA, rho = 1, stubs = TRUE, ...)

## S3 method for class 'pdata.frame'
fdiff(x, n = 1, diff = 1, fill = NA, logdiff = FALSE, rho = 1, stubs = TRUE, ...)
## S3 method for class 'pdata.frame'
D(x, n = 1, diff = 1, cols = is.numeric, fill = NA, rho = 1, stubs = TRUE,
  keep.ids = TRUE, ...)
## S3 method for class 'pdata.frame'
Dlog(x, n = 1, diff = 1, cols = is.numeric, fill = NA, rho = 1, stubs = TRUE,
     keep.ids = TRUE, ...)

# Methods for compatibility with dplyr:

## S3 method for class 'grouped_df'
fdiff(x, n = 1, diff = 1, t = NULL, fill = NA, logdiff = FALSE, rho = 1, stubs = TRUE,
      keep.ids = TRUE, ...)
## S3 method for class 'grouped_df'
D(x, n = 1, diff = 1, t = NULL, fill = NA, rho = 1, stubs = TRUE,
  keep.ids = TRUE, ...)
## S3 method for class 'grouped_df'
Dlog(x, n = 1, diff = 1, t = NULL, fill = NA, rho = 1, stubs = TRUE,
     keep.ids = TRUE, ...)
```

Arguments

x	a numeric vector, matrix, data.frame, panel-series (plm::pseries), panel-data.frame (plm::pdata.frame) or grouped tibble (dplyr::grouped_df).
n	a integer vector indicating the number of lags or leads.
diff	a vector of integers > 1 indicating the order of differencing / log-differencing.
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
by	*data.frame method*: Same as g, but also allows one- or two-sided formulas i.e. ~ group1 or var1 + var2 ~ group1 + group2. See Examples.
t	same input as g, to indicate the time-variable. For safe computation of differences on unordered time-series and panels. *Notes*: data.frame method also allows name, index or one-sided formula i.e. ~time. grouped_df method also allows lazy-evaluation i.e. time (no quotes).

cols	*data.frame method*: Select columns to difference using a function, column names or indices. Default: All numeric variables. *Note*: cols is ignored if a two-sided formula is passed to by.
fill	value to insert when vectors are shifted. Default is NA.
logdiff	logical. TRUE Computes log-differences instead. See Details.
rho	double. Autocorrelation parameter. Set to a value between 0 and 1 for quasi-differencing. However any numeric value can be supplied.
stubs	logical. TRUE will rename all differenced columns by adding prefixes "LnDdiff." / "FnDdiff." for differences "LnDlogdiff." / "FnDlogdiff." for log-differences and replacing "D" / "Dlog" with "QD" / "QDlog" for quasi-differences.
keep.ids	*data.frame / pdata.frame / grouped_df methods*: Logical. Drop all panel-identifiers from the output (which includes all variables passed to by or t). *Note*: For panel-data.frame's and grouped tibbles identifiers are dropped, but the 'index' / 'groups' attributes are kept.
...	arguments to be passed to or from other methods.

Details

By default, fdiff/D/Dlog return x with all columns differenced / log-differenced. Differences are computed as repeat(diff) x[i] -rho*x[i-n], and log-differences as repeat(diff) log(x[i]) -rho*log(x[i-n]). If rho < 1, this becomes quasi- (or partial) differencing, which is a technique suggested by Cochrane and Orcutt (1949) to deal with serial correlation in regression models, where rho is typically estimated by running a regression of the model residuals on the lagged residuals. Setting diff = 2 returns differences of differences etc... and setting n = 2 returns simple differences computed by subtracting twice-lagged x from x. It is also possible to compute forward differences by passing negative n values. n also supports arbitrary vectors of integers (lags), and diff supports positive sequences of integers (differences):

If more than one value is passed to n and/or diff, the data is expanded-wide as follows: If x is an atomic vector or time-series, a (time-series) matrix is returned with columns ordered first by lag, then by difference. If x is a matrix or data.frame, each column is expanded in like manor such that the output has ncol(x)*length(n)*length(diff) columns ordered first by column name, then by lag, then by difference.

With groups/panel-identifiers supplied to g/by, fdiff/D/Dlog efficiently compute panel-differences. If t is left empty, the data needs to be ordered such that all values belonging to a group are consecutive and in the right order. It is not necessary that the groups themselves occur in the right order. If time-variable(s) are supplied to t, the panel is fully identified and differences can be securely computed even if the data is completely unordered.

fdiff/D/Dlog supports balanced panels and unbalanced panels where various individuals are observed for different time-sequences (both start, end and duration of observation can differ for each individual), but does not natively support irregularly spaced time-series and panels. For computational details and efficiency considerations see the help page for flag. A work-around for differencing irregular panels is easily achieved with the help of seqid.

It is also possible to compute differences on unordered vectors / time-series (thus utilizing t but leaving g/by empty).

The methods applying to *plm* objects (panel-series and panel-data.frames) automatically utilize the panel-identifiers attached to these objects and thus securely compute fully identified panel-differences. If these objects have > 2 panel-identifiers attached to them, the last identifier is assumed to be the time-variable, and the others are taken as grouping-variables and interacted.

Value

`x` differenced `diff` times using lags `n` of itself. Quasi and log-differences are toggled by the `rho` and `logdiff` arguments or the `Dlog` operators. Computations can be grouped by `g`/`by` and/or ordered by `t`. See Details and Examples.

References

Cochrane, D.; Orcutt, G. H. (1949). Application of Least Squares Regression to Relationships Containing Auto-Correlated Error Terms. *Journal of the American Statistical Association*. 44 (245): 32-61.

See Also

`flag/L/F`, `fgrowth/G`, Time-Series and Panel-Series, Collapse Overview

Examples

```
## Simple Time-Series: AirPassengers
D(AirPassengers)                      # 1st difference, same as fdiff(AirPassengers)
D(AirPassengers,-1)                   # forward difference
Dlog(AirPassengers)                   # log-difference
D(AirPassengers,1,2)                  # second difference
Dlog(AirPassengers,1,2)               # second log-difference
D(AirPassengers,12)                   # seasonal difference (data is monthly)
D(AirPassengers,                      # quasi-difference, See a better example below
  rho = pwcor(AirPassengers, L(AirPassengers)))                  #

D(AirPassengers,-2:2,1:3)           # sequence of leaded/lagged and iterated differences

# let's do some visual analysis
plot(AirPassengers)                    # plot the series - seasonal pattern is evident
plot(stl(AirPassengers, "periodic"))  # Seasonal decomposition
plot(D(AirPassengers,c(1,12),1:2))   # plotting ordinary and seasonal first and second differences
plot(stl(window(D(AirPassengers,12),  # Taking seasonal differences removes most seasonal variation
                1950), "periodic"))


## Time-Series Matrix of 4 EU Stock Market Indicators, recorded 260 days per year
plot(D(EuStockMarkets, c(0, 260)))                    # Plot series and annual differnces
mod <- lm(DAX ~., L(EuStockMarkets, c(0, 260)))      # Regressing the DAX on its annual lag
summary(mod)                                        # and the levels and annual lags others
r <- residuals(mod)                                      # Obtain residuals
pwcor(r, L(r))                                           # Residual Autocorrelation
fFtest(r, L(r))                                        # F-test of residual autocorrelation
                                                         # (better use lmtest::bgtest)
modCO <- lm(QD1.DAX ~., D(L(EuStockMarkets, c(0, 260)), # Cochrane-Orcutt (1949) estimation
```

```
                        rho = pwcor(r, L(r))))
summary(modCO)
rCO <- residuals(modCO)
fFtest(rCO, L(rCO))                                    # No more autocorrelation

## World Development Panel Data
head(fdiff(num_vars(wlddev), 1, 1,               # Computes differences of numeric variables
           wlddev$country, wlddev$year))             # fdiff requires external inputs...
head(D(wlddev, 1, 1, ~country, ~year))                # Differences of numeric variables
head(D(wlddev, 1, 1, ~country))                  # Without t: Works because data is ordered
head(D(wlddev, 1, 1, PCGDP + LIFEEX ~ country, ~year))  # Difference of GDP & Life Expectancy
head(D(wlddev, 0:1, 1, ~ country, ~year, cols = 9:10))  # Same, also retaining original series
head(D(wlddev, 0:1, 1, ~ country, ~year, 9:10,          # Dropping id columns
       keep.ids = FALSE))

# Dynamic Panel-Data Models:
summary(lm(D(PCGDP,1,1,iso3c,year) ~              # Diff. GDP regressed on it's lagged level
          L(PCGDP,1,iso3c,year) +                 # and the difference of Life Expanctancy
            D(LIFEEX,1,1,iso3c,year), data = wlddev))

g = qF(wlddev$country)                           # Omitting t and precomputing g allows for
summary(lm(D(PCGDP,1,1,g) ~ L(PCGDP,1,g) +          # a bit more parsimonious specification
                           D(LIFEEX,1,1,g), wlddev))

summary(lm(D1.PCGDP ~.,                             # Now adding level and lagged level of
L(D(wlddev,0:1,1, ~ country, ~year,9:10),0:1,        # LIFEEX and lagged differences rates
  ~ country, ~year, keep.ids = FALSE)[-1]))

## Using plm can make things easier, but avoid attaching or 'with' calls:
pwlddev <- plm::pdata.frame(wlddev, index = c("country","year"))
head(D(pwlddev, 0:1, 1, 9:10))                     # Again differences of LIFEEX and PCGDP
PCGDP <- pwlddev$PCGDP                                 # A panel-Series of GDP per Capita
D(PCGDP)                                               # Differencing the panel series.
summary(lm(D1.PCGDP ~.,                              # Running the dynamic model again ->
           data = L(D(pwlddev,0:1,1,9:10),0:1,          # code becomes a bit simpler
                    keep.ids = FALSE)[-1]))

# One could be tempted to also do something like this, but THIS DOES NOT WORK!!!:
# lm drops the attributes (-> with(pwlddev, PCGDP) drops attr. so D.default and L.matrix are used)
summary(lm(D(PCGDP) ~ L(D(PCGDP,0:1)) + L(D(LIFEEX,0:1),0:1), pwlddev))

# To make it work, one needs to create pseries (note: attach(pwlddev) also won't work)
LIFEEX <- pwlddev$LIFEEX
summary(lm(D(PCGDP) ~ L(D(PCGDP,0:1)) + L(D(LIFEEX,0:1),0:1))) # THIS WORKS !!

## Using dplyr:
library(dplyr)
wlddev %>% group_by(country) %>%
         select(PCGDP,LIFEEX) %>% fdiff(0:1,1:2)     # Adding a first and second difference
wlddev %>% group_by(country) %>%
             select(year,PCGDP,LIFEEX) %>% D(0:1,1:2,year) # Also using t (safer)
wlddev %>% group_by(country) %>%                           # Ddropping id's
             select(year,PCGDP,LIFEEX) %>% D(0:1,1:2,year, keep.ids = FALSE)
```

`ffirst, flast` *Fast (Grouped) First and Last Value for Matrix-Like Objects*

Description

`ffirst` and `flast` are S3 generic functions that (column-wise) returns the first and last values in `x`, (optionally) grouped by `g`. The `TRA` argument can further be used to transform `x` using its (groupwise) first and last values.

Usage

```
ffirst(x, ...)
flast(x, ...)

## Default S3 method:
ffirst(x, g = NULL, TRA = NULL, na.rm = TRUE,
       use.g.names = TRUE, ...)
## Default S3 method:
flast(x, g = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = TRUE, ...)

## S3 method for class 'matrix'
ffirst(x, g = NULL, TRA = NULL, na.rm = TRUE,
       use.g.names = TRUE, drop = TRUE, ...)
## S3 method for class 'matrix'
flast(x, g = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'data.frame'
ffirst(x, g = NULL, TRA = NULL, na.rm = TRUE,
       use.g.names = TRUE, drop = TRUE, ...)
## S3 method for class 'data.frame'
flast(x, g = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'grouped_df'
ffirst(x, TRA = NULL, na.rm = TRUE,
       use.g.names = FALSE, keep.group_vars = TRUE, ...)
## S3 method for class 'grouped_df'
flast(x, TRA = NULL, na.rm = TRUE,
      use.g.names = FALSE, keep.group_vars = TRUE, ...)
```

Arguments

`x`	a vector, matrix, data.frame or grouped tibble (`dplyr::grouped_df`).

g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
TRA	an integer or quoted operator indicating the transformation to perform: 1 - "replace_fill" | 2 - "replace" | 3 - "-" | 4 - "-+" | 5 - "/" | 6 - "%" | 7 - "+" | 8 - "*" | 9 - "%%" | 10 - "-%%". See TRA.
na.rm	logical. Skip missing values and choose the first / last non-missing value i.e. if the first (1) / last (n) value is NA, take the second (2) / second-to-last (n-1) value etc...
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.frame method). No row-names are generated for data.tables and grouped tibbles.
drop	*matrix and data.frame method:* drop dimensions and return an atomic vector if g = NULL and TRA = NULL.
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation.
...	arguments to be passed to or from other methods.

Value

ffirst returns the first value in x, grouped by g, or (if TRA is used) x transformed by its first value, grouped by g. Similarly flast returns the last value in x, ...

See Also

Fast Statistical Functions, Collapse Overview

Examples

```
## default vector method
ffirst(airquality$Ozone)                   # Simple first value
ffirst(airquality$Ozone, airquality$Month) # Grouped first value
ffirst(airquality$Ozone, airquality$Month,
      na.rm = FALSE)                  # Grouped first, but without skipping initial NA's

## data.frame method
ffirst(airquality)
ffirst(airquality, airquality$Month)
ffirst(airquality, airquality$Month, na.rm = FALSE) # Again first Ozone measurement in month 6 is NA

## matrix method
aqm <- qM(airquality)
ffirst(aqm)
ffirst(aqm, airquality$Month) # etc...

## method for grouped tibbles - for use with dplyr
library(dplyr)
airquality %>% group_by(Month) %>% ffirst
airquality %>% group_by(Month) %>% select(Ozone) %>% ffirst(na.rm = FALSE)
```

```
# Note: All examples generalize to flast!
```

`fFtest` *Fast F-test of Linear Models (with Factors)*

Description

`fFtest` computes an R-squared based F-test for the exclusion of the variables in `exc`, where the full (unrestricted) model is defined by variables supplied to both `exc` and `X`. The test is efficient and designed for cases where both `exc` and `X` may contain multiple factors and continuous variables.

Usage

```
fFtest(y, exc, X = NULL, full.df = TRUE, ...)
```

Arguments

`y`	a numeric vector: The dependent variable.
`exc`	a numeric vector, factor, numeric matrix or list / data.frame of numeric vectors and/or factors: Variables to test / exclude.
`X`	a numeric vector, factor, numeric matrix or list / data.frame of numeric vectors and/or factors: Covariates to include in both the restricted (without `exc`) and unrestricted model. If left empty (`X = NULL`), the test amounts to the F-test of the regression of `y` on `exc`.
`full.df`	logical. If `TRUE` (default), the degrees of freedom are calculated as if both restricted and unrestricted models were estimated using `lm()` (i.e. as if factors were expanded to matrices of dummies). `FALSE` only uses one degree of freedom per factor.
`...`	other arguments passed to `lfe::demeanlist`, the workhorse function underlying `fHDwithin`.

Details

Factors and continuous regressors are efficiently projected out using `fHDwithin`, and the option `full.df` regulates whether a degree of freedom is subtracted for each used factor level (equivalent to dummy-variable estimator / expanding factors), or only one degree of freedom per factor (fixed-effects estimation / treating factors as variables). The test automatically removes missing values and considers only the complete cases of `y,exc` and `X`. Unused factor levels in `exc` and `X` are dropped.

Value

A 5 x 3 numeric matrix of statistics. The columns contain statistics:

1. the R-squared of the model
2. the numerator degrees of freedom i.e. the number of variables (k) and used factor levels if `full.df = TRUE`

3. the denominator degrees of freedom: N - k - 1.
4. the F-statistic
5. the corresponding P-value

The rows show these statistics for:

1. the Full (unrestricted) Model (`y ~ exc + X`)
2. the Restricted Model (`y ~ X`)
3. the Exclusion Restriction of `exc`. The R-squared shown is simply the difference of the full and restricted R-Squared's, not the R-Squared of the model `y ~ exc`.

If `X = NULL`, only a vector of the same 5 statistics testing the model (`y ~ exc`) is shown.

See Also

`fHDbetween/HDB` and `fHDwithin/HDW`, Data Transformations, Collapse Overview

Examples

```
## We could use fFtest as a seasonality test:
fFtest(AirPassengers, qF(cycle(AirPassengers)))  # Testing for level-seasonality
fFtest(AirPassengers, qF(cycle(AirPassengers)),  # Seasonality test around a cubic trend
        poly(seq_along(AirPassengers), 3))

## A more classical example with only continuous variables
fFtest(mtcars$mpg, mtcars[c("cyl","vs")], mtcars[c("hp","carb")])

## Now encoding cyl and vs as factors
fFtest(mtcars$mpg, dapply(mtcars[c("cyl","vs")], qF), mtcars[c("hp","carb")])

## Using iris data: A factor and a continuous variable excluded
fFtest(iris$Sepal.Length, iris[4:5], iris[2:3])

## Testing the significance of country-FE in regression of GDP on life expectancy
fFtest(wlddev$PCGDP, wlddev$iso3c, wlddev$LIFEEX)

## Ok, country-FE are significant, what about adding time-FE
fFtest(wlddev$PCGDP, qF(wlddev$year), wlddev[c("iso3c","LIFEEX")])

# Same test done using lm:
data <- na_omit(get_vars(wlddev, c("iso3c","year","PCGDP","LIFEEX")))
full <- lm(PCGDP ~ LIFEEX + iso3c + qF(year), data)
rest <- lm(PCGDP ~ LIFEEX + iso3c, data)
anova(rest, full)
```

fgrowth *Fast Growth Rates for Time-Series and Panel Data*

Description

fgrowth is a S3 generic to compute (sequences of) suitably lagged / leaded and iterated growth rates, obtained with via the exact method of computation of through log differencing. By default growth rates are provided in percentage terms, but any scale factor can be applied. The growth operator G is a parsimonious wrapper around fgrowth. Apart from being more parsimonious, it provides a bit more flexibility than fgrowth when applied to data frames.

Usage

```
fgrowth(x, n = 1, diff = 1, ...)
      G(x, n = 1, diff = 1, ...)

## Default S3 method:
fgrowth(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA,
        logdiff = FALSE, scale = 100, stubs = TRUE, ...)
## Default S3 method:
G(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA,
  logdiff = FALSE, scale = 100, stubs = TRUE, ...)

## S3 method for class 'matrix'
fgrowth(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA,
        logdiff = FALSE, scale = 100, stubs = TRUE, ...)
## S3 method for class 'matrix'
G(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA,
  logdiff = FALSE, scale = 100, stubs = TRUE, ...)

## S3 method for class 'data.frame'
fgrowth(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA,
        logdiff = FALSE, scale = 100, stubs = TRUE, ...)
## S3 method for class 'data.frame'
G(x, n = 1, diff = 1, by = NULL, t = NULL, cols = is.numeric,
  fill = NA, logdiff = FALSE, scale = 100, stubs = TRUE, keep.ids = TRUE, ...)

# Methods for compatibility with plm:

## S3 method for class 'pseries'
fgrowth(x, n = 1, diff = 1, fill = NA, logdiff = FALSE, scale = 100, stubs = TRUE, ...)
## S3 method for class 'pseries'
G(x, n = 1, diff = 1, fill = NA, logdiff = FALSE, scale = 100, stubs = TRUE, ...)

## S3 method for class 'pdata.frame'
fgrowth(x, n = 1, diff = 1, fill = NA, logdiff = FALSE, scale = 100, stubs = TRUE, ...)
## S3 method for class 'pdata.frame'
```

```
G(x, n = 1, diff = 1, cols = is.numeric, fill = NA,
  logdiff = FALSE, scale = 100, stubs = TRUE, keep.ids = TRUE, ...)

# Methods for compatibility with dplyr:

## S3 method for class 'grouped_df'
fgrowth(x, n = 1, diff = 1, t = NULL, fill = NA, logdiff = FALSE, scale = 100,
        stubs = TRUE, keep.ids = TRUE, ...)
## S3 method for class 'grouped_df'
G(x, n = 1, diff = 1, t = NULL, fill = NA, logdiff = FALSE, scale = 100,
  stubs = TRUE, keep.ids = TRUE, ...)
```

Arguments

x	a numeric vector, matrix, data.frame, panel-series (plm::pseries), panel-data.frame (plm::pdata.frame) or grouped tibble (dplyr::grouped_df).
n	a integer vector indicating the number of lags or leads.
diff	a vector of integers > 1 indicating the order of taking growth rates.
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
by	*data.frame method*: Same as g, but also allows one- or two-sided formulas i.e. ~ group1 or var1 + var2 ~ group1 + group2. See Examples.
t	same input as g, to indicate the time-variable. For safe computation of growth rates on unordered time-series and panels. *Notes*: data.frame method also allows name, index or one-sided formula i.e. ~time. grouped_df method also allows lazy-evaluation i.e. time (no quotes).
cols	*data.frame method*: Select columns to compute growth rates using a function, column names or indices. Default: All numeric variables. *Note*: cols is ignored if a two-sided formula is passed to by.
fill	value to insert when vectors are shifted. Default is NA.
logdiff	logical. Compute log-differences instead of exact growth rates. See Details.
scale	logical. Scale factor post-applied to growth rates, default is 100 which gives growth rates in percentage terms. See Details.
stubs	logical. TRUE will rename all computed columns by adding a prefix "LnGdiff." / "FnGdiff.", or "LnDlogdiff." / "FnDlogdiff." if logdiff = TRUE.
keep.ids	*data.frame / pdata.frame / grouped_df methods*: Logical. Drop all panel-identifiers from the output (which includes all variables passed to by or t). *Note*: For panel-data.frame's and grouped tibbles identifiers are dropped, but the 'index' / 'groups' attributes are kept.
...	arguments to be passed to or from other methods.

Details

fgrowth/G by default computes exact growth rates using repeat(diff) (x[i] -x[i-n])/x[i-n]*scale, and, if 'logdiff = TRUE' approximate growth rates using repeat(diff) (log(x[i]) -log(x[i-n]))*scale. So for diff > 1 it computes growth rate of growth rates etc.. For further details see the help pages for fdiff and flag.

Value

x where the growth rate was taken diff times using lags n of itself, scaled by scale. Computations can be grouped by g/by and/or ordered by t. See Details and Examples.

See Also

flag/L/F, fdiff/D/Dlog, Time-Series and Panel-Series, Collapse Overview

Examples

```
## Simple Time-Series: AirPassengers
G(AirPassengers)                      # growth rate, same as fgrowth(AirPassengers)
G(AirPassengers, logdiff = TRUE)      # log-difference
G(AirPassengers,1,2)                  # growth rate of growth rate
G(AirPassengers,12)                   # seasonal growth rate (data is monthly)

G(AirPassengers,-2:2,1:3)           # sequence of leaded/lagged and iterated growth rates

# let's do some visual analysis
plot(G(AirPassengers,c(0,1,12)))
plot(stl(window(G(AirPassengers,12),  # Taking seasonal growth rate removes most seasonal variation
                1950), "periodic"))


## Time-Series Matrix of 4 EU Stock Market Indicators, recorded 260 days per year
plot(G(EuStockMarkets,c(0,260)))                    # Plot series and annual growth rates
summary(lm(L260G1.DAX ~., G(EuStockMarkets,260)))  # Annual growth rate of DAX regressed on the
                                                    # growth rates of the other indicators

## World Development Panel Data
head(fgrowth(num_vars(wlddev), 1, 1,              # Computes growth rates of numeric variables
          wlddev$country, wlddev$year))           # fgrowth requires externall inputs...
head(G(wlddev, 1, 1, ~country, ~year))            # Growth of numeric variables, id's attached
head(G(wlddev, 1, 1, ~country))                   # Without t: Works because data is ordered
head(G(wlddev, 1, 1, PCGDP + LIFEEX ~ country, ~year))  # Growth of GDP per Capita & Life Expectancy
head(G(wlddev, 0:1, 1, ~ country, ~year, cols = 9:10))  # Same, also retaining original series
head(G(wlddev, 0:1, 1, ~ country, ~year, 9:10,          # Dropping id columns
       keep.ids = FALSE))

# Dynamic Panel-Data Models:
summary(lm(G(PCGDP,1,1,iso3c,year) ~              # GDP growth regressed on it's lagged level
          L(PCGDP,1,iso3c,year) +                 # and the growth rate of Life Expanctancy
             G(LIFEEX,1,1,iso3c,year), data = wlddev))

g = qF(wlddev$country)                           # Omitting t and precomputing g allows for a
summary(lm(G(PCGDP,1,1,g) ~ L(PCGDP,1,g) +          # bit more parsimonious specification
                            G(LIFEEX,1,1,g), wlddev))

summary(lm(G1.PCGDP ~.,                             # Now adding level and lagged level of
L(G(wlddev,0:1,1, ~ country, ~year,9:10),0:1,           # LIFEEX and lagged growth rates
  ~ country, ~year, keep.ids = FALSE)[-1]))
```

```
## Using plm can make things easier, but avoid attaching or 'with' calls:
pwlddev <- plm::pdata.frame(wlddev, index = c("country","year"))
head(G(pwlddev, 0:1, 1, 9:10))                    # Again growth rates of LIFEEX and PCGDP
PCGDP <- pwlddev$PCGDP                                 # A panel-Series of GDP per Capita
G(PCGDP)                                               # Growth rate of the panel series.
summary(lm(G1.PCGDP ~.,                               # Running the dynamic model again ->
           data = L(G(pwlddev,0:1,1,9:10),0:1,           # code becomes a bit simpler
                    keep.ids = FALSE)[-1]))

# One could be tempted to also do something like this, but THIS DOES NOT WORK!!!:
# lm drops the attributes (-> with(pwlddev, PCGDP) drops attr. so G.default and L.matrix are used)
summary(lm(G(PCGDP) ~ L(G(PCGDP,0:1)) + L(G(LIFEEX,0:1),0:1), pwlddev))

# To make it work, one needs to create pseries (note: attach(pwlddev) also won't work)
LIFEEX <- pwlddev$LIFEEX
summary(lm(G(PCGDP) ~ L(G(PCGDP,0:1)) + L(G(LIFEEX,0:1),0:1))) # THIS WORKS !!

## Using dplyr:
library(dplyr)
wlddev %>% group_by(country) %>%
             select(PCGDP,LIFEEX) %>% fgrowth(0:1)         # Adding growth rates
wlddev %>% group_by(country) %>%
             select(year,PCGDP,LIFEEX) %>%
             fgrowth(0:1, t = year)                          # Also using t (safer)
```

`fHDbetween, fHDwithin` *Higher-Dimensional Centering and Linear Prediction*

Description

`fHDbetween` is a generalization of `fbetween` to efficiently predict with multiple factors and linear models (i.e. predict with vectors/factors, matrices, or data.frames/lists where the latter may contain multiple factor variables). Similarly `fHDwithin` is a generalization of `fwithin` to center on multiple factors and partial-out linear models.

The corresponding operators `HDB` and `HDW` also exist and additionally allow to predict / partial out full `lm()` formulas with interactions between variables.

Usage

```
fHDbetween(x, ...)
 fHDwithin(x, ...)
       HDB(x, ...)
       HDW(x, ...)

## Default S3 method:
fHDbetween(x, fl, w = NULL, na.rm = TRUE, fill = FALSE, ...)
## Default S3 method:
fHDwithin(x, fl, w = NULL, na.rm = TRUE, fill = FALSE, ...)
```

```
## Default S3 method:
HDB(x, fl, w = NULL, na.rm = TRUE, fill = FALSE, ...)
## Default S3 method:
HDW(x, fl, w = NULL, na.rm = TRUE, fill = FALSE, ...)

## S3 method for class 'matrix'
fHDbetween(x, fl, w = NULL, na.rm = TRUE, fill = FALSE, ...)
## S3 method for class 'matrix'
fHDwithin(x, fl, w = NULL, na.rm = TRUE, fill = FALSE, ...)
## S3 method for class 'matrix'
HDB(x, fl, w = NULL, na.rm = TRUE, fill = FALSE, stub = "HDB.", ...)
## S3 method for class 'matrix'
HDW(x, fl, w = NULL, na.rm = TRUE, fill = FALSE, stub = "HDW.", ...)

## S3 method for class 'data.frame'
fHDbetween(x, fl, w = NULL, na.rm = TRUE, fill = FALSE,
           variable.wise = FALSE, ...)
## S3 method for class 'data.frame'
fHDwithin(x, fl, w = NULL, na.rm = TRUE, fill = FALSE,
          variable.wise = FALSE, ...)
## S3 method for class 'data.frame'
HDB(x, fl, w = NULL, cols = is.numeric, na.rm = TRUE, fill = FALSE,
    variable.wise = FALSE, stub = "HDB.", ...)
## S3 method for class 'data.frame'
HDW(x, fl, w = NULL, cols = is.numeric, na.rm = TRUE, fill = FALSE,
    variable.wise = FALSE, stub = "HDW.", ...)

# Methods for compatibility with plm:

## S3 method for class 'pseries'
fHDbetween(x, w = NULL, na.rm = TRUE, fill = TRUE, ...)
## S3 method for class 'pseries'
fHDwithin(x, w = NULL, na.rm = TRUE, fill = TRUE, ...)
## S3 method for class 'pseries'
HDB(x, w = NULL, na.rm = TRUE, fill = TRUE, ...)
## S3 method for class 'pseries'
HDW(x, w = NULL, na.rm = TRUE, fill = TRUE, ...)

## S3 method for class 'pdata.frame'
fHDbetween(x, w = NULL, na.rm = TRUE, fill = TRUE,
           variable.wise = TRUE, ...)
## S3 method for class 'pdata.frame'
fHDwithin(x, w = NULL, na.rm = TRUE, fill = TRUE,
          variable.wise = TRUE, ...)
## S3 method for class 'pdata.frame'
HDB(x, w = NULL, cols = is.numeric, na.rm = TRUE, fill = TRUE,
    variable.wise = TRUE, stub = "HDB.", ...)
## S3 method for class 'pdata.frame'
```

```
HDW(x, w = NULL, cols = is.numeric, na.rm = TRUE, fill = TRUE,
    variable.wise = TRUE, stub = "HDW.", ...)
```

Arguments

x	a numeric vector, matrix, data.frame, panel-series (plm::pseries) or panel-data.frame (plm::pdata.frame).
fl	a numeric vector, factor, matrix, data.frame or list (which may or may not contain factors). In the data.frame method fl can also be a one-or two sided lm() formula with variables contained in x. Interactions (:) and full interactions (*) are supported! See Examples.
w	a vector of (non-negative) weights. Currently only weighted centering on multiple factors is supported, not weighted linear models.
cols	*data.frame methods*: Select columns to center (partial-out) or predict using column names, indices or a function. Unless specified otherwise all numeric columns are selected. If NULL, all variables are selected.
na.rm	remove missing values from both x and fl. by default rows with missing values in x or fl are removed. In that case an attribute "na.rm" is attached containing the rows removed.
fill	If na.rm = TRUE, fill = TRUE will not remove rows with missing values in x or fl, but fill them with NA's.
variable.wise	*data.frame methods*: Setting variable.wise = TRUE will process each column individually i.e. use all non-missing cases in each column and in fl (fl is only checked for missing values if na.rm = TRUE). This is a lot less efficient but uses all data available in each column.
stub	a prefix / stub to rename all transformed columns. FALSE will not rename columns.
...	further arguments passed to lfe::demeanlist (if fl contains factors), or to / from other methods.

Details

fHDbetween/HDB and fHDwithin/HDW can be understood as generalizations of lfe::demeanlist to continuous-data and formula input, and more choices dealing with missing values. They are powerful tools for complex high-dimensional linear prediction problems involving large factors and datasets, but can just as well handle ordinary regression problems. Intended areas of use are to efficiently obtain residuals and predicted values from data, and to prepare data for complex linear models involving multiple levels of fixed effects. Such models can now be fitted using lm() on data prepared with fHDwithin / HDW (relying on bootstrapped SE's for inference, or implementing the appropriate corrections). See Examples.

If fl is a vector or matrix, the result are identical to lm i.e. fHDbetween / HDB returns fitted(lm(x ~ fl)) and fHDwithin / HDW residuals(lm(x ~ fl)). If fl is a list containing factors, all variables in x and non-factor variables in fl are centered on these factors using the method of alternating projections implemented by lfe::demeanlist. Afterwards the centered data is regressed on the centered predictors. If fl is just a list of factors, fHDwithin/HDW returns the centered data and fHDbetween/HDB the corresponding means. Take as a most general example a list fl

= list(fct1,fct2,...,var1,var2,...) where fcti are factors and vari are continuous variables. The output of fHDwithin/HDW | fHDbetween/HDB will then be identical to calling resid | fitted on lm(x ~ fct1 + fct2 + ... + var1 + var2 + ...). The computations performed by fHDwithin/HDW and fHDbetween/HDB are however much faster and more memory efficient than lm because factors are not passed to stats::model.matrix and expanded to matrices of dummies but projected out using lfe::demeanlist.

The formula interface to the data.frame method (only supported by the operators HDW | HDB) provides ease of use and allows for additional modelling complexity. For example it is possible to project out formulas like HDW(data,~ fct1*var1 + fct2:fct3 + var2:fct2:fct3 + var1:var2:var3 + poly(var5,3)*fct5) containing simple (:) or full (*) interactions of factors with continuous variables or polynomials of continuous variables, and two-or three-way interactions of factors and continuous variables. If the formula is one-sided as in the example above (the space left of (~) is left empty), the formula is applied to all variables selected through cols. The specification provided in cols (default: all numeric variables not used in the formula) can be overridden by supplying one-or more dependent variables. For example HDW(data,var1 + var2 ~ fct1 + fct2) will return a data.frame with var1 and var2 centered on fct1 and fct2.

The special methods for plm::pseries and plm::pdata.frame center a panel-series or variables in a panel-data.frame on all panel-identifiers. By default in these methods fill = TRUE and variable.wise = TRUE, so missing values are kept. This change in the default arguments was done to ensure a coherent framework of functions and operators applied to *plm* panel-data classes.

Value

HDB returns fitted values of regressing x on fl. HDW returns residuals. See Details and Examples.

Note

Weights are currently only supported for centering / averaging, not for linear regression.

Caution with full (*) and factor-continuous variable interactions:: In general full interactions specified with (*) can be very slow on large data, and lfe::demeanlist is also not very speedy on interaction between factors and continuous variables, so these structures should be used with caution (don't just specify an interaction like that on a large dataset, start with smaller data and see how long computations take. Upon further updates of lfe::demeanlist, performance might improve).

On the differences between fHDwithin/HDW... and fwithin/W...::

- fHDwithin/HDW can center data on multiple factors and also partial out continuous variables while fwithin/W only centers on one factor, but does that very efficiently...
- HDW(data,~ qF(group1) + qF(group2)) simultaneously centers numeric variables in data on group1 and group2, while W(data,~ group1 + group2) centers data on the interaction of group1 and group2. The equivalent operation in HDW would be: HDW(data,~ qF(group1):qF(group2)).
- W always does computations on the variable-wise complete observations (in both matrices and data.frames), whereas by default HDW removes all cases missing in either x or fl. In short, W(data,~ group1 + group2) is actually equivalent to HDW(data,~ qF(group1):qF(group2),variable.wise = TRUE). HDW(data,~ qF(group1):qF(group2)) would remove any missing cases.
- fbetween/B and fwithin/W have options to fill missing cases using group-averages and to add the overall mean back to group-demeaned data. These options are not available in

fHDbetween/HDB and fHDwithin/HDW. Since HDB and HDW by default remove missing cases, they also don't have options to keep grouping-columns as in B and W.

See Also

fbetween/B and fwithin/W, fscale/STD, TRA, fFtest, Data Transformations, Collapse Overview

Examples

```
HDW(mtcars$mpg, mtcars$carb)                    # Simple regression problems..
HDW(mtcars$mpg, mtcars[-1])
HDW(mtcars$mpg, qM(mtcars[-1]))
HDW(qM(mtcars[3:4]), mtcars[1:2])
HDW(iris[1:2], iris[3:4])                # Partialling columns 3 and 4 out of colums 1 and 2
HDW(iris[1:2], iris[3:5])                     # Adding the Species factor -> fixed effect
HDW(wlddev, PCGDP + LIFEEX ~ iso3c + qF(year)) # Partialling out 2 fixed effects (iso3c is factor)
HDW(wlddev, PCGDP + LIFEEX ~ iso3c + qF(year), variable.wise = TRUE) # Variable-wise computations
HDW(wlddev, PCGDP + LIFEEX ~ iso3c + qF(year) + ODA) # Adding ODA as a continuouus regressor
HDW(wlddev, PCGDP + LIFEEX ~ iso3c:qF(decade) + qF(year) + ODA) # Country-decade and year FE's

# More complex examples (Currently only recommended for smaller data)
lm(HDW.mpg ~ HDW.hp, data = HDW(mtcars, ~ factor(cyl)*carb + vs + wt:gear + wt:gear:carb))
lm(mpg ~ hp + factor(cyl)*carb + vs + wt:gear + wt:gear:carb, data = mtcars)

lm(HDW.mpg ~ HDW.hp, data = HDW(mtcars, ~ factor(cyl)*carb + vs + wt:gear))
lm(mpg ~ hp + factor(cyl)*carb + vs + wt:gear, data = mtcars)

lm(HDW.mpg ~ HDW.hp, data = HDW(mtcars, ~ cyl*carb + vs + wt:gear))
lm(mpg ~ hp + cyl*carb + vs + wt:gear, data = mtcars)

lm(HDW.mpg ~ HDW.hp, data = HDW(mtcars, mpg + hp ~ cyl*carb + factor(cyl)*poly(drat,2)))
lm(mpg ~ hp + cyl*carb + factor(cyl)*poly(drat,2), data = mtcars)
```

---

flag *Fast Lags and Leads for Time-Series and Panel Data*

Description

flag is an S3 generic to compute (sequences of) lags and leads. L and F are wrappers around flag representing the lag- and lead-operators, such that L(x,-1) = F(x,1) = F(x) and L(x,-3:3) = F(x,3:-3). L and F provide more flexibility than flag when applied to data frames (i.e. column subsetting, formula input and id-variable-preservation capabilities...), but are otherwise identical.

(flag is more of a programmers function in style of the Fast Statistical Functions while L and F are more practical to use in regression formulas or for computations on data frames.)

Usage

```
flag(x, n = 1, ...)
   L(x, n = 1, ...)
   F(x, n = 1, ...)

## Default S3 method:
flag(x, n = 1, g = NULL, t = NULL, fill = NA, stubs = TRUE, ...)
## Default S3 method:
L(x, n = 1, g = NULL, t = NULL, fill = NA, stubs = TRUE, ...)
## Default S3 method:
F(x, n = 1, g = NULL, t = NULL, fill = NA, stubs = TRUE, ...)

## S3 method for class 'matrix'
flag(x, n = 1, g = NULL, t = NULL, fill = NA, stubs = TRUE, ...)
## S3 method for class 'matrix'
L(x, n = 1, g = NULL, t = NULL, fill = NA, stubs = TRUE, ...)
## S3 method for class 'matrix'
F(x, n = 1, g = NULL, t = NULL, fill = NA, stubs = TRUE, ...)

## S3 method for class 'data.frame'
flag(x, n = 1, g = NULL, t = NULL, fill = NA, stubs = TRUE, ...)
## S3 method for class 'data.frame'
L(x, n = 1, by = NULL, t = NULL, cols = is.numeric,
  fill = NA, stubs = TRUE, keep.ids = TRUE, ...)
## S3 method for class 'data.frame'
F(x, n = 1, by = NULL, t = NULL, cols = is.numeric,
  fill = NA, stubs = TRUE, keep.ids = TRUE, ...)

# Methods for compatibility with plm:

## S3 method for class 'pseries'
flag(x, n = 1, fill = NA, stubs = TRUE, ...)
## S3 method for class 'pseries'
L(x, n = 1, fill = NA, stubs = TRUE, ...)
## S3 method for class 'pseries'
F(x, n = 1, fill = NA, stubs = TRUE, ...)

## S3 method for class 'pdata.frame'
flag(x, n = 1, fill = NA, stubs = TRUE, ...)
## S3 method for class 'pdata.frame'
L(x, n = 1, cols = is.numeric, fill = NA, stubs = TRUE,
  keep.ids = TRUE, ...)
## S3 method for class 'pdata.frame'
F(x, n = 1, cols = is.numeric, fill = NA, stubs = TRUE,
  keep.ids = TRUE, ...)

# Methods for compatibility with dplyr:
```

```
## S3 method for class 'grouped_df'
flag(x, n = 1, t = NULL, fill = NA, stubs = TRUE, keep.ids = TRUE, ...)
## S3 method for class 'grouped_df'
L(x, n = 1, t = NULL, fill = NA, stubs = TRUE, keep.ids = TRUE, ...)
## S3 method for class 'grouped_df'
F(x, n = 1, t = NULL, fill = NA, stubs = TRUE, keep.ids = TRUE, ...)
```

Arguments

x	a vector, matrix, data.frame, panel-series (plm::pseries), panel-data.frame (plm::pdata.frame) or grouped tibble (dplyr::grouped_df). Data must not be numeric.
n	an integer vector indicating the lags/leads to compute.
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
by	*data.frame method*: Same as g, but also allows one- or two-sided formulas i.e. ~ group1 or var1 + var2 ~ group1 + group2. See Examples.
t	same input as g, to indicate the time-variable. For safe computation of lags/leads on unordered time-series and panels. *Note*: Data frame method also allows one-sided formula i.e. ~time, and grouped_df method also allows lazy-evaluation i.e. time (no quotes).
cols	*data.frame method*: Select columns to lag/lead using a function, column names or indices. Default: All numeric variables. *Note*: cols is ignored if a two-sided formula is passed to by.
fill	value to insert when vectors are shifted. Default is NA.
stubs	logical. TRUE will rename all lagged / leaded columns by adding a stub or prefix "Ln." / "Fn.".
keep.ids	*data.frame / pdata.frame / grouped_df methods*: Logical. Drop all panel-identifiers from the output (which includes all variables passed to by or t). *Note*: For panel-data.frame's and grouped tibbles identifiers are dropped, but the 'index' / 'groups' attributes are kept.
...	arguments to be passed to or from other methods.

Details

If a single integer is passed to n, and g/by and t are left empty, flag/L/F just returns x with all columns lagged / leaded by n. If length(n)>1, and x is an atomic vector, flag/L/F returns a matrix with lags / leads computed in the same order as passed to n. If instead x is a matrix / data.frame, a matrix / data.frame with ncol(x)*length(n) columns is returned where columns are sorted first by variable and then by lag (so all lags computed on a variable are grouped together). x can be of any standard data type.

With groups/panel-identifiers supplied to g/by, flag/L/F efficiently computes a panel-lag by shifting the entire vector(s) but inserting fill elements in the right places. If t is left empty, the data needs to be ordered such that all values belonging to a group are consecutive and in the right order. It is not necessary that the groups themselves occur in the right order. If a time-variable is supplied to t (or a list of time-variables uniquely identifying the time-dimension), the panel is fully identified and lags / leads can be securely computed even if the data is completely unordered.

flag/L/F supports balanced panels and unbalanced panels where various individuals are observed for different time-sequences (both start, end and duration of observation can differ for each individual). flag/L/F does not natively support irregularly spaced time-series and panels, that is situations where there are either gaps in time and/or repeated observations in the same time-period for some individual (see also computational details below). For such cases the function seqid can ge used to generate an appropriate panel-identifier (i.e. splitting individuals with an irregular time-sequence into multiple individuals with regular time-sequences before applying flag/L/F).

It is also possible to compute lags / leads on unordered time-series (thus utilizing t but leaving g/by empty), although this is probably more rare to encounter than unordered panels. Irregularly spaced time-series can also be lagged using a panel- identifier generated with seqid.

Computationally, if both g/by and t are supplied, flag/L/F uses two initial passes to create an ordering through which the data are accessed. First-pass: Calculate group sizes and the minimum time-value for each individual. Second-pass: Generate the ordering by placing the current element index into the vector slot obtained by adding the cumulative group size and the current time-value subtracted its individual-minimum together. This method of computation is faster than any sort-based method and delivers optimal performance if the panel-id supplied to g/by is already a factor variable, and if t is either an integer or factor variable. If g/by is not factor or t is not factor or integer, qG or GRP will be called to group the respective identifier and this can be expensive, so for optimal performance prepare the data (or use *plm* classes). A caveat of not using sort-based methods is that gaps or repeated values in time are only recognized towards the end of the second pass where they cannot be rectified anymore, and thus flag/L/F does not natively support irregular panels but throws an error.

The methods applying to *plm* objects (panel-series and panel-data.frames) automatically utilize the factor panel-identifiers attached to these objects and thus securely and efficiently compute fully identified panel-lags. If these objects have > 2 panel-identifiers attached to them, the last identifier is assumed to be the time-variable, and the others are taken as grouping-variables and interacted. I note that flag/L/F is significantly faster than plm::lag/plm::lead since the latter is written in R and based on a Split-Apply-Combine logic.

Value

x lagged / leaded n-times, grouped by g/by, ordered by t. See Details and Examples.

See Also

fdiff/D/Dlog, fgrowth/G, Time-Series and Panel-Series, Collapse Overview

Examples

```
## Simple Time-Series: AirPassengers
L(AirPassengers)                    # 1 lag
F(AirPassengers)                    # 1 lead

all_identical(L(AirPassengers),     # 3 identical ways of computing 1 lag
              flag(AirPassengers),
              F(AirPassengers,-1))

L(AirPassengers,-1:3)               # 1 lead and 3 lags - output as matrix
```

```
## Time-Series Matrix of 4 EU Stock Market Indicators, 1991-1998
tsp(EuStockMarkets)                                  # Data is recorded on 260 days per year
freq <- frequency(EuStockMarkets)
plot(stl(EuStockMarkets[,"DAX"], freq))                 # There is some obvious seasonality
L(EuStockMarkets,-1:3*freq)                              # 1 annual lead and 3 annual lags
summary(lm(DAX ~., data = L(EuStockMarkets,-1:3*freq))) # DAX regressed on it's own annual lead,
                                                # lags and the lead/lags of the other series

## World Development Panel Data
head(flag(wlddev, 1, wlddev$iso3c, wlddev$year))        # This lags all variables,
head(L(wlddev, 1, ~iso3c, ~year))                       # This lags all numeric variables
head(L(wlddev, 1, ~iso3c))                           # Without t: Works because data is ordered
head(L(wlddev, 1, PCGDP + LIFEEX ~ iso3c, ~year))     # This lags GDP per Capita & Life Expectancy
head(L(wlddev, 0:2, ~ iso3c, ~year, cols = 9:10))     # Same, also retaining original series
head(L(wlddev, 1:2, PCGDP + LIFEEX ~ iso3c, ~year,      # Two lags, dropping id columns
       keep.ids = FALSE))

# Different ways of regressing GDP on its's lags and life-Expectancy and it's lags
summary(lm(PCGDP ~ ., L(wlddev, 0:2, ~iso3c, ~year, 9:10, keep.ids = FALSE)))    # 1 - Precomputing
summary(lm(PCGDP ~ L(PCGDP,1:2,iso3c,year) + L(LIFEEX,0:2,iso3c,year), wlddev))  # 2 - Ad-hoc
summary(lm(PCGDP ~ L(PCGDP,1:2,iso3c) + L(LIFEEX,0:2,iso3c), wlddev))         # 3 - same no year
g = qF(wlddev$iso3c); t = qF(wlddev$year)                                  # 4- Precomputing
summary(lm(PCGDP ~ L(PCGDP,1:2,g,t) + L(LIFEEX,0:2,g,t), wlddev))             # panel-id's

## Using plm:
pwlddev <- plm::pdata.frame(wlddev, index = c("iso3c","year"))
head(L(pwlddev, 0:2, 9:10))                               # Again 2 lags of GDP and LIFEEX
PCGDP <- pwlddev$PCGDP                                   # A panel-Series of GDP per Capita
L(PCGDP)                                                        # Lagging the panel series
summary(lm(PCGDP ~ ., L(pwlddev, 0:2, 9:10, keep.ids = FALSE))) # Running the lm again: WORKS!
# THIS DOES NOT WORK: Unfortunately lm drops the attributes of the columns,
# so L.default is used here and ordinary lags are computed. (with and attach don't retain attr.)
summary(lm(PCGDP ~ L(PCGDP,1:2) + L(LIFEEX,0:2), pwlddev))
LIFEEX <- pwlddev$LIFEEX                                 # To make it work, create pseries
summary(lm(PCGDP ~ L(PCGDP,1:2) + L(LIFEEX,0:2)))               # THIS WORKS !!

## Using dplyr:
library(dplyr)
wlddev %>% group_by(iso3c) %>% select(PCGDP,LIFEEX) %>% L(0:2)
wlddev %>% group_by(iso3c) %>% select(year,PCGDP,LIFEEX) %>% L(0:2,year) # Also using t (safer)
```

---

fmean *Fast (Grouped, Weighted) Mean for Matrix-Like Objects*

Description

fmean is a generic function that computes the (column-wise) mean of x, (optionally) grouped by g and/or weighted by w. The TRA argument can further be used to transform x using its (grouped, weighted) mean.

Usage

```
fmean(x, ...)

## Default S3 method:
fmean(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = TRUE, ...)

## S3 method for class 'matrix'
fmean(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'data.frame'
fmean(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'grouped_df'
fmean(x, w = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = FALSE, keep.group_vars = TRUE, keep.w = TRUE, ...)
```

Arguments

x	a numeric vector, matrix, data.frame or grouped tibble (dplyr::grouped_df).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
w	a numeric vector of (non-negative) weights, may contain missing values.
TRA	an integer or quoted operator indicating the transformation to perform: 1 - "replace_fill" | 2 - "replace" | 3 - "-" | 4 - "-+" | 5 - "/" | 6 - "%" | 7 - "+" | 8 - "*" | 9 - "%%" | 10 - "-%%". See TRA.
na.rm	logical. Skip missing values in x. Defaults to TRUE and implemented at very little computational cost. If na.rm = FALSE a NA is returned when encountered.
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.frame method). No row-names are generated for data.tables and (default) grouped tibbles.
drop	*matrix and data.frame method:* drop dimensions and return an atomic vector if g = NULL and TRA = NULL.
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation.
keep.w	*grouped_df method:* Logical. Retain summed weighting variable after computation (if contained in grouped_df).
...	arguments to be passed to or from other methods.

Details

Missing-value removal as controlled by the na.rm argument is done very efficiently by simply skipping them in the computation (thus setting na.rm = FALSE on data with no missing values doesn't

give extra speed). Large performance gains can nevertheless be achieved in the presence of missing values if na.rm = FALSE, since then the corresponding computation is terminated once a NA is encountered and NA is returned (unlike base::mean which just runs through without any checks).

The weighted mean is computed as sum(x * w) / sum(w). If na.rm = TRUE, missing values will be removed from both x and w i.e. utilizing only x[complete.cases(x,w)] and w[complete.cases(x,w)].

This all seamlessly generalizes to grouped computations, which are performed in a single pass (without splitting the data) and therefore extremely fast.

When applied to data frame's with groups or drop = FALSE, fmean preserves all column attributes (such as variable labels) but does not distinguish between classed and unclassed object (thus applying fmean to a factor column will give a 'malformed factor' error). The attributes of the data frame itself are also preserved.

Value

The (w weighted) mean of x, grouped by g, or (if TRA is used) x transformed by its mean, grouped by g.

See Also

fmedian, fmode, Fast Statistical Functions, Collapse Overview

Examples

```
## default vector method
mpg <- mtcars$mpg
fmean(mpg)                          # Simple mean
fmean(mpg, w = mtcars$hp)           # Weighted mean: Weighted by hp
fmean(mpg, TRA = "-")               # Simple transformation: demeaning (See also ?W)
fmean(mpg, mtcars$cyl)              # Grouped mean
fmean(mpg, mtcars[8:9])             # another grouped mean.
g <- GRP(mtcars[c(2,8:9)])
fmean(mpg, g)                       # Pre-computing groups speeds up the computation
fmean(mpg, g, mtcars$hp)            # Grouped weighted mean
fmean(mpg, g, TRA = "-")            # Demeaning by group
fmean(mpg, g, mtcars$hp, "-")       # Group-demeaning using weighted group means

## data.frame method
fmean(mtcars)
fmean(mtcars, g)
fmean(fgroup_by(mtcars, cyl, vs, am))  # another way of doing it...
fmean(mtcars, g, TRA = "-") # etc...


## matrix method
m <- qM(mtcars)
fmean(m)
fmean(m, g)
fmean(m, g, TRA = "-") # etc...

## method for grouped tibbles - for use with dplyr
```

```
library(dplyr)
mtcars %>% group_by(cyl,vs,am) %>% fmean           # Ordinary
mtcars %>% group_by(cyl,vs,am) %>% fmean(hp)       # Weighted
mtcars %>% group_by(cyl,vs,am) %>% fmean(hp,"-")   # Weighted Transform
mtcars %>% group_by(cyl,vs,am) %>%
           select(mpg,hp) %>% fmean(hp,"-")        # Only mpg

mtcars %>% fgroup_by(cyl,vs,am) %>%              # Equivalent but faster !!
           fselect(mpg,hp) %>% fmean(hp,"-")
```

`fmedian` *Fast (Grouped) Median Value for Matrix-Like Objects*

Description

`fmedian` is a generic function that computes the (column-wise) median value of all values in `x`, (optionally) grouped by `g`. The `TRA` argument can further be used to transform `x` using its (grouped) median value.

Usage

```
fmedian(x, ...)

## Default S3 method:
fmedian(x, g = NULL, TRA = NULL, na.rm = TRUE,
        use.g.names = TRUE, ...)

## S3 method for class 'matrix'
fmedian(x, g = NULL, TRA = NULL, na.rm = TRUE,
        use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'data.frame'
fmedian(x, g = NULL, TRA = NULL, na.rm = TRUE,
        use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'grouped_df'
fmedian(x, TRA = NULL, na.rm = TRUE,
        use.g.names = FALSE, keep.group_vars = TRUE, ...)
```

Arguments

`x`	a numeric vector, matrix, data.frame or grouped tibble (`dplyr::grouped_df`).
`g`	a factor, `GRP` object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a `GRP` object) used to group `x`.
`TRA`	an integer or quoted operator indicating the transformation to perform: 1 - "replace_fill" | 2 - "replace" | 3 - "-" | 4 - "-+" | 5 - "/" | 6 - "%" | 7 - "+" | 8 - "*" | 9 - "%%" | 10 - "-%%". See `TRA`.

na.rm	logical. Skip missing values in x. Defaults to TRUE and implemented at very little computational cost. If na.rm = FALSE a NA is returned when encountered.
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.frame method). No row-names are generated for data.tables and (default) grouped tibbles.
drop	*matrix and data.frame method:* drop dimensions and return an atomic vector if g = NULL and TRA = NULL.
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation.
...	arguments to be passed to or from other methods.

Details

Median value estimation is done using std::nth_element in C++, which is an efficient partial sorting algorithm. A downside of this is that vectors need to be copied first and then partially sorted, thus fmedian currently requires additional memory equal to the size of the object (x).

Grouped computations are currently performed by mapping the data to a sparse-array directed by g and then partially sorting each row (group) of that array. For reasons I don't fully understand this requires less memory than a full deep copy which is done with no groups.

When applied to data frame's with groups or drop = FALSE, fmedian preserves all column attributes (such as variable labels) but does not distinguish between classed and unclassed objects. The attributes of the data frame itself are also preserved.

Value

The median value of x, grouped by g, or (if TRA is used) x transformed by its median value, grouped by g.

See Also

fmean, fmode, Fast Statistical Functions, Collapse Overview

Examples

```
## default vector method
mpg <- mtcars$mpg
fmedian(mpg)                          # Simple median value
fmedian(mpg, TRA = "-")               # Simple transformation: Subtract median value
fmedian(mpg, mtcars$cyl)              # Grouped median value
fmedian(mpg, mtcars[c(2,8:9)])        # More groups...
g <- GRP(mtcars, ~ cyl + vs + am)     # Precomputing groups gives more speed !!
fmedian(mpg, g)
fmedian(mpg, g, TRA = "-")            # Groupwise subtract median value

## data.frame method
fmedian(mtcars)
fmedian(mtcars, TRA = "-")
fmedian(mtcars, g)
```

```
fmean(fgroup_by(mtcars, cyl, vs, am))   # another way of doing it...
fmedian(mtcars, g, use.g.names = FALSE) # No row-names generated

## matrix method
m <- qM(mtcars)
fmedian(m)
fmedian(m, TRA = "-")
fmedian(m, g) # etc...

## method for grouped tibbles - for use with dplyr
library(dplyr)
mtcars %>% group_by(cyl,vs,am) %>% fmedian
mtcars %>% fgroup_by(cyl,vs,am) %>% fmedian # Faster grouping!
mtcars %>% fgroup_by(cyl,vs,am) %>% fmedian("-") # De-median
mtcars %>% fgroup_by(cyl,vs,am) %>% fselect(mpg) %>% fmedian
```

`fmin, fmax` *Fast (Grouped) Maxima and Minima for Matrix-Like Objects*

Description

`fmax` and `fmin` are generic functions that compute the (column-wise) maximum and minimum value of all values in `x`, (optionally) grouped by `g`. The `TRA` argument can further be used to transform `x` using its (grouped) maximum or minimum value.

Usage

```
fmax(x, ...)
fmin(x, ...)

## Default S3 method:
fmax(x, g = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, ...)
## Default S3 method:
fmin(x, g = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, ...)

## S3 method for class 'matrix'
fmax(x, g = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, drop = TRUE, ...)
## S3 method for class 'matrix'
fmin(x, g = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'data.frame'
fmax(x, g = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, drop = TRUE, ...)
## S3 method for class 'data.frame'
```

```
fmin(x, g = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'grouped_df'
fmax(x, TRA = NULL, na.rm = TRUE,
     use.g.names = FALSE, keep.group_vars = TRUE, ...)
## S3 method for class 'grouped_df'
fmin(x, TRA = NULL, na.rm = TRUE,
     use.g.names = FALSE, keep.group_vars = TRUE, ...)
```

Arguments

x	a numeric vector, matrix, data.frame or grouped tibble (dplyr::grouped_df).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
TRA	an integer or quoted operator indicating the transformation to perform: 1 - "replace_fill" | 2 - "replace" | 3 - "-" | 4 - "-+" | 5 - "/" | 6 - "%" | 7 - "+" | 8 - "*" | 9 - "%%" | 10 - "-%%". See TRA.
na.rm	logical. Skip missing values in x. Defaults to TRUE and implemented at very little computational cost. If na.rm = FALSE a NA is returned when encountered.
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.frame method). No row-names are generated for data.tables and grouped tibbles.
drop	*matrix and data.frame method:* drop dimensions and return an atomic vector if g = NULL and TRA = NULL.
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation.
...	arguments to be passed to or from other methods.

Details

Missing-value removal as controlled by the na.rm argument is done at no extra cost since in C++ any logical comparison involving NA or NaN evaluates to FALSE. Large performance gains can nevertheless be achieved in the presence of missing values if na.rm = FALSE, since then the corresponding computation is terminated once a NA is encountered and NA is returned (unlike base::max and base::min which just run through without any checks).

This all seamlessly generalizes to grouped computations, which are performed in a single pass (without splitting the data) and therefore extremely fast.

When applied to data frame's with groups or drop = FALSE, fmax and fmin preserve all column attributes (such as variable labels) but do not distinguish between classed and unclassed objects. The attributes of the data frame itself are also preserved.

Value

fmax returns the maximum value of x, grouped by g, or (if TRA is used) x transformed by its maximum value, grouped by g. Analogous, fmin returns the minimum value ...

See Also

Fast Statistical Functions, Collapse Overview

Examples

```
## default vector method
mpg <- mtcars$mpg
fmax(mpg)                          # maximum value
fmin(mpg)                    # minimum value (all examples below use fmax but apply to fmin)
fmax(mpg, TRA = "%")             # Simple transformation: Take percentage of maximum value
fmax(mpg, mtcars$cyl)              # Grouped maximum value
fmax(mpg, mtcars[c(2,8:9)])        # More groups...
g <- GRP(mtcars, ~ cyl + vs + am) # Precomputing groups gives more speed !!
fmax(mpg, g)
fmax(mpg, g, TRA = "%")            # Groupwise percentage of maximum value
fmax(mpg, g, TRA = "replace")      # Groupwise replace by maximum value

## data.frame method
fmax(mtcars)
fmax(mtcars, TRA = "%")
fmax(mtcars, g)
fmax(mtcars, g, use.g.names = FALSE) # No row-names generated

## matrix method
m <- qM(mtcars)
fmax(m)
fmax(m, TRA = "%")
fmax(m, g) # etc...

## method for grouped tibbles - for use with dplyr
library(dplyr)
mtcars %>% group_by(cyl,vs,am) %>% fmax
mtcars %>% group_by(cyl,vs,am) %>% fmax("%")
mtcars %>% group_by(cyl,vs,am) %>% select(mpg) %>% fmax
```

---

fmode *Fast (Grouped, Weighted) Statistical Mode for Matrix-Like Objects*

Description

fmode is a generic function and returns the (column-wise) statistical mode i.e. the most frequent value of x, (optionally) grouped by g and/or weighted by w. The TRA argument can further be used to transform x using its (grouped, weighted) mode.

Usage

```
fmode(x, ...)

## Default S3 method:
```

```
fmode(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = TRUE, ...)

## S3 method for class 'matrix'
fmode(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'data.frame'
fmode(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'grouped_df'
fmode(x, w = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = FALSE, keep.group_vars = TRUE, keep.w = TRUE, ...)
```

Arguments

x	a vector, matrix, data.frame or grouped tibble (dplyr::grouped_df).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
w	a numeric vector of (non-negative) weights, may contain missing values.
TRA	an integer or quoted operator indicating the transformation to perform: 1 - "replace_fill" | 2 - "replace" | 3 - "-" | 4 - "-+" | 5 - "/" | 6 - "%" | 7 - "+" | 8 - "*" | 9 - "%%" | 10 - "-%%". See TRA.
na.rm	logical. Skip missing values in x. Defaults to TRUE and implemented at very little computational cost. If na.rm = FALSE, NA is treated as any other value.
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.frame method). No row-names are generated for data.tables and grouped tibbles.
drop	*matrix and data.frame method:* drop dimensions and return an atomic vector if g = NULL and TRA = NULL.
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation.
keep.w	*grouped_df method:* Logical. Retain sum of weighting variable after computation (if contained in grouped_df).
...	arguments to be passed to or from other methods.

Details

fmode implements a pretty fast algorithm to find the statistical mode utilizing index- hashing implemented in the Rcpp::sugar::IndexHash class.

If all values are distinct, the first value is returned. If there are multiple distinct values having the top frequency, the first value established as having the top frequency when passing through the data from element 1 to element n is returned. If na.rm = FALSE, NA is not removed but treated as any other value (i.e. it's frequency is counted). If all values are NA, NA is always returned.

The weighted mode is computed by summing up the weights for all distinct values and choosing the value with the largest sum. If `na.rm = TRUE`, missing values will be removed from both `x` and `w` i.e. utilizing only `x[complete.cases(x,w)]` and `w[complete.cases(x,w)]`.

This all seamlessly generalizes to grouped computations, which are currently performed by mapping the data to a sparse-array directed by `g` and then going group-by group.

`fmode` preserves all the attributes of the objects it is applied to (apart from names or row-names which are adjusted as necessary). If a data frame is passed to `fmode` and `drop = TRUE`, `base::unlist` will be called on the result, which might or might not be sensible depending on the data at hand.

Value

The statistical mode of `x`, grouped by `g`, or (if `TRA` is used) `x` transformed by its mode, grouped by `g`. See also Details.

See Also

`fmean`, `fmedian`, Fast Statistical Functions, Collapse Overview

Examples

```
## World Development Data
attach(wlddev)
## default vector method
fmode(PCGDP)                    # Numeric mode
fmode(PCGDP, iso3c)             # Grouped numeric mode
fmode(PCGDP, iso3c, LIFEEX)     # Grouped and weighted numeric mode
fmode(region)                   # Factor mode
fmode(date)                 # Date mode (defaults to first value since panel is balanced)
fmode(country)                  # Character mode (also defaults to first value)
fmode(OECD)                     # Logical mode
                           # ...all the above can also be performed grouped and weighted
## matrix method
m <- qM(airquality)
fmode(m)
fmode(m, na.rm = FALSE)         # NA frequency is also counted
fmode(m, airquality$Month)      # Groupwise
fmode(m, w = airquality$Day)    # Weighted: Later days in the month are given more weight
fmode(m>50, airquality$Month)   # Groupwise logical mode
                                # etc ...
## data.frame method
fmode(wlddev)                   # Gives one row
fmode(wlddev, drop = TRUE)      # calling unlist -> coerce to character vector
fmode(wlddev, iso3c)            # Grouped mode
fmode(wlddev, iso3c, LIFEEX)    # Grouped and weighted mode

detach(wlddev)
```

fNdistinct *Fast (Grouped) Distinct Value Count for Matrix-Like Objects*

Description

fNdistinct is a generic function that (column-wise) computes the number of distinct values in x, (optionally) grouped by g. It is significantly faster than length(unique(x)). The TRA argument can further be used to transform x using its (grouped) distinct value count.

Usage

```
fNdistinct(x, ...)

## Default S3 method:
fNdistinct(x, g = NULL, TRA = NULL, na.rm = TRUE,
           use.g.names = TRUE, ...)

## S3 method for class 'matrix'
fNdistinct(x, g = NULL, TRA = NULL, na.rm = TRUE,
           use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'data.frame'
fNdistinct(x, g = NULL, TRA = NULL, na.rm = TRUE,
           use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'grouped_df'
fNdistinct(x, TRA = NULL, na.rm = TRUE,
           use.g.names = FALSE, keep.group_vars = TRUE, ...)
```

Arguments

x	a vector, matrix, data.frame or grouped tibble (dplyr::grouped_df).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
TRA	an integer or quoted operator indicating the transformation to perform: 1 - "replace_fill" | 2 - "replace" | 3 - "-" | 4 - "-+" | 5 - "/" | 6 - "%" | 7 - "+" | 8 - "*" | 9 - "%%" | 10 - "-%%". See TRA.
na.rm	logical. TRUE: Skip missing values in x (faster computation). FALSE: Also consider 'NA' as one distinct value.
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.frame method). No row-names are generated for data.tables and grouped tibbles.
drop	*matrix and data.frame method:* drop dimensions and return an atomic vector if g = NULL and TRA = NULL.

keep.group_vars
: *grouped_df method:* Logical. FALSE removes grouping variables after computation.

...
: arguments to be passed to or from other methods.

Details

fNdistinct implements a fast algorithm to find the number of distinct values utilizing index- hashing implemented in the Rcpp::sugar::IndexHash class.

If na.rm = TRUE (the default), missing values will be skipped yielding substantial performance gains in data with many missing values. If na.rm = TRUE, missing values will simply be treated as any other value and read into the hash-map. Thus with the former, a numeric vector c(1.25,NaN,3.56,NA) will have a distinct value count of 2, whereas the latter will return a distinct value count of 4.

Grouped computations are currently performed by mapping the data to a sparse-array directed by g and then hash-mapping each group. This is often not much slower than using a larger hash-map for the entire data when g = NULL.

fNdistinct preserves all attributes of non-classed vectors / columns, and only the 'label' attribute (if available) of classed vectors / columns (i.e. dates or factors). When applied to data frames and matrices, the row-names are adjusted as necessary.

Value

Integer. The number of distinct values in x, grouped by g, or (if TRA is used) x transformed by its distinct value count, grouped by g.

See Also

fNobs, Fast Statistical Functions, Collapse Overview

Examples

```
## default vector method
fNdistinct(airquality$Solar.R)                   # Simple distinct value count
fNdistinct(airquality$Solar.R, airquality$Month) # Grouped distinct value count

## data.frame method
fNdistinct(airquality)
fNdistinct(airquality, airquality$Month)
fNdistinct(wlddev)                               # Works with data of all types!
head(fNdistinct(wlddev, wlddev$iso3c))

## matrix method
aqm <- qM(airquality)
fNdistinct(aqm)                             # Also works for character or logical matrices
fNdistinct(aqm, airquality$Month)

## method for grouped tibbles - for use with dplyr:
library(dplyr)
airquality %>% group_by(Month) %>% fNdistinct
wlddev %>% group_by(country) %>%
```

```
select(PCGDP,LIFEEX,GINI,ODA) %>% fNdistinct
```

---

fNobs *Fast (Grouped) Observation Count for Matrix-Like Objects*

Description

fNobs is a generic function that (column-wise) computes the number of non-missing values in x, (optionally) grouped by g. It is much faster than sum(!is.na(x)). The TRA argument can further be used to transform x using its (grouped) observation count.

Usage

```
fNobs(x, ...)

## Default S3 method:
fNobs(x, g = NULL, TRA = NULL, use.g.names = TRUE, ...)

## S3 method for class 'matrix'
fNobs(x, g = NULL, TRA = NULL, use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'data.frame'
fNobs(x, g = NULL, TRA = NULL, use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'grouped_df'
fNobs(x, TRA = NULL, use.g.names = FALSE, keep.group_vars = TRUE, ...)
```

Arguments

x	a vector, matrix, data.frame or grouped tibble (dplyr::grouped_df).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
TRA	an integer or quoted operator indicating the transformation to perform: 1 - "replace_fill" | 2 - "replace" | 3 - "-" | 4 - "-+" | 5 - "/" | 6 - "%" | 7 - "+" | 8 - "*" | 9 - "%%" | 10 - "-%%". See TRA.
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.frame method). No row-names are generated for data.tables and grouped tibbles.
drop	*matrix and data.frame method:* drop dimensions and return an atomic vector if g = NULL and TRA = NULL.
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation.
...	arguments to be passed to or from other methods.

Details

fNobs preserves all attributes of non-classed vectors / columns, and only the 'label' attribute (if available) of classed vectors / columns (i.e. dates or factors). When applied to data frames and matrices, the row-names are adjusted as necessary.

Value

Integer. The number of non-missing observations in x, grouped by g, or (if TRA is used) x transformed by its number of non-missing observations, grouped by g.

See Also

fNdistinct, Fast Statistical Functions, Collapse Overview

Examples

```
## default vector method
fNobs(airquality$Solar.R)                    # Simple Nobs
fNobs(airquality$Solar.R, airquality$Month) # Grouped Nobs

## data.frame method
fNobs(airquality)
fNobs(airquality, airquality$Month)
fNobs(wlddev)                                # Works with data of all types!
head(fNobs(wlddev, wlddev$iso3c))

## matrix method
aqm <- qM(airquality)
fNobs(aqm)                                  # Also works for character or logical matrices
fNobs(aqm, airquality$Month)

## method for grouped tibbles - for use with dplyr
library(dplyr)
airquality %>% group_by(Month) %>% fNobs
wlddev %>% group_by(country) %>%
           select(PCGDP,LIFEEX,GINI,ODA) %>% fNobs
```

---

fprod *Fast (Grouped, Weighted) Product for Matrix-Like Objects*

Description

fprod is a generic function that computes the (column-wise) product of all values in x, (optionally) grouped by g and/or weighted by w. The TRA argument can further be used to transform x using its (grouped) product.

Usage

```
fprod(x, ...)

## Default S3 method:
fprod(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = TRUE, ...)

## S3 method for class 'matrix'
fprod(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'data.frame'
fprod(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'grouped_df'
fprod(x, w = NULL, TRA = NULL, na.rm = TRUE,
      use.g.names = FALSE, keep.group_vars = TRUE, keep.w = TRUE, ...)
```

Arguments

x	a numeric vector, matrix, data.frame or grouped tibble (dplyr::grouped_df).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
w	a numeric vector of (non-negative) weights, may contain missing values.
TRA	an integer or quoted operator indicating the transformation to perform: 1 - "replace_fill" | 2 - "replace" | 3 - "-" | 4 - "-+" | 5 - "/" | 6 - "%" | 7 - "+" | 8 - "*" | 9 - "%%" | 10 - "-%%". See TRA.
na.rm	logical. Skip missing values in x. Defaults to TRUE and implemented at very little computational cost. If na.rm = FALSE a NA is returned when encountered.
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.frame method). No row-names are generated for data.tables and (default) grouped tibbles.
drop	*matrix and data.frame method:* drop dimensions and return an atomic vector if g = NULL and TRA = NULL.
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation.
keep.w	*grouped_df method:* Logical. Retain product of weighting variable after computation (if contained in grouped_df).
...	arguments to be passed to or from other methods.

Details

Non-grouped product computations internally utilize long-doubles in C++, for additional numeric precision.

Missing-value removal as controlled by the na.rm argument is done very efficiently by simply skipping them in the computation (thus setting na.rm = FALSE on data with no missing values doesn't give extra speed). Large performance gains can nevertheless be achieved in the presence of missing values if na.rm = FALSE, since then the corresponding computation is terminated once a NA is encountered and NA is returned (unlike base::prod which just runs through without any checks).

This all seamlessly generalizes to grouped computations, which are performed in a single pass (without splitting the data) and therefore extremely fast.

The weighted product is computed as prod(x * w). If na.rm = TRUE, missing values will be removed from both x and w i.e. utilizing only x[complete.cases(x,w)] and w[complete.cases(x,w)].

When applied to data frame's with groups or drop = FALSE, fprod preserves all column attributes (such as variable labels) but does not distinguish between classed and unclassed objects. The attributes of the data frame itself are also preserved.

Value

The product of x, grouped by g, or (if TRA is used) x transformed by its product, grouped by g.

See Also

fsum, Fast Statistical Functions, Collapse Overview

Examples

```
## default vector method
mpg <- mtcars$mpg
fprod(mpg)                          # Simple product
fprod(mpg, w = mtcars$hp)           # Weighted product
fprod(mpg, TRA = "/")               # Simple transformation: Divide by product
fprod(mpg, mtcars$cyl)              # Grouped product
fprod(mpg, mtcars$cyl, mtcars$hp)   # Weighted grouped product
fprod(mpg, mtcars[c(2,8:9)])        # More groups...
g <- GRP(mtcars, ~ cyl + vs + am)   # Precomputing groups gives more speed !!
fprod(mpg, g)
fprod(mpg, g, TRA = "/")            # Groupwise divide by product

## data.frame method
fprod(mtcars)
fprod(mtcars, TRA = "/")
fprod(mtcars, g)
fprod(mtcars, g, use.g.names = FALSE) # No row-names generated

## matrix method
m <- qM(mtcars)
fprod(m)
fprod(m, TRA = "/")
fprod(m, g) # etc...

## method for grouped tibbles - for use with dplyr
library(dplyr)
mtcars %>% group_by(cyl,vs,am) %>% fprod(hp)   # Weighted grouped product
```

```
mtcars %>% fgroup_by(cyl,vs,am) %>% fprod(hp)  # Equivalent but faster
mtcars %>% fgroup_by(cyl,vs,am) %>% fprod(TRA = "/")
mtcars %>% fgroup_by(cyl,vs,am) %>% fselect(mpg) %>% fprod
```

`fscale` *Fast (Grouped, Weighted) Scaling and Centering of Matrix-like Objects*

Description

`fscale` is a generic function to efficiently standardize (scale and center) data. `STD` is a wrapper around `fscale` representing the 'standardization operator', with more options than `fscale` when applied to matrices and data frames. Standardization can be simple or groupwise, ordinary or weighted.

Note: For centering without scaling see `fwithin/W`.

Usage

```
fscale(x, ...)
   STD(x, ...)

## Default S3 method:
fscale(x, g = NULL, w = NULL, na.rm = TRUE, mean = 0, sd = 1, ...)
## Default S3 method:
STD(x, g = NULL, w = NULL, na.rm = TRUE, mean = 0, sd = 1, ...)

## S3 method for class 'matrix'
fscale(x, g = NULL, w = NULL, na.rm = TRUE, mean = 0, sd = 1, ...)
## S3 method for class 'matrix'
STD(x, g = NULL, w = NULL, na.rm = TRUE, mean = 0, sd = 1,
    stub = "STD.", ...)

## S3 method for class 'data.frame'
fscale(x, g = NULL, w = NULL, na.rm = TRUE, mean = 0, sd = 1, ...)
## S3 method for class 'data.frame'
STD(x, by = NULL, w = NULL, cols = is.numeric, na.rm = TRUE,
    mean = 0, sd = 1, stub = "STD.", keep.by = TRUE, keep.w = TRUE, ...)

# Methods for compatibility with plm:

## S3 method for class 'pseries'
fscale(x, effect = 1L, w = NULL, na.rm = TRUE, mean = 0, sd = 1, ...)
## S3 method for class 'pseries'
STD(x, effect = 1L, w = NULL, na.rm = TRUE, mean = 0, sd = 1, ...)

## S3 method for class 'pdata.frame'
fscale(x, effect = 1L, w = NULL, na.rm = TRUE, mean = 0, sd = 1, ...)
```

```
## S3 method for class 'pdata.frame'
STD(x, effect = 1L, w = NULL, cols = is.numeric, na.rm = TRUE,
    mean = 0, sd = 1, stub = "STD.", keep.ids = TRUE, keep.w = TRUE, ...)

# Methods for compatibility with dplyr:

## S3 method for class 'grouped_df'
fscale(x, w = NULL, na.rm = TRUE, mean = 0, sd = 1,
       keep.group_vars = TRUE, keep.w = TRUE, ...)
## S3 method for class 'grouped_df'
STD(x, w = NULL, na.rm = TRUE, mean = 0, sd = 1,
    stub = "STD.", keep.group_vars = TRUE, keep.w = TRUE, ...)
```

Arguments

x	a numeric vector, matrix, data.frame, panel-series (plm::pseries), panel-data.frame (plm::pdata.frame) or grouped tibble (dplyr::grouped_df).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
by	*STD data.frame method*: Same as g, but also allows one- or two-sided formulas i.e. ~ group1 or var1 + var2 ~ group1 + group2. See Examples.
cols	*data.frame method*: Select columns to scale using a function, column names or indices. Default: All numeric variables. *Note*: cols is ignored if a two-sided formula is passed to by.
w	a numeric vector of (non-negative) weights. STD data frame and pdata.frame methods also allow a one-sided formula i.e. ~ weightcol. The grouped_df (*dplyr*) method supports lazy-evaluation. See Examples.
na.rm	logical. skip missing values in x or w when computing means and sd's.
effect	*plm* methods: Select which panel identifier should be used as grouping variable. 1L means first variable in the plm::index, 2L the second etc. if more than one integer is supplied, the corresponding index-variables are interacted.
stub	a prefix or stub to rename all transformed columns. FALSE will not rename columns.
mean	the mean to center on (default is 0). If mean = FALSE, no centering will be performed. In that case the scaling is mean-preserving. A numeric value different from 0 (i.e. mean = 5) will be added to the data after subtracting out the mean(s), such that the data will have a mean of 5. A special option when performing grouped scaling and centering is mean = "overall.mean". In that case the overall mean of the data will be added after subtracting out group means.
sd	the standard deviation to scale the data to (default is 1). A numeric value different from 0 (i.e. sd = 3) will scale the data to have a standard deviation of 3. A special option when performing grouped scaling is sd = "within.sd". In that case the within standard deviation (= the standard deviation of the group-centered series) will be calculated and applied to each group. The results is that the variance of the data within each group is harmonized without forcing a certain variance (such as 1).

keep.by, keep.ids, keep.group_vars
: *data.frame, pdata.frame and grouped_df methods*: Logical. Retain grouping / panel-identifier columns in the output. For STD.data.frame this only works if grouping variables were passed in a formula.

keep.w
: *data.frame, pdata.frame and grouped_df methods*: Logical. Retain column containing the weights in the output. Only works if w is passed as formula / lazy-expression.

...
: arguments to be passed to or from other methods.

Details

If g = NULL, fscale by default (column-wise) subtracts the mean or weighted mean (if w is supplied) from all data points in x, and then divides this difference by the standard deviation or frequency-weighted standard deviation (if w is supplied). The result is that all columns in x will have mean 0 and standard deviation 1. Alternatively, data can be scaled to have a mean of mean and a standard deviation of sd. If mean = FALSE the data is only scaled (not centered) such that the mean of the data is preserved.

Means and standard deviations are computed using Welford's numerically stable online algorithm.

With groups supplied to g, this standardizing becomes groupwise, so that in each group (in each column) the data points will have mean mean and standard deviation sd. Naturally if mean = FALSE then each group is just scaled and the mean is preserved. For centering without scaling see fwithin.

If na.rm = FALSE and a NA or NaN is encountered, the mean and sd for that group will be NA, and all data points belonging to that group will also be NA in the output.

If na.rm = TRUE, means and sd's are computed (column-wise) on the available data points, and also the weight vector can have missing values. In that case (w also has missing values), the weighted mean an sd are computed on (column-wise) complete.cases(x,w), and x is scaled using these statistics. *Note* that fscale will not insert a missing value in x if the weight for that value is missing, rather, that value will be scaled using a weighted mean and standard-deviated computed without itself! (The intention here is that a few (randomly) missing weights shouldn't break the computation when na.rm = TRUE, but it is not meant for weight vectors with many missing values. If you don't like this behavior, you should prepare your data using x[is.na(w),] <- NA, or impute your weight vector for non-missing x).

Special options for grouped scaling are mean = "overall.mean" and sd = "within.sd". The former group-centers vectors on the overall mean of the data (see fwithin for more details) and the latter scales the data in each group to have the within-group standard deviation (= the standard deviation of the group-centered data). Thus scaling a grouped vector with options mean = "overall.mean" and sd = "within.sd" amounts to removing all differences in the mean and standard deviations between these groups. In weighted computations, mean = "overall.mean" will subtract weighted group-means from the data and add the overall weighted mean of the data, whereas sd = "within.sd" will compute the weighted within- standard deviation and apply it to each group.

Value

x standardized (mean = mean, standard deviation = sd), grouped by g/by, weighted with w. See Details.

See Also

fwithin/W, Fast Statistical Functions, TRA, Data Transformations, Collapse Overview

Examples

```
## Simple Scaling & Centering / Standardizing
fscale(mtcars)                    # Doesn't rename columns
STD(mtcars)                       # By default adds a prefix
qsu(STD(mtcars))                  # See that is works
qsu(STD(mtcars, mean = 5, sd = 3)) # Assigning a mean of 5 and a standard deviation of 3
qsu(STD(mtcars, mean = FALSE))    # No centering: Scaling is mean-preserving

## Panel-Data
head(fscale(get_vars(wlddev,9:12), wlddev$iso3c))  # Standardizing 4 series within each country
head(STD(wlddev, ~iso3c, cols = 9:12))              # Same thing using STD, id's added
pwcor(fscale(get_vars(wlddev,9:12), wlddev$iso3c)) # Correlaing panel-series after standardizing

fmean(get_vars(wlddev, 9:12))                       # This calculates the overall means
fsd(fwithin(get_vars(wlddev, 9:12), wlddev$iso3c)) # This calculates the within standard deviations
qsu(fscale(get_vars(wlddev, 9:12), wlddev$iso3c,   # This group-centers on the overall mean and
   mean = "overall.mean", sd = "within.sd"),     # group-scales to the within standard deviation
    by = wlddev$iso3c)                           # -> data harmonized in the first 2 moments


## Using plm
pwlddev <- plm::pdata.frame(wlddev, index = c("iso3c","year"))
head(STD(pwlddev))                        # Standardizing all numeric variables by country
head(STD(pwlddev, effect = 2L))              # Standardizing all numeric variables by year

## Weighted Standardizing
weights = abs(rnorm(nrow(wlddev)))
head(fscale(get_vars(wlddev,9:12), wlddev$iso3c, weights))
head(STD(wlddev, ~iso3c, weights, 9:12))

# Using dplyr
library(dplyr)
wlddev %>% group_by(iso3c) %>% select(PCGDP,LIFEEX) %>% STD
wlddev %>% group_by(iso3c) %>% select(PCGDP,LIFEEX) %>% STD(weights) # weighted standardizing
wlddev %>% group_by(iso3c) %>% select(PCGDP,LIFEEX,ODA) %>% STD(ODA) # weighting by ODA ->
# ..keeps the weight column unless keep.w = FALSE
```

---

fsubset *Fast Subsetting*

Description

fsubset returns subsets of vectors, matrices or data frames which meet conditions. It is programmed very efficiently and uses C source code from the *data.table* package. Especially for data.frame's it is significantly (4-5 times) faster than base::subset (or dplyr::filter). The methods also provide more functionality compared to base::subset. The function ss provides a significantly faster alternative to [.data.frame.

Usage

```
fsubset(x, ...)
sbt(x, ...)     # Shortcut for fsubset

## Default S3 method:
fsubset(x, subset, ...)

## S3 method for class 'matrix'
fsubset(x, subset, ..., drop = FALSE)

## S3 method for class 'data.frame'
fsubset(x, subset, ...)

# Fast subsetting data.frames (replaces `[`)
ss(data, i, j)
```

Arguments

x	object to be subsetted.
data	a data.frame.
subset	logical expression indicating elements or rows to keep: missing values are taken as false. The default and matrix methods only support logical vectors or row-indices (or a character vector of rownames if the matrix has rownames; the data.frame method also supports logical vectors or row-indices).
...	For the matrix data.frame method: multiple comma-separated expressions indicating columns to select. Otherwise: further arguments to be passed to or from other methods.
drop	passed on to [indexing operator. Only available for the matrix method.
i	positive or negative row-indices or a logical vector to subset the rows of data.
j	a vector or column names, positive or negative indices or a suitable logical vector to subset the columns of data. *Note:* Negative indices are converted to positive ones using j <-seq_along(data)[j].

Details

fsubset is a generic function, with methods supplied for matrices, data frames and vectors (including lists). It represents an improvement in both speed and functionality over base::subset. The non-generic function ss is an improvement of [.data.frame. For subsetting columns alone, please see selecting and replacing columns.

For ordinary vectors, the result is .Call(C_subsetVector,x,subset), where C_subsetVector is an internal function in the *data.table* package. The subset can be integer or logical. Appropriate errors are delivered for wrong use.

For matrices the implementation is all base-R but slightly more efficient and more versatile than base::subset.matrix. Thus it is possible to subset matrix rows using logical or integer vectors, or character vectors matching rownames. The drop argument is passed on to the indexing method for matrices.

For both matrices and data frames, the ... argument can be used to subset columns, and is evaluated in a non-standard way. Thus it can support vectors of column names, indices or logical vectors, but also multiple comma separated column names passed without quotes, each of which may also be replaced by a sequence of columns i.e. `col1:coln` (see examples).

For data frames, the `subset` argument is also evaluated in a non-standard way. Thus next to vector of row-indices or logical vectors, it supports logical expressions of the form `col2 > 5 & col2 < col3` etc. (see examples). The data frame method uses `C_subsetDT`, an internal C function from the *data.table* package to subset data.frames, hence it is significantly faster than `base::subset.data.frame`. If fast data frame subsetting is required but no non-standard evaluation, the function `ss` is slightly simpler and faster.

Factors may have empty levels after subsetting; unused levels are not automatically removed. See `droplevels` for a way to drop all unused levels from a data frame.

Value

An object similar to `x` containing just the selected elements (for a vector), rows and columns (for a matrix or data frame).

See Also

`fselect`, `get_vars`, `ftransform`, Data Frame Manipulation, Collapse Overview

Examples

```
fsubset(airquality, Temp > 80, Ozone, Temp)
fsubset(airquality, Day == 1, -Temp)
fsubset(airquality, Day == 1, -(Day:Temp))
fsubset(airquality, Day == 1, Ozone:Wind)
fsubset(airquality, Day == 1 & !is.na(Ozone), Ozone:Wind, Month)

fsubset(airquality, 1:10, 2:3)
ss(airquality, 1:10, 2:3)         # Slightly faster !
```

---

fsum *Fast (Grouped, Weighted) Sum for Matrix-Like Objects*

Description

`fsum` is a generic function that computes the (column-wise) sum of all values in `x`, (optionally) grouped by `g` and/or weighted by `w` (i.e. to calculate survey totals). The `TRA` argument can further be used to transform `x` using its (grouped, weighted) sum.

Usage

```
fsum(x, ...)

## Default S3 method:
fsum(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
```

```
     use.g.names = TRUE, ...)

## S3 method for class 'matrix'
fsum(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'data.frame'
fsum(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'grouped_df'
fsum(x, w = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = FALSE, keep.group_vars = TRUE, keep.w = TRUE, ...)
```

Arguments

x	a numeric vector, matrix, data.frame or grouped tibble (dplyr::grouped_df).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
w	a numeric vector of (non-negative) weights, may contain missing values.
TRA	an integer or quoted operator indicating the transformation to perform: 1 - "replace_fill" | 2 - "replace" | 3 - "-" | 4 - "-+" | 5 - "/" | 6 - "%" | 7 - "+" | 8 - "*" | 9 - "%%" | 10 - "-%%". See TRA.
na.rm	logical. Skip missing values in x. Defaults to TRUE and implemented at very little computational cost. If na.rm = FALSE a NA is returned when encountered.
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.frame method). No row-names are generated for data.tables and (default) grouped tibbles.
drop	*matrix and data.frame method:* drop dimensions and return an atomic vector if g = NULL and TRA = NULL.
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation.
keep.w	*grouped_df method:* Logical. Retain summed weighting variable after computation (if contained in grouped_df).
...	arguments to be passed to or from other methods.

Details

Missing-value removal as controlled by the na.rm argument is done very efficiently by simply skipping them in the computation (thus setting na.rm = FALSE on data with no missing values doesn't give extra speed). Large performance gains can nevertheless be achieved in the presence of missing values if na.rm = FALSE, since then the corresponding computation is terminated once a NA is encountered and NA is returned (unlike base::sum which just runs through without any checks).

The weighted sum (i.e. survey total) is computed as sum(x * w). If na.rm = TRUE, missing values will be removed from both x and w i.e. utilizing only x[complete.cases(x,w)] and w[complete.cases(x,w)].

This all seamlessly generalizes to grouped computations, which are performed in a single pass (without splitting the data) and therefore extremely fast. See Benchmark and Examples below.

When applied to data frame's with groups or `drop = FALSE`, `fsum` preserves all column attributes (such as variable labels) but does not distinguish between classed and unclassed objects. The attributes of the data frame itself are also preserved.

Value

The (`w` weighted) sum of `x`, grouped by `g`, or (if `TRA` is used) `x` transformed by its sum, grouped by `g`.

See Also

`fprod`, Fast Statistical Functions, Collapse Overview

Examples

```
## default vector method
mpg <- mtcars$mpg
fsum(mpg)                          # Simple sum
fsum(mpg, w = mtcars$hp)           # Weighted sum (total): Weighted by hp
fsum(mpg, TRA = "%")           # Simple transformation: obtain percentages of mpg
fsum(mpg, mtcars$cyl)              # Grouped sum
fsum(mpg, mtcars$cyl, mtcars$hp)  # Weighted grouped sum (total)
fsum(mpg, mtcars[c(2,8:9)])        # More groups...
g <- GRP(mtcars, ~ cyl + vs + am) # Precomputing groups gives more speed !!
fsum(mpg, g)
fmean(mpg, g) == fsum(mpg, g) / fNobs(mpg, g)
fsum(mpg, g, TRA = "%")            # Percentages by group

## data.frame method
fsum(mtcars)
fsum(mtcars, TRA = "%")
fsum(mtcars, g)
fsum(mtcars, g, TRA = "%")

## matrix method
m <- qM(mtcars)
fsum(m)
fsum(m, TRA = "%")
fsum(m, g)
fsum(m, g, TRA = "%")

## method for grouped tibbles - for use with dplyr
library(dplyr)
mtcars %>% group_by(cyl,vs,am) %>% fsum(hp)  # Weighted grouped sum (total)
mtcars %>% fgroup_by(cyl,vs,am) %>% fsum(hp) # Equivalent but faster !!
mtcars %>% fgroup_by(cyl,vs,am) %>% fsum(TRA = "%")
mtcars %>% fgroup_by(cyl,vs,am) %>% fselect(mpg) %>% fsum
```

Benchmark

```
## Let's run some benchmarks and compare fsum against data.table and base::rowsum
# Starting with small data
mtcDT <- qDT(mtcars)
f <- qF(mtcars$cyl)

library(microbenchmark)
microbenchmark(mtcDT[, lapply(.SD, sum), by = f],
               rowsum(mtcDT, f, reorder = FALSE),
               fsum(mtcDT, f, na.rm = FALSE), unit = "relative")
# My results:
                    expr      min       lq     mean   median       uq      max neval cld
 mtcDT[, lapply(.SD, sum), by = f] 145.436928 123.542134 88.681111 98.336378 71.880479 85.217726  100   c
 rowsum(mtcDT, f, reorder = FALSE)  2.833333  2.798203  2.489064  2.937889  2.425724  2.181173  100  b
    fsum(mtcDT, f, na.rm = FALSE)  1.000000  1.000000  1.000000  1.000000  1.000000  1.000000  100 a

# Now larger data
tdata <- qDT(replicate(100, rnorm(1e5), simplify = FALSE)) # 100 columns with 100.000 obs
f <- qF(sample.int(1e4, 1e5, TRUE))                  # A factor with 10.000 groups

microbenchmark(tdata[, lapply(.SD, sum), by = f],
               rowsum(tdata, f, reorder = FALSE),
               fsum(tdata, f, na.rm = FALSE), unit = "relative")
# My results:
                     expr     min      lq     mean   median       uq      max neval cld
 tdata[, lapply(.SD, sum), by = f] 2.646992 2.975489 2.834771 3.081313 3.120070 1.2766475  100   c
 rowsum(tdata, f, reorder = FALSE) 1.747567 1.753313 1.629036 1.758043 1.839348 0.2720937  100  b
    fsum(tdata, f, na.rm = FALSE) 1.000000 1.000000 1.000000 1.000000 1.000000 1.0000000  100 a
```

---

ftransform *Fast Transform and Compute Columns on a Data Frame*

Description

`ftransform` is a much faster update of `base::transform` for data frames. It returns the data frame with new columns computed and/or existing columns modified or deleted. `settransform` does all of that by reference i.e. it modifies the data frame in the global environment. `fcompute` can be used to compute new columns from the columns in a data frame and returns only the computed columns.

Usage

```
# Modify and return 'data.frame'
ftransform(X, ...)
tfm(X, ...)            # Shortcut for ftransform

# Modify 'data.frame' by reference
```

```
settransform(X, ...)
settfm(X, ...)         # Shortcut for settransform

# Compute and return new 'data.frame' from existing one
fcompute(X, ...)
```

Arguments

X	a data.frame.
...	further arguments of the form column = value. The value can be a combination of other columns, a scalar value, or NULL, which deletes column.

Details

The ... arguments to ftransform are tagged vector expressions, which are evaluated in the data frame X. The tags are matched against names(X), and for those that match, the value replace the corresponding variable in X, and the others are appended to X. It is also possible to delete columns by assigning NULL to them, i.e. ftransform(data,column = NULL) removes column from the data.

The function settransform does all of that by reference, but uses base-R's copy-on modify semantics, which is equivalent to replacing the data with <- (thus it is still memory efficient but the data will have a different memory address after each call of settransform).

Finally, the function fcompute functions just like ftransform, but returns only the changed / computed columns without modifying or appending the data in X.

Value

The modified data.frame X, or, for fcompute, a new data.frame with the columns computed on X. All attributes of X are preserved.

See Also

with, within, Data Frame Manipulation, Collapse Overview

Examples

```
## ftransform modifies and returns a data.frame
ftransform(airquality, Ozone = -Ozone)
ftransform(airquality, new = -Ozone, Temp = (Temp-32)/1.8)
ftransform(airquality, new = -Ozone, new2 = 1, Temp = NULL)  # Deleting Temp
ftransform(airquality, Ozone = NULL, Temp = NULL)            # Deleting columns

## settransform modifies a data.frame in the global environment
airquality_c <- airquality
settransform(airquality_c, Ratio = Ozone / Temp, Ozone = NULL, Temp = NULL)
head(airquality_c)
rm(airquality_c)

## fcompute only returns the modified / computed data
fcompute(airquality, Ozone = -Ozone)
fcompute(airquality, new = -Ozone, Temp = (Temp-32)/1.8)
```

```
fcompute(airquality, new = -Ozone, new2 = 1)
```

`fvar, fsd` *Fast (Grouped, Weighted) Variance and Standard Deviation for Matrix-Like Objects*

Description

`fvar` and `fsd` are generic functions that compute the (column-wise) variance and standard deviation of `x`, (optionally) grouped by `g` and/or frequency-weighted by `w`. The `TRA` argument can further be used to transform `x` using its (grouped, weighted) variance/sd.

Usage

```
fvar(x, ...)
fsd(x, ...)

## Default S3 method:
fvar(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, stable.algo = TRUE, ...)
## Default S3 method:
fsd(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
    use.g.names = TRUE, stable.algo = TRUE, ...)

## S3 method for class 'matrix'
fvar(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, drop = TRUE, stable.algo = TRUE, ...)
## S3 method for class 'matrix'
fsd(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
    use.g.names = TRUE, drop = TRUE, stable.algo = TRUE, ...)

## S3 method for class 'data.frame'
fvar(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, drop = TRUE, stable.algo = TRUE, ...)
## S3 method for class 'data.frame'
fsd(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
    use.g.names = TRUE, drop = TRUE, stable.algo = TRUE, ...)

## S3 method for class 'grouped_df'
fvar(x, w = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = FALSE, keep.group_vars = TRUE, keep.w = TRUE,
     stable.algo = TRUE, ...)
## S3 method for class 'grouped_df'
fsd(x, w = NULL, TRA = NULL, na.rm = TRUE,
    use.g.names = FALSE, keep.group_vars = TRUE, keep.w = TRUE,
    stable.algo = TRUE, ...)
```

Arguments

x	a numeric vector, matrix, data.frame or grouped tibble (dplyr::grouped_df).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
w	a numeric vector of (non-negative) weights, may contain missing values.
TRA	an integer or quoted operator indicating the transformation to perform: 1 - "replace_fill" | 2 - "replace" | 3 - "-" | 4 - "-+" | 5 - "/" | 6 - "%" | 7 - "+" | 8 - "*" | 9 - "%%" | 10 - "-%%". See TRA.
na.rm	logical. Skip missing values in x. Defaults to TRUE and implemented at very little computational cost. If na.rm = FALSE a NA is returned when encountered.
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.frame method). No row-names are generated for data.tables and grouped tibbles.
drop	*matrix and data.frame method:* drop dimensions and return an atomic vector if g = NULL and TRA = NULL.
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation.
keep.w	*grouped_df method:* Logical. Retain summed weighting variable after computation (if contained in grouped_df).
stable.algo	logical. TRUE (default) use Welford's numerically stable online algorithm. FALSE implements a faster but numerically unstable one-pass method. See Details.
...	arguments to be passed to or from other methods.

Details

Welford's online algorithm used by default to compute the variance is well described here (the section *Weighted incremental algorithm* also shows how the weighted variance is obtained by this algorithm).

If stable.algo = FALSE, the variance is computed in one-pass as (sum(x^2)-n*mean(x)^2)/(n-1), where sum(x^2) is the sum of squares from which the expected sum of squares n*mean(x)^2 is subtracted, normalized by n-1 (Bessel's correction). This is numerically unstable if sum(x^2) and n*mean(x)^2 are large numbers very close together, which will be the case for large n, large x-values and small variances (catastrophic cancellation occurs, leading to a loss of numeric precision). Numeric precision is however still maximized through the internal use of long doubles in C++, and the fast algorithm can be up to 4-times faster compared to Welford's method.

The weighted variance is computed with frequency weights as (sum(x^2*w)-sum(w)*weighted.mean(x,w)^2)/(sum(w)-1
If na.rm = TRUE, missing values will be removed from both x and w i.e. utilizing only x[complete.cases(x,w)] and w[complete.cases(x,w)].

Missing-value removal as controlled by the na.rm argument is done very efficiently by simply skipping the values (thus setting na.rm = FALSE on data with no missing values doesn't give extra speed). Large performance gains can nevertheless be achieved in the presence of missing values if na.rm = FALSE, since then the corresponding computation is terminated once a NA is encountered and NA is returned.

This all seamlessly generalizes to grouped computations, which are performed in a single pass (without splitting the data) and therefore extremely fast.

When applied to data frame's with groups or `drop = FALSE`, `fvar/fsd` preserves all column attributes (such as variable labels) but does not distinguish between classed and unclassed object (thus applying `fvar/fsd` to a factor column will give a 'malformed factor' error, and applying it to a date variable will give an error or a pretty weird date). The attributes of the `data.frame` itself are also preserved.

Value

`fvar` returns the variance of `x`, grouped by `g`, or (if `TRA` is used) `x` transformed by its variance, grouped by `g`. `fsd` computes the standard deviation of `x` in like manor.

See Also

Fast Statistical Functions, Collapse Overview

Examples

```
## default vector method
fvar(mtcars$mpg)                      # Simple variance (all examples also hold for fvar!)
fsd(mtcars$mpg)                             # Simple standard deviation
fsd(mtcars$mpg, w = mtcars$hp)              # Weighted sd: Weighted by hp
fsd(mtcars$mpg, TRA = "/")             # Simple transformation: scaling (See also ?fscale)
fsd(mtcars$mpg, mtcars$cyl)                 # Grouped sd
fsd(mtcars$mpg, mtcars$cyl, mtcars$hp)      # Grouped weighted sd
fsd(mtcars$mpg, mtcars$cyl, TRA = "/")      # Scaling by group
fsd(mtcars$mpg, mtcars$cyl, mtcars$hp, "/") # Group-scaling using weighted group sds

## data.frame method
fsd(iris)                           # This works, although 'Species' is a factor variable
fsd(mtcars, drop = FALSE)           # This works, all columns are numeric variables
fsd(iris[-5], iris[5])         # By Species: iris[5] is still a list, and thus passed to GRP()
fsd(iris[-5], iris[[5]])        # Same thing much faster: fsd recognizes 'Species' is a factor
fsd(iris[-5], iris[[5]], TRA = "/") # Data scaled by species (see also fscale)

## matrix method
m <- qM(mtcars)
fsd(m)
fsd(m, mtcars$cyl) # etc...

## method for grouped tibbles - for use with dplyr:
library(dplyr)
mtcars %>% group_by(cyl,vs,am) %>% fsd
mtcars %>% group_by(cyl,vs,am) %>% fsd(keep.group_vars = FALSE) # remove grouping columns
mtcars %>% group_by(cyl,vs,am) %>% fsd(hp)      # Weighted by hp
mtcars %>% group_by(cyl,vs,am) %>% fsd(hp, "/") # Weighted scaling transformation
```

GGDC10S *Groningen Growth and Development Centre 10-Sector Database*

Description

The GGDC 10-Sector Database provides a long-run internationally comparable dataset on sectoral productivity performance in Africa, Asia, and Latin America. Variables covered in the data set are annual series of value added (in local currency), and persons employed for 10 broad sectors.

Usage

```
data("GGDC10S")
```

Format

A data frame with 5027 observations on the following 16 variables.

`Country` *char*: Country (43 countries)

`Regioncode` *char*: ISO3 Region code

`Region` *char*: Region (6 World Regions)

`Variable` *char*: Variable (Value Added or Employment)

`Year` *num*: Year (67 Years, 1947-2013)

`AGR` *num*: Agriculture

`MIN` *num*: Mining

`MAN` *num*: Manufacturing

`PU` *num*: Utilities

`CON` *num*: Construction

`WRT` *num*: Trade, restaurants and hotels

`TRA` *num*: Transport, storage and communication

`FIRE` *num*: Finance, insurance, real estate and business services

`GOV` *num*: Government services

`OTH` *num*: Community, social and personal services

`SUM` *num*: Summation of sector GDP

Source

https://www.rug.nl/ggdc/productivity/10-sector/

References

Timmer, M. P., de Vries, G. J., & de Vries, K. (2015). "Patterns of Structural Change in Developing Countries." . In J. Weiss, & M. Tribe (Eds.), *Routledge Handbook of Industry and Development.* (pp. 65-83). Routledge.

See Also

wlddev, Collapse Overview

Examples

```
namlab(GGDC10S, class = TRUE)
qsu(GGDC10S, ~ Variable, ~ Variable + Country, vlabels = TRUE)

## Not run:
library(data.table)
library(ggplot2)

## World Regions Structural Change Plot

dat <- GGDC10S
fselect(dat, AGR:OTH) <- replace_outliers(dapply(fselect(dat, AGR:OTH), `*`, 1 / dat$SUM),
                                          0, NA, "min")
dat$Variable <- recode_char(dat$Variable, VA = "Value Added Share", EMP = "Employment Share")
dat <- collap(dat, ~ Variable + Region + Year, cols = 6:15)
dat <- melt(qDT(dat), 1:3, variable.name = "Sector", na.rm = TRUE)

ggplot(aes(x = Year, y = value, fill = Sector), data = dat) +
  geom_area(position = "fill", alpha = 0.9) + labs(x = NULL, y = NULL) +
  theme_linedraw(base_size = 14) + facet_grid(Variable ~ Region, scales = "free_x") +
  scale_fill_manual(values = sub("#00FF66FF", "#00CC66", rainbow(10))) +
  scale_x_continuous(breaks = scales::pretty_breaks(n = 7), expand = c(0, 0))+
  scale_y_continuous(breaks = scales::pretty_breaks(n = 10), expand = c(0, 0),
                     labels = scales::percent) +
 theme(axis.text.x = element_text(angle = 315, hjust = 0, margin = ggplot2::margin(t = 0)),
       strip.background = element_rect(colour = "grey30", fill = "grey30"))

# A function to plot the structural change of an arbitrary country

plotGGDC <- function(ctry) {
dat <- fsubset(GGDC10S, Country == ctry, Variable, Year, AGR:SUM)
fselect(dat, AGR:OTH) <- replace_outliers(dapply(fselect(dat, AGR:OTH), `*`, 1 / dat$SUM),
                                          0, NA, "min")
dat$SUM <- NULL
dat$Variable <- recode_char(dat$Variable, VA = "Value Added Share", EMP = "Employment Share")
dat <- melt(qDT(dat), 1:2, variable.name = "Sector", na.rm = TRUE)

ggplot(aes(x = Year, y = value, fill = Sector), data = dat) +
  geom_area(position = "fill", alpha = 0.9) + labs(x = NULL, y = NULL) +
  theme_linedraw(base_size = 14) + facet_wrap( ~ Variable) +
  scale_fill_manual(values = sub("#00FF66", "#00CC66", rainbow(10))) +
  scale_x_continuous(breaks = scales::pretty_breaks(n = 7), expand = c(0, 0)) +
  scale_y_continuous(breaks = scales::pretty_breaks(n = 10), expand = c(0, 0),
                     labels = scales::percent) +
 theme(axis.text.x = element_text(angle = 315, hjust = 0, margin = ggplot2::margin(t = 0)),
       strip.background = element_rect(colour = "grey20", fill = "grey20"),
       strip.text = element_text(face = "bold"))
}
```

```
plotGGDC("BWA")

## End(Not run)
```

`groupid` *Generate Run-Length Type Group-Id*

Description

`groupid` is an enhanced version of `data.table::rleid` for atomic vectors. It generates a run-length type group-id where consecutive identical values are assigned the same integer. It is a generalization as it can be applied to unordered vectors, generate group id's starting from an arbitrary value, and skip missing values.

Usage

```
groupid(x, o = NULL, start = 1L, na.skip = FALSE, check.o = TRUE)
```

Arguments

`x`	a atomic vector of any type. Attributes are not considered.
`o`	an (optional) integer ordering vector specifying the order by which to pass through `x`.
`start`	integer. The starting value of the resulting group-id. Default is starting from 1. For C++ programmers, starting from 0 could be a better choice.
`na.skip`	logical. Skip missing values i.e. if `TRUE` something like `groupid(c("a",NA,"a"))` gives `c(1,NA,1)` whereas `FALSE` gives `c(1,2,3)`.
`check.o`	logical. Programmers option: `FALSE` prevents checking that each element of `o` is in the range `[1,length(x)]`, it only checks the length of `o`. This gives some extra speed, but will terminate R if any element of `o` is too large or too small.

Value

An integer vector of class 'qG'. See `qG`.

See Also

`seqid`, `qG`, Fast (Ordered) Grouping, Collapse Overview

Examples

```
groupid(airquality$Month)
groupid(airquality$Month, start = 0)
groupid(wlddev$country)

## Same thing since country is alphabetically ordered: (groupid is faster..)
all.equal(groupid(wlddev$country), qG(wlddev$country, na.exclude = FALSE))

## When data is unordered, group-id can be generated through an ordering..
uo <- order(rnorm(fnrow(airquality)))
monthuo <- airquality$Month[uo]
o <- order(monthuo)
groupid(monthuo, o)
identical(groupid(monthuo, o)[o], unattrib(groupid(airquality$Month)))
```

GRP *Fast Grouping /* collapse *Grouping Objects*

Description

GRP performs fast, ordered and unordered, groupings of vectors and data.frames (or lists of vectors) using radixorderv. The output is a list-like object of class 'GRP' which can be printed, plotted and used as an efficient input to all of *collapse*'s fast functions, operators, as well as collap, BY and TRA.

fgroup_by is similar to dplyr::group_by but faster. It creates a 'grouped_df', but with a 'GRP' object attached - for faster dplyr-like programming with *collapse*'s fast functions.

There are also several conversion methods to convert to and from 'GRP' objects. The most important of these is GRP.grouped_df, which returns a 'GRP' object from a 'grouped_df' created with fgroup_by or dplyr::group_by.

Usage

```
GRP(X, ...)

## Default S3 method:
GRP(X, by = NULL, sort = TRUE, decreasing = FALSE, na.last = TRUE,
    return.groups = TRUE, return.order = FALSE, ...)

## S3 method for class 'factor'
GRP(X, ...)
## S3 method for class 'qG'
GRP(X, ...)
## S3 method for class 'pseries'
GRP(X, effect = 1L, ...)
## S3 method for class 'pdata.frame'
GRP(X, effect = 1L, ...)
## S3 method for class 'grouped_df'
```

```
GRP(X, ...)

# Identify, get group names, and convert GRP object to factor
is.GRP(x)
group_names.GRP(x, force.char = TRUE)
as.factor.GRP(x, ordered = FALSE)

# Fast version of dplyr::group_by for use with fast functions, see details
fgroup_by(X, ..., sort = TRUE, decreasing = FALSE, na.last = TRUE, return.order = FALSE)

# This gets grouping columns from a grouped_df created with dplyr::group_by or fgroup_by
fgroup_vars(X, return = "data")

## S3 method for class 'GRP'
print(x, n = 6, ...)

## S3 method for class 'GRP'
plot(x, breaks = "auto", type = "s", horizontal = FALSE, ...)
```

Arguments

X	a vector, list of columns or data.frame (default method), or a classed object (conversion/extractor methods).
x	a GRP object.
by	if X is a data.frame or list, by can indicate columns to use for the grouping (by default all columns are used). Columns must be passed using a vector of column names, indices, or using a one-sided formula i.e. ~ col1 + col2.
sort	logical. This argument only affects character vectors / columns passed. If FALSE, these are not ordered but simply grouped in the order of first appearance of unique elements. This provides a slight performance gain if only grouping but not alphabetic ordering is required (argument passed to radixorderv).
ordered	logical. TRUE adds a class 'ordered' i.e. generates an ordered factor.
decreasing	logical. Should the sort order be increasing or decreasing? Can be a vector of length equal to the number of arguments in X / by (argument passed to radixorderv).
na.last	logical. if missing values are encountered in grouping vector/columns, assign them to the last group (argument passed to radixorderv).
return.groups	logical. include the unique groups in the created 'GRP' object.
return.order	logical. include the output from radixorderv in the created 'GRP' object.
force.char	logical. Always output group names as character vector, even if a single numeric vector was passed to GRP.default.
effect	*plm* methods: Select which panel identifier should be used as grouping variable. 1L means first variable in the plm::index, 2L the second etc.. More than one variable can be supplied.
return	an integer or string specifying what fgroup_vars should return. The options are:

Int.	*String*	*Description*
1	"data"	full grouping columns (default)
2	"unique"	unique rows of grouping columns
3	"names"	names of grouping columns
4	"indices"	integer indices of grouping columns
5	"named_indices"	named integer indices of grouping columns
6	"logical"	logical selection vector of grouping columns
7	"named_logical"	named logical selection vector of grouping columns

`n`	integer. Number of groups to print out.
`breaks`	integer. Number of breaks in the histogram of group-sizes.
`type`	linetype for plot.
`horizontal`	logical. `TRUE` arranges plots next to each other, instead of above each other.
`...`	for `fgroup_by`: unquoted comma-separated column names of grouping columns. Otherwise: arguments to be passed to or from other methods.

Details

`GRP` is a central function in the *collapse* package because it provides the key inputs to facilitate easy and efficient groupwise-programming at the `C/C++` level: Information about (1) the number of groups (2) an integer group-id indicating which values / rows belong to which group and (3) information about the size of each group. Provided with these informations, *collapse*'s Fast Statistical Functions pre-allocate intermediate and result vectors of the right sizes and (in most cases) perform grouped statistical computations in a single pass through the data.

The sorting and ordering functionality for `GRP` only affects (2), that is groups receive different integer-id's depending on whether the groups are sorted `sort = TRUE`, and in which order (argument `decreasing`). This in-turn changes the order of values/rows in the output of *collapse* functions. *Note* that `sort = FALSE` is only effective on character vectors. Numeric grouping vectors will always produce ordered groupings.

Next to group, there is the function `fgroup_by` as a significantly faster alternative to `dplyr::group_by`. It creates a grouped tibble by attaching a 'GRP' object to a data frame. *collapse* functions with a grouped_df method applied to that data frame will yield grouped computations. Note that `fgroup_by` can only be used in combination with *collapse* functions, not with *dplyr* verbs such as `summarize` or `mutate`.

`GRP` is an S3 generic function with one default method supporting vector and list input and several conversion methods:

The conversion of factors to 'GRP' objects by `GRP.factor` involves obtaining the number of groups calling `ng <-flevels(f)` and then computing the count of each level using `tabulate(f,ng)`. The integer group-id (2) is already given by the factor itself after removing the levels and class attributes and replacing any missing values with `ng + 1L`. The levels are put in a list and moved to position (4) in the 'GRP' object, which is reserved for the unique groups. Going from factor to 'GRP' object thus only requires a tabulation of the levels, whereas creating a factor from a 'GRP' object using `as.factor.GRP` does not involve any computations, but may involve interactions if multiple grouping columns were used (which are then interacted to produce unique factor levels) or `as.character` conversions if the grouping column(s) were numeric (which are potentially expensive).

The method GRP.grouped_df takes the 'groups' attribute from a grouped tibble and converts it to a 'GRP' object. If the grouped tibble was generated using fgroup_by, all work is done already. If it was created using dplyr::group_by, a C++ routine is called to efficiently convert the grouping object.

Note: For faster factor generation and a factor-light class 'qG' which avoids the coercion of factor levels to character also see qF and qG.

Value

A list-like object of class 'GRP' containing information about the number of groups, the observations (rows) belonging to each group, the size of each group, the unique group names / definitions, whether the groups are ordered or not and (optionally) the ordering vector used to perform the ordering. The object is structured as follows:

List-index	*Element-name*	*Content type*	*Content description*
[[1]]	N.groups	integer(1)	Number of Groups
[[2]]	group.id	integer(NROW(X))	An integer group-identifier
[[3]]	group.sizes	integer(N.groups)	Vector of group sizes
[[4]]	groups	unique(X) or NULL	Unique groups (same format as input, sorted if sort =
[[5]]	group.vars	character	The names of the grouping variables
[[6]]	ordered	logical(2)	[1]- TRUE if sort = TRUE, [2]- TRUE if X already sor
[[7]]	order	integer(NROW(X)) or NULL	Ordering vector from radixorderv or NULL if returr
[[8]]	call	call	The GRP() call, obtained from match.call()

See Also

qF, qG, finteraction, Collapse Overview

Examples

```
## default method
GRP(mtcars$cyl)
GRP(mtcars, ~ cyl + vs + am)    # or GRP(mtcars, c("cyl","vs","am")) or GRP(mtcars, c(2,8:9))
g <- GRP(mtcars, ~ cyl + vs + am) # saving the object
plot(g)                           # plotting it
group_names.GRP(g)                # retain group names
fsum(mtcars, g)             # compute the sum of mtcars, grouped by variables cyl, vs and am.

## convert factor to GRP object
GRP(iris$Species)

## dplyr integration
library(dplyr)
mtcars %>% group_by(cyl,vs,am) %>% GRP    # get GRP object from a dplyr grouped tibble
mtcars %>% group_by(cyl,vs,am) %>% fmean  # grouped mean using dplyr grouping
mtcars %>% fgroup_by(cyl,vs,am) %>% fmean # faster alternative with collapse grouping
```

`is.regular-is.unlistable` *Regular Objects and Unlistable Lists*

Description

A regular R object is an R object that is either atomic or a list - checked with `is.regular`. A (nested) list composed of regular objects at each level is unlistable - checked with `is.unlistable`.

Usage

```
is.regular(x)
is.unlistable(l)
```

Arguments

`x`	a R object.
`l`	a list.

Details

`is.regular` is simply defined as `is.atomic(x) || is.list(x)`. `is.unlistable` is defined as `all(unlist(rapply2d(l,is.regular),use.names = FALSE))`. It could of course also be defined as `all(rapply(l,is.atomic))`, but the above is a lot more efficient if `l` contains `data.frame`'s.

Value

`logical(1)` - `TRUE` or `FALSE`.

See Also

`ldepth`, `has_elem`, List Processing, Collapse Overview

Examples

```
is.regular(list(1,2))
is.regular(2)
is.regular(a ~ c)
l <- list(1, 2, list(3, 4, "b", FALSE))
is.regular(l)
is.unlistable(l)
l <- list(1, 2, list(3, 4, "b", FALSE, e ~ b))
is.regular(l)
is.unlistable(l)
```

ldepth *Determine the Depth / Level of Nesting of a List*

Description

ldepth provides the depth of a list or list-like structure.

Usage

```
ldepth(l, DF.as.list = TRUE)
```

Arguments

l	a list.
DF.as.list	treat data.frame's as sub-lists?

Details

The depth or level or nesting of a list or list-like structure (i.e. a classed object) is found by recursing down to the bottom of the list and adding an integer count of 1 for each level passed. For example the depth of a data.frame is 1. If a data.frame has list-columns, the depth is 2. However for reasons of efficiency, if l is not a data.frame and DF.as.list = TRUE, data.frame's found inside l will not be checked for list column's but assumed to have a depth of 1.

Value

A single integer indicating the depth of the list.

See Also

is.unlistable, has_elem, List Processing, Collapse Overview

Examples

```
l = list(1, 2)
ldepth(l)
l = list(1, 2, mtcars)
ldepth(l)
ldepth(l, DF.as.list = FALSE)
l = list(1, 2, list(4, 5, list(6, mtcars)))
ldepth(l)
ldepth(l, DF.as.list = FALSE)
```

psacf *Auto- and Cross- Covariance and -Correlation Function Estimation for Panel-Series*

Description

psacf, pspacf and psccf compute (and by default plot) estimates of the auto-, partial auto- and cross- correlation or covariance functions for panel-vectors and plm::pseries. They are analogues to stats::acf, stats::pacf and stats::ccf.

Usage

```
psacf(x, ...)
pspacf(x, ...)
psccf(x, y, ...)

## Default S3 method:
psacf(x, g, t = NULL, lag.max = NULL, type = c("correlation", "covariance","partial"),
      plot = TRUE, gscale = TRUE, ...)
## Default S3 method:
pspacf(x, g, t = NULL, lag.max = NULL, plot = TRUE, gscale = TRUE, ...)
## Default S3 method:
psccf(x, y, g, t = NULL, lag.max = NULL, type = c("correlation", "covariance"),
      plot = TRUE, gscale = TRUE, ...)

## S3 method for class 'pseries'
psacf(x, lag.max = NULL, type = c("correlation", "covariance","partial"),
      plot = TRUE, gscale = TRUE, ...)
## S3 method for class 'pseries'
pspacf(x, lag.max = NULL, plot = TRUE, gscale = TRUE, ...)
## S3 method for class 'pseries'
psccf(x, y, lag.max = NULL, type = c("correlation", "covariance"),
      plot = TRUE, gscale = TRUE, ...)

## S3 method for class 'data.frame'
psacf(x, by, t = NULL, cols = is.numeric, lag.max = NULL,
    type = c("correlation", "covariance","partial"), plot = TRUE, gscale = TRUE, ...)
## S3 method for class 'data.frame'
pspacf(x, by, t = NULL, cols = is.numeric, lag.max = NULL,
       plot = TRUE, gscale = TRUE, ...)

 ## S3 method for class 'pdata.frame'
psacf(x, cols = is.numeric, lag.max = NULL,
    type = c("correlation", "covariance","partial"), plot = TRUE, gscale = TRUE, ...)
## S3 method for class 'pdata.frame'
pspacf(x, cols = is.numeric, lag.max = NULL, plot = TRUE, gscale = TRUE, ...)
```

Arguments

x, y	a numeric vector, panel-series (plm::pseries), data.frame or panel-data-frame (plm::pdata.frame).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x,y.
by	*data frame method*: Same input as g, but also allows one- or two-sided formulas using the variables in x, i.e. ~ idvar or var1 + var2 ~ idvar1 + idvar2.
t	same input as g, to indicate the time-variable. For secure computations on unordered panel-vectors. Data frame method also takes one-sided formula i.e. ~time.
cols	*data.frame method*: Select columns using a function, column names or indices. *Note*: cols is ignored if a two-sided formula is passed to by.
lag.max	maximum lag at which to calculate the acf. Default is 2*sqrt(length(x)/ng) where ng is the number of groups in the panel-series / supplied to g.
type	character string giving the type of acf to be computed. Allowed values are "correlation" (the default), "covariance" or "partial".
plot	logical. If TRUE (the default) the acf is plotted.
gscale	logical. Do a groupwise scaling / standardization of x,y (using collapse::fscale and the groups supplied to g) before computing panel-autocovariances / correlations.
...	further arguments to be passed to stats:::plot.acf.

Details

If gscale = TRUE data are standardized within each group (using collapse::fscale) such that the group-mean is 0 and the group-standard deviation is 1. This is strongly recommended for most panels to get rid of individual-specific heterogeneity which would corrupt the ACF computations.

After scaling, psacf, pspacf and psccf compute the ACF/CCF by creating a matrix of panel-lags of the series using collapse::flag and then correlating this matrix with the series (x,y) using stats::cor and pairwise-complete observations. This may require a lot of memory on large data, but is done because passing a sequence of lags to collapse::flag and thus calling collapse::flag and stats::cor one time is much faster than calling them lag.max times. The partial ACF is computed from the ACF in the same way as in stats::pacf.

Value

An object of class "acf", see ?stats::acf. The result is returned invisibly if plot is TRUE.

Note

For plm::pseries and plm::pdata.frame, the first index variable is taken to be the group-id and the second the time variable. If more than 2 index variables are attached to plm::pseries, the last one is taken as the time variable and the others are taken as group-id's and interacted.

The pdata.frame method only works for properly subsetted objects of class 'pdata.frame'. A list of 'pseries' won't work.

See Also

Time-Series and Panel-Series, Collapse Overview

Examples

```
## World Development Panel Data
head(wlddev)                                                    # see also help(wlddev)
psacf(wlddev$PCGDP, wlddev$country, wlddev$year)                # ACF of GDP per Capita
psacf(wlddev, PCGDP ~ country, ~year)                         # Same using data.frame method
psacf(wlddev$PCGDP, wlddev$country)                          # The Data is sorted, can omit t
pspacf(wlddev$PCGDP, wlddev$country)                            # Partial ACF
psccf(wlddev$PCGDP, wlddev$LIFEEX, wlddev$country)           # CCF with Life-Expectancy at Birth

psacf(wlddev, PCGDP + LIFEEX + ODA ~ country, ~year)         # ACF and CCF of GDP, LIFEEX and ODA
psacf(wlddev, ~ country, ~year, c(9:10,12))                     # Same, using cols argument
pspacf(wlddev, ~ country, ~year, c(9:10,12))                    # Partial ACF

## Using plm:
pwlddev <- plm::pdata.frame(wlddev, index = c("country","year"))# Creating a Panel-Data Frame
PCGDP <- pwlddev$PCGDP                                        # Panel-Series of GDP per Capita
LIFEEX <- pwlddev$LIFEEX                                     # Panel-Series of Life Expectancy
psacf(PCGDP)                                                # Same as above, more parsimonious
pspacf(PCGDP)
psccf(PCGDP, LIFEEX)
psacf(pwlddev[c(9:10,12)])
pspacf(pwlddev[c(9:10,12)])
```

---

psmat *Matrix / Array from Panel-Series*

Description

`psmat` efficiently expands a panel-vector or `plm::pseries` into a matrix. If a data frame or `plm::pdata.frame` is passed, `psmat` returns (default) a 3D array or a list of such matrices.

Usage

```
psmat(x, ...)

## Default S3 method:
psmat(x, g, t = NULL, transpose = FALSE, ...)

## S3 method for class 'pseries'
psmat(x, transpose = FALSE, ...)

## S3 method for class 'data.frame'
psmat(x, by, t = NULL, cols = NULL, transpose = FALSE, array = TRUE, ...)
```

```
## S3 method for class 'pdata.frame'
psmat(x, cols = NULL, transpose = FALSE, array = TRUE, ...)


## S3 method for class 'psmat'
plot(x, legend = FALSE, colours = legend, labs = NULL, ...)
```

Arguments

x	a vector, panel-series (plm::pseries), data.frame or panel-data.frame (plm::pdata.frame).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x. If the panel is balanced an integer indicating the number of groups can also be supplied. See Examples.
by	*data frame method*: Same input as g, but also allows one- or two-sided formulas using the variables in x, i.e. ~ idvar or var1 + var2 ~ idvar1 + idvar2.
t	same inputs as g, to indicate the time-variable or second identifier(s). g and t together should fully identify the panel. If t = NULL, the data is assumed sorted and seq_col is used to generate rownames.
cols	*data.frame method*: Select columns using a function, column names or indices. *Note*: cols is ignored if a two-sided formula is passed to by.
transpose	logical. TRUE generates the matrix such that g/by -> columns, t -> rows. Default is g/by -> rows, t -> columns.
array	*data.frame / pdata.frame methods*: logical. TRUE returns a 3D array (if just one column is selected a matrix is returned). Otherwise always return a list of matrices.
...	arguments to be passed to or from other methods, or for the plot method additional arguments passed to ts.plot.
legend	logical. Automatically create a legend of panel-groups.
colours	logical. Automatically colour by panel-groups.
labs	provide a character-vector of variable labels / series titles when plotting an array.

Details

For plm::pseries, the first index variable is taken to be the group-id and the second the time variable. If more than 2 index variables are attached to plm::pseries, the last one is taken as the time variable and the others are taken as group-id's and interacted.

Value

a matrix or 3D array containing the data in x, where by default the rows constitute the groups-ids (g/by) and the columns the time variable or individual ids (t). 3D arrays contain the variables in the 3rd dimension. The objects have a class 'psmat', and also a 'transpose' attribute indicating whether transpose = TRUE or transpose = FALSE.

Note

The `pdata.frame` method only works for properly subsetted objects of class 'pdata.frame'. A list of 'pseries' won't work. There also exist simple `aperm` and `[` (subset) methods for 'psmat' objects. These differ from the default methods only by keeping the class and the 'transpose' attribute.

See Also

Time-Series and Panel-Series, Collapse Overview

Examples

```
## World Development Panel Data
head(wlddev)                                                    # View data
qsu(wlddev, pid = ~ iso3c, cols = 9:12, vlabels = TRUE)         # Sumarizing data
str(psmat(wlddev$PCGDP, wlddev$iso3c, wlddev$year))            # Generating matrix of GDP
r <- psmat(wlddev, PCGDP ~ iso3c, ~ year)                 # Same thing using data.frame method
plot(r, main = vlabels(wlddev)[9], xlab = "Year")               # Plot the matrix
str(r)                                                          # See srructure
str(psmat(wlddev$PCGDP, wlddev$iso3c))                   # The Data is sorted, could omit t
str(psmat(wlddev$PCGDP, 216))                             # This panel is also balanced, so
# ..indicating the number of groups would be sufficient to obtain a matrix

ar <- psmat(wlddev, ~ iso3c, ~ year, 9:12)                # Get array of transposed matrices
str(ar)
plot(ar)
plot(ar, legend = TRUE)
plot(psmat(collap(wlddev, ~region+year, cols = 9:12),        # More legible and fancy plot
           ~region, ~year), legend = TRUE,
     labs = vlabels(wlddev)[9:12])

psml <- psmat(wlddev, ~ iso3c, ~ year, 9:12, array = FALSE)    # This gives list of ps-matrices
head(unlist2d(psml, "Variable", "Country", id.factor = TRUE))  # Using unlist2d, can generate DF

## Using plm simplifies things
pwlddev <- plm::pdata.frame(wlddev, index = c("iso3c","year"))  # Creating a Panel-Data Frame
PCGDP <- pwlddev$PCGDP                                    # A panel-Series of GDP per Capita
psmat(PCGDP)                                              # Same as above, more parsimonious
plot(psmat(PCGDP))
plot(psmat(pwlddev[9:12]))
plot(psmat(G(pwlddev[9:12])))                            # Here plotting panel- growth rates
```

`pwcor, pwcov, pwNobs` *Pairwise Correlations, Covariances and Observation Count*

Description

Computes pairwise Pearsons correlations, covariances and observation counts. Pairwise correlations and covariances can be computed together with observation counts and p-values, and output

as 3D array (default) or list of matrices. For an equivalent and faster implementation of pwcor see Hmisc::rcorr (written in Fortran). A major feature of pwcor and pwcov is their sophisticated print method.

Usage

```
pwcor(X, ..., N = FALSE, P = FALSE, array = TRUE, use = "pairwise.complete.obs")

pwcov(X, ..., N = FALSE, P = FALSE, array = TRUE, use = "pairwise.complete.obs")

pwNobs(X)

## S3 method for class 'pwcor'
print(x, digits = 2L, sig.level = 0.05, show = c("all","lower.tri","upper.tri"),
      spacing = 1L, ...)

## S3 method for class 'pwcov'
print(x, digits = 2L, sig.level = 0.05, show = c("all","lower.tri","upper.tri"),
      spacing = 1L, ...)
```

Arguments

X	a matrix or data.frame, for pwcor and pwcov all columns must be numeric.
x	an object of class 'pwcor' / 'pwcov'.
N	logical. TRUE also computes pairwise observation counts.
P	logical. TRUE also computes pairwise p-values (same as cor.test).
array	logical. If N = TRUE or P = TRUE, TRUE (default) returns output as 3D array whereas FALSE returns a list of matrices.
use	argument passed to cor / cov.
digits	integer. The number of digits to round to in print.
sig.level	numeric. P-value threshold below which a '*' is displayed above significant coefficients if P = TRUE.
show	character. The part of the correlation / covariance matrix to display.
spacing	integer. Controls the spacing between different reported quantities in the print-out of the matrix: 0 - compressed, 1 - single space, 2 - double space.
...	other arguments passed to cor or cov. Only sensible if P = FALSE.

Value

a numeric matrix, 3D array or list of matrices of the computed statistics. For pwcor and pwcov the object has a class 'pwcor' and 'pwcov', respectively.

See Also

qsu, Collapse Overview

Examples

```
mna <- na_insert(mtcars)
pwcor(mna)
pwcov(mna)
pwNobs(mna)
pwcor(mna, N = TRUE)
pwcor(mna, P = TRUE)
pwcor(mna, N = TRUE, P = TRUE)
aperm(pwcor(mna, N = TRUE, P = TRUE))
print(pwcor(mna, N = TRUE, P = TRUE), digits = 3, sig.level = 0.01, show = "lower.tri")
pwcor(mna, N = TRUE, P = TRUE, array = FALSE)
print(pwcor(mna, N = TRUE, P = TRUE, array = FALSE), show = "lower.tri")
```

---

qF *Fast Factor Generation and Vector Grouping*

Description

qF, shorthand for 'quick-factor' implements very fast (ordered) factor generation from atomic vectors using either radix ordering or index hashing.

qG, shorthand for 'quick-group', generates a kind of factor-light without the levels attribute but instead an attribute providing the number of levels. Optionally the levels / groups can be attached, but without converting them to character. Objects have a class 'qG'.

finteraction generates a factor by interacting multiple vectors or factors. In that process missing values are always replaced with a level and unused levels are always dropped.

Usage

```
qF(x, ordered = FALSE, na.exclude = TRUE, sort = TRUE,
   method = c("auto", "radix", "hash"))

qG(x, ordered = FALSE, na.exclude = TRUE, sort = TRUE,
   return.groups = FALSE, method = c("auto", "radix", "hash"))

is.qG(x)

finteraction(..., ordered = FALSE, sort = TRUE)
```

Arguments

x	a atomic vector, factor or quick-group.
ordered	logical. Adds a class 'ordered'.
na.exclude	logical. TRUE preserves missing values (i.e. NA level is generated).
sort	logical. TRUE sorts the levels.
method	an integer or character string specifying the method of computation:

Int.	*String*	*Description*
1	"auto"	automatic selection: hash for character, logical or if `length(x) < 500`, else radix.
2	"radix"	use radix ordering to generate factors. See Details.
3	"hash"	use index hashing to generate factors. See Details.

`return.groups` logical. `TRUE` returns the unique elements / groups / levels of `x` in an attribute called 'groups'. Unlike `qF`, they are not converted to character.

`...` multiple atomic vectors or factors, or a single list of equal-length vectors or factors. See Details.

Details

These functions are quite important. Whenever a vector is passed to a *collapse* function such as `fmean(mtcars,mtcars$cyl)`, is is grouped using `qF` or `qG`.

`qF` is a combination of `as.factor` and `factor`. Applying it to a vector i.e. `qF(x)` gives the same result as `as.factor(x)`. `qF(x,ordered = TRUE)` generates and ordered factor (same as `factor(x,ordered = TRUE)`), and `qF(x,na.exclude = FALSE)` generates a level for missing values (same as `factor(x,exclude = NULL)`). An important addition is that `qF(x,na.exclude = FALSE)` also adds a class 'na.included'. This prevents *collapse* functions from checking missing values in the factor, and is thus computationally more efficient. Thus factors used in grouped operations should always be generated using `qF(x,na.exclude = FALSE)`. Setting `sort = FALSE` gathers the levels in a random order (unless `method = "radix"` and `x` is numeric, in which case the levels are always sorted). This can provide a speed improvement for non-numeric `x`.

There are two methods of computation: radix ordering and index hashing. Radix ordering is done through combining the functions `radixorder` and `groupid`. It is generally faster than index hashing for large numeric data (although there are exceptions). Index hashing is done using `Rcpp::sugar::sort_unique` and `Rcpp::sugar::match`. It is generally faster for character data. For logical data, a super fast one-pass method was written which is subsumed in the hash method. Regarding speed: In general `qF` is around 5x faster than `as.factor` on character data and about 30x faster on numeric data. Automatic method dispatch typically does a good job delivering optimal performance.

`qG` is in the first place a programmers function. It generates a factor-'light' consisting of only an integer grouping vector and an attribute providing the number of groups. It is faster and more memory efficient than `GRP` for grouping atomic vectors, which is the main reason it exists. The fact that it (optionally) returns the unique groups / levels without converting them to character is an added bonus (this also provides a small performance gain compared to `qF`).

`finteraction` is simply a wrapper around `as.factor.GRP(GRP.default(X,sort = TRUE))`, where X is replaced by the arguments in '...' combined in a list. See `GRP` for computational details. In general: All vectors, factors, or lists of vectors / factors passed can be interacted. Interactions always create a level for missing values and always drop any unused levels.

Value

`qF` returns an (ordered) factor. `qG` returns an object of class 'qG': an integer grouping vector with an attribute 'N.groups' indicating the number of groups, and, if `return.groups = TRUE`, an attribute 'groups' containing the vector of unique groups / elements in `x` corresponding to the integer-id.

Note

Neither qF nor qG can reorder groups / factor levels. These objects can however be converted into one another using qF/qG, and it is also possible to add a class 'ordered' (ordered = TRUE) and to create am extra level / integer for missing values (na.exclude = FALSE).

See Also

groupid, GRP, Fast (Ordered) Grouping, Collapse Overview

Examples

```
cylF <- qF(mtcars$cyl)     # Factor from atomic vector
cylG <- qG(mtcars$cyl)     # Quick-group from atomic vector
cylG                       # See the simple structure of this object

cf  <- qF(wlddev$country)  # Bigger data
cf2 <- qF(wlddev$country, na.exclude = FALSE)  # With na.included class
dat <- num_vars(wlddev)

# cf2 is faster in grouped operations because no missing value check is performed
library(microbenchmark)
microbenchmark(fmax(dat, cf), fmax(dat, cf2))

finteraction(mtcars$cyl, mtcars$vs)  # Interacting two variables (can be factors)
finteraction(mtcars)                 # A more crude example...
```

---

qsu *Fast (Grouped, Weighted) Summary Statistics for Cross-Sectional and Panel-Data*

Description

qsu, shorthand for quick-summary, is an extremely fast summary command inspired by the (xt)summarize command in the STATA statistical software.

It computes a set of 7 statistics (nobs, mean, sd, min, max, skewness and kurtosis) using a numerically stable one-pass method generalized from Welford's Algorithm. Statistics can be computed weighted, by groups, and also within-and between entities (for panel-data, see Details).

Usage

```
## Default S3 method:
qsu(x, g = NULL, pid = NULL, w = NULL, higher = FALSE, array = TRUE, ...)

## S3 method for class 'matrix'
qsu(x, g = NULL, pid = NULL, w = NULL, higher = FALSE, array = TRUE, ...)

## S3 method for class 'data.frame'
```

```
qsu(x, by = NULL, pid = NULL, w = NULL, cols = NULL,
    higher = FALSE, array = TRUE, vlabels = FALSE,...)

# Methods for compatibility with plm:

## S3 method for class 'pseries'
qsu(x, g = NULL, w = NULL, effect = 1L, higher = FALSE, array = TRUE, ...)

## S3 method for class 'pdata.frame'
qsu(x, by = NULL, w = NULL, cols = NULL, effect = 1L,
    higher = FALSE, array = TRUE, vlabels = FALSE, ...)


## S3 method for class 'qsu'
print(x, digits = 2, nonsci.digits = 9, na.print = "-",
      return = FALSE, print.gap = 2, ...)
```

Arguments

x	a numeric vector, matrix, data.frame, panel-series (plm::pseries) or panel-data.frame (plm::pdata.frame)
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
by	*(p)data.frame method*: Same as g, but also allows one- or two-sided formulas i.e. ~ group1 + group2 or var1 + var2 ~ group1 + group2. See Examples.
pid	same input as g/by: Specify a panel-identifier to also compute statistics on between- and within- transformed data. data.frame method also supports one- or two-sided formulas. Transformations are taken independently from grouping with g/by (grouped statistics are computed on the transformed data). However, passing any LHS variables to pid will overwrite any LHS variables passed to by.
w	a vector of (non-negative) weights. Adding weights will compute the weighted mean, sd, skewness and kurtosis, and transform the data using weighted individual means if pid is used.
cols	select columns to summarize using column names, indices or a function (i.e. is.numeric). Two-sided formulas passed to by or pid overwrite cols.
higher	logical. Add higher moments (skewness and kurtosis).
array	logical. If computations have more than 2 dimensions (up to a maximum of 4D: variables, statistics, groups and panel-decomposition) output to array, else output (nested) list of matrices.
vlabels	logical. Use variable labels in the summary. See vlabels.
effect	*plm* methods: Select which panel identifier should be used for between and within transformations of the data. 1L means first variable in the plm::index, 2L the second etc.. More than one variable can be supplied.
...	arguments to be passed to or from other methods.
digits	the number of digits to print after the comma/dot.

nonsci.digits	the number of digits to print before resorting to scientific notation (default is to print out numbers with up to 9 digits and print larger numbers scientifically).
na.print	character string to substitute for missing values.
return	logical. Don't print but instead return the formatted object.
print.gap	integer. Spacing between printed columns. Passed to print.default.

Details

The algorithm used to compute statistics is well described here (see sections *Welford's online algorithm*, *Weighted incremental algorithm* and *Higher-order statistics*. Skewness and kurtosis are calculated as described in Higher-order statistics and are mathematically identical to those implemented in the moments package. Just note that qsu computes the kurtosis (like momens::kurtosis), not the excess-kurtosis (= kurtosis - 3) defined in Higher-order statistics. The *Weighted incremental algorithm* described can easily be generalized to higher-order statistics).

Grouped computations specified with g/by are carried out extremely efficiently as in fsum (in a single pass, without splitting the data).

If pid is used, qsu performs a panel-decomposition of each variable and computes 3 sets of statistics: Statistics computed on the 'Overall' (raw) data, statistics computed on the 'Between' - transformed (pid - averaged) data, and statistics computed on the 'Within' - transformed (pid - demeaned) data.

More formally, let x (bold) be a panel vector of data for N individuals indexed by i, recorded for T periods, indexed by t. xit then denotes a single data-point belonging to individual i in time-period t (t/T must not represent time). Then xi. denotes the average of all values for individual i (averaged over t), and by extension xN. is the vector (length N) of such averages for all individuals. If no groups are supplied to g/by, the 'Between' statistics are computed on xN., the vector of individual averages. (This means that for a non-balanced panel or in the presence of missing values, the 'Overall' mean computed on x can be slightly different than the 'Between' mean computed on xN.). If groups are supplied to g/by, xN. is expanded to the vector xi. (length N x T) by replacing each value xit in x with xi., while preserving missing values in x. Grouped Between-statistics are then computed on xi., with the only difference that the number ob observations ('Between-N') reported for each group is the number of distinct non-missing values of xi. in each group (not the total number of non-missing values of xi. in each group, which is already reported in 'Overall-N').

'Within' statistics are always computed on the vector x - xi. + x.., where x.. is simply the 'Overall' mean computed from x, which is added back to preserve the level of the data. The 'Within' mean computed on this data will always be identical to the 'Overall' mean. In the summary output, qsu reports not 'N', which would be identical to the 'Overall-N', but 'T', the average number of time-periods of data available for each individual obtained as 'T' = 'Overall-N / 'Between-N'. See Examples.

Apart from 'N/T' and the extrema, the standard-deviations ('SD') computed on between- and within- transformed data are extremely valuable because they indicate how much of the variation in a panel-variable is between-individuals and how much of the variation is within-individuals (over time). At the extremes, variables that have common values across individuals (such as the time-variable 't' in a balanced panel), can readily be identified as individual-invariant because the 'Between-SD' on this variable is 0 and the 'Within-SD' is equal to the 'Overall-SD'. Analogous, time-invariant individual characteristics (such as the individual-id 'i') have a 0 'Within-SD' and a 'Between-SD' equal to the 'Overall-SD'.

qsu comes with it's own print method which by default writes out up to 9 digits at 2 decimal places. Larger numbers are printed in scientific format. for numbers between 7 and 9 digits, a comma ',' is placed after the 6th digit to designate the millions. Missing values are printed using '-'.

Value

A matrix, array or list of matrices of summary statistics. All matrices and arrays have a class 'qsu' and a class 'table' attached, responding i.e. to `print.qsu` and `aperm.table`...

Note

If weights `w` are used together with `pid`, transformed data is computed using weighted individual means i.e. weighted `xi.` and weighted `x...` Weighted statistics are subsequently computed on this weighted-transformed data.

See Also

`descr`, `pwcor`, Fast Statistical Functions, Collapse Overview

Examples

```
## World Development Panel Data
# Simple Summaries -------------------------
qsu(wlddev)                                 # Simple summary
qsu(wlddev, vlabels = TRUE)                 # Display variable labels
qsu(wlddev, higher = TRUE)                  # Add skewness and kurtosis

# Grouped Summaries ------------------------
qsu(wlddev, ~ region, vlabels = TRUE)       # Statistics by World Bank Region
qsu(wlddev, PCGDP + LIFEEX ~ income)      # Summarize GDP per Capita and Life Expectancy by
stats <- qsu(wlddev, ~ region + income,     # World Bank Income Level
             cols = 9:10, higher = TRUE)    # Same variables, by both region and income
aperm(stats)                                # A different perspective on the same stats

# Panel-Data Summaries ---------------------
qsu(wlddev, pid = ~ iso3c, vlabels = TRUE)  # Adding between and within countries statistics
# -> They show amongst other things that year and decade are individual-invariant,
# that we have GINI-data on only 161 countries, with only 8.42 observations per country on average,
# and that GDP, LIFEEX and GINI vary more between-countries, but ODA received varies more within
# countries over time.

# Using plm:
pwlddev <- plm::pdata.frame(wlddev,      # Creating a Panel-Data Frame frame from this data
                  index = c("iso3c","year"))
qsu(pwlddev)                          # Summary for pdata.frame -> qsu(wlddev, pid = ~ iso3c)
qsu(pwlddev$PCGDP)                       # Default summary for Panel-Series (class pseries)
qsu(G(pwlddev$PCGDP))                        # Summarizing GDP growth, see also ?G

# Grouped Panel-Data Summaries -------------
qsu(wlddev, ~ region, ~ iso3c, cols = 9:12) # Panel-Statistics by region
psr <- qsu(pwlddev, ~ region, cols = 9:12)  # Same on plm pdata.frame
```

```
psr                                          # -> Gives a 4D array
print.qsu(psr[,"N/T",,])                     # Checking out the number of observations:
# In North america we only have 3 countries, for the GINI we only have 3.91 observations on average
# for 45 Sub-Saharan-African countries, etc...
print.qsu(psr[,"SD",,])                      # Considering only standard deviations
# -> In all regions variations in inequality (GINI) between countries are greater than variations
# in inequality within countries. The opposite is true for Life-Expectancy in all regions apart
# from Europe, etc...
psrl <- qsu(wlddev, ~ region, ~ iso3c,       # Same, but output as nested list
            cols = 9:12, array = FALSE)
psrl                                    # We can use unlist2d to create a tidy data.frame
head(unlist2d(psrl, c("Variable","Trans"),
                    row.names = "Region"))

# Weighted Summaries -----------------------
n <- nrow(wlddev)
weights <- abs(rnorm(n))                     # Generate random weights
qsu(wlddev, w = weights, higher = TRUE)    # Computed weighted mean, SD, skewness and kurtosis
weightsNA <- weights                  # Weights may contain missing values... inserting 1000
weightsNA[sample.int(n, 1000)] <- NA
qsu(wlddev, w = weightsNA, higher = TRUE)  # But now these values are removed from all variables

# Grouped and panel-summaries can also be weighted in the same manor
```

`radixorder` *Fast Radix-Based Ordering*

Description

A slight modification of `base::order(...,method = "radix")` that is more programmer friendly and, importantly, provides features for ordered grouping of data (similar to `data.table:::forderv` which has more or less the same source code). `radixorderv` is a programmers version directly supporting vector and list input. Apart from added grouping features, the source code and standard functionality is identical to `base::order(...,method = "radix")`.

Usage

```
radixorder(..., na.last = TRUE, decreasing = FALSE, starts = FALSE,
           group.sizes = FALSE, sort = TRUE)

radixorderv(x, na.last = TRUE, decreasing = FALSE, starts = FALSE,
            group.sizes = FALSE, sort = TRUE)
```

Arguments

`...`	comma-separated atomic vectors to order.
`x`	an atomic vector or list of atomic vectors such as a data frame.
`na.last`	for controlling the treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed.

decreasing	logical. Should the sort order be increasing or decreasing? Can be a vector of length equal to the number of arguments in ... / x.
starts	logical. TRUE returns an attribute 'starts' containing the first element of each new group i.e. the row denoting the start of each new group if the data were sorted using the computed ordering vector. See Examples.
group.sizes	logical. TRUE returns an attribute 'group.sizes' containing sizes of each group in the same order as groups are encountered if the data were sorted using the computed ordering vector. See Examples.
sort	logical. This argument only affects character vectors / columns passed. If FALSE, these are not ordered but simply grouped in the order of first appearance of unique elements. This provides a slight performance gain if only grouping but not alphabetic ordering is required.

Value

An integer ordering vector, with attributes if starts = TRUE or group.sizes = TRUE. The attributes are 'starts' giving a vector of group starts in the ordered data, 'group.sizes' giving the vector of group sizes, and always included an attribute 'maxgrpn' providing the size of the largest group, and an attribute 'sorted' indicating whether the input data was already sorted.

See Also

Fast (Ordered) Grouping, Collapse Overview

Examples

```
radixorder(mtcars$mpg)
mtcars[radixorder(mtcars$mpg), ]
radixorder(mtcars$cyl, mtcars$vs)

o <- radixorder(mtcars$cyl, mtcars$vs, starts = TRUE)
st <- attr(o, "starts")
mtcars[o, ]
mtcars[o[st], c("cyl", "vs")]  # Unique groups

# Note that if attr(o, "sorted") == TRUE, then all(o[st] == st)
radixorder(rep(1:3, each = 3), starts = TRUE)

# Group sizes
radixorder(mtcars$cyl, mtcars$vs, group.sizes = TRUE)

# Both
radixorder(mtcars$cyl, mtcars$vs, starts = TRUE, group.sizes = TRUE)
```

rapply2d *Recursively Apply a Function to a List of Data Objects*

Description

rapply2d is a recursive version of lapply with two key differences to rapply: (1) Data frames are considered as final objects, not as (sub-)lists, and (2) the result is never simplified / unlisted.

Usage

```
rapply2d(l, FUN, ...)
```

Arguments

l	a list.
FUN	a function that can be applied to all elements in l.
...	additional elements passed to FUN.

Value

A list of the same structure as l, where FUN was applied to all elements.

See Also

unlist2d, List Processing, Collapse Overview

Examples

```
l <- list(mtcars, list(mtcars, as.matrix(mtcars)))
rapply2d(l, fmean)
unlist2d(rapply2d(l, fmean))
```

select-replace-vars *Fast Select, Replace or Add Data Frame Columns*

Description

Efficiently select and replace (or add) a subset of columns from (to) a data frame. This can be done by data type, or using expressions, column names, indices, logical vectors, selector functions or regular expressions matching column names.

Usage

```
## Select and replace variables, analgous to dplyr::select but significantly faster
fselect(x, ..., return = "data")
fselect(x, ...) <- value
slt(x, ..., return = "data")   # Shortcut for fselect
slt(x, ...) <- value           # Shortcut for fselect<-

## Select and replace columns by names, indices, logical vectors,
## regular expressions or using functions to identify columns

get_vars(x, vars, return = "data",
         regex = FALSE, ...)
get_vars(x, vars, regex = FALSE, ...) <- value
      gv(x, vars, return = "data",              # Shortcut for get_vars
         regex = FALSE, ...)
      gv(x, vars, regex = FALSE, ...) <- value  # Shortcut for get_vars<-

## Add columns at any position within a data.frame

add_vars(x, ..., pos = "end")
add_vars(x, pos = "end") <- value
      av(x, ..., pos = "end")               # Shortcut for add_vars
      av(x, pos = "end") <- value           # Shortcut for add_vars<-

## Select and replace columns by data type

num_vars(x, return = "data")
num_vars(x) <- value
      nv(x, return = "data")       # Shortcut for num_vars
      nv(x) <- value               # Shortcut for num_vars<-
cat_vars(x, return = "data")       # Categorical variables, see is.categorical
cat_vars(x) <- value
char_vars(x, return = "data")
char_vars(x) <- value
fact_vars(x, return = "data")
fact_vars(x) <- value
logi_vars(x, return = "data")
logi_vars(x) <- value
Date_vars(x, return = "data")      # See is.Date
Date_vars(x) <- value
```

Arguments

x	a data.frame.
value	a data.frame or list of columns whose dimensions exactly match those of the extracted subset of x. If only 1 variable is in the subset of x, value can also be an atomic vector or matrix, provided that NROW(value) == nrow(x).

`vars` a vector of column names, indices (can be negative), a suitable logical vector, a vector of regular expressions matching column names (if `regex = TRUE`). It is also possible to pass a function returning `TRUE` or `FALSE` when applied to the columns of `x`.

`return` an integer or string specifying what the selector function should return. The options are:

Int.	*String*	*Description*
1	"data"	subset of data.frame (default)
2	"names"	column names
3	"indices"	column indices
4	"named_indices"	named column indices
5	"logical"	logical selection vector
6	"named_logical"	named logical vector

Note: replacement functions only replace data, However column names are replaced together with the data.

`regex` logical. `TRUE` will do regular expression search on the column names of `x` using a (vector of) regular expression(s) passed to `vars`. Matching is done using `grep`.

`pos` the position where columns are added in the data.frame. `"end"` (default) will append the data.frame at the end (right) side. "front" will add columns in front (left). Alternatively one can pass a vector of positions (matching `length(value)` if value is a list). In that case the other columns will be shifted around the new ones while maintaining their order.

`...` for `fselect`: column names and expressions. for `get_vars`: further arguments passed to `grep`, if `regex = TRUE`. For `add_vars`: Same as `value`. A single argument passed may also be a vector or matrix, multiple arguments must each be a list (they are combined using `c(...)`).

Details

`get_vars(<-)` is around 2x faster than `` `[.data.frame` `` and 8x faster than `` `[<-.data.frame` ``, so the common operation `data[cols] <-someFUN(data[cols])` can be made 10x more efficient (abstracting from computations performed by `someFUN`) using `get_vars(data,cols) <-someFUN(get_vars(data,cols))` or the shorthand `gv(data,cols) <-someFUN(gv(data,cols))`.

Similarly type-wise operations like `data[sapply(data,is.numeric)]` or `data[sapply(data,is.numeric)] <-value` are facilitated and more efficient using `num_vars(data)` and `num_vars(data) <-value` or the shortcuts `nv` and `nv<-` etc.

`fselect` provides an efficient alternative to `dplyr::select`, allowing the selection of variables based on expressions evaluated within the data.frame, see Examples. It is about 100x faster than `dplyr::select` but also more simple as it does not provide special methods for grouped tibbles.

Finally, `add_vars(data1,data2,data3,...)` is a lot faster than `cbind(data1,data2,data3,...)`, and preserves the attributes of `data1` (i.e. it is like adding columns to `data1`). The replacement function `add_vars(data) <-someFUN(get_vars(data,cols))` efficiently appends `data` with computed columns. The `pos` argument allows adding columns at positions other than the end (right) of the data frame, see Examples.

All functions introduced here perform their operations class-independent. They all basically work like this: (1) save the attributes of x, (2) unclass x, (3) subset, replace or append x as a list, (4) modify the "names" component of the attributes of x accordingly and (5) efficiently attach the attributes again to the result from step (3). Thus they can freely be applied to data.table's, grouped tibbles, panel-data frames and other classes and will return an object of exactly the same class and the same attributes.

Note

When lists of unequal-length columns are offered as replacements this yields a malformed data.frame (which will also print a warning in the console i.e. you will notice that). The functions here only check the length of the first column, which is one of the reasons why they are so fast.

See Also

fsubset, ftransform, Data Frame Manipulation, Collapse Overview

Examples

```
## Wold Development Data
head(fselect(wlddev, country, year, PCGDP))                # Fast dplyr-like selecting
head(fselect(wlddev, -country, -year, -PCGDP))
head(fselect(wlddev, country, year, PCGDP:ODA))
head(fselect(wlddev, -(PCGDP:ODA)))
fselect(wlddev, country, year, PCGDP:ODA) <- NULL          # Efficient deleting
head(wlddev)
rm(wlddev)

head(num_vars(wlddev))                                     # Select numeric variables
head(cat_vars(wlddev))                                  # Select categorical (non-numeric) vars
head(get_vars(wlddev, is.categorical))                     # Same thing

num_vars(wlddev) <- num_vars(wlddev)                # Replace Numeric Variables by themselves
get_vars(wlddev,is.numeric) <- get_vars(wlddev,is.numeric) # Same thing

head(get_vars(wlddev, 9:12))                         # Select columns 9 through 12, 2x faster
head(get_vars(wlddev, -(9:12)))                            # All except columns 9 through 12
head(get_vars(wlddev, c("PCGDP","LIFEEX","GINI","ODA")))   # Select using column names
head(get_vars(wlddev, "[[:upper:]]", regex = TRUE))      # Same thing: match upper-case var. names

get_vars(wlddev, 9:12) <- get_vars(wlddev, 9:12)        # 9x faster wlddev[9:12] <- wlddev[9:12]
add_vars(wlddev) <- STD(gv(wlddev,9:12), wlddev$iso3c)    # Add Standardized columns 9 through 12
head(wlddev)                                               # gv and av are shortcuts

get_vars(wlddev, 13:16) <- NULL                     # Efficient Deleting added columns again
av(wlddev, "front") <- STD(gv(wlddev,9:12), wlddev$iso3c)  # Again adding in Front
head(wlddev)
get_vars(wlddev, 1:4) <- NULL                              # Deleting
av(wlddev,c(10,12,14,16)) <- W(wlddev,~iso3c, cols = 9:12, # Adding next to original variables
                               keep.by = FALSE)
head(wlddev)
get_vars(wlddev, c(10,12,14,16)) <- NULL                   # Deleting
```

seqid *Generate Group-Id from Integer Sequences*

Description

seqid can be used to group sequences of integers in a vector, e.g. seqid(c(1:3,5:7)) becomes c(rep(1,3),rep(2,3)). It also supports increments > 1, unordered sequences, and missing values in the sequence.

Some applications are to facilitate identification of, and grouped operations on, (irregular) time-series and panels.

Usage

```
seqid(x, o = NULL, del = 1L, start = 1L, na.skip = FALSE,
      skip.seq = FALSE, check.o = TRUE)
```

Arguments

x	a factor or integer vector. Numeric vectors will be converted to integer i.e. rounded.
o	an (optional) integer ordering vector specifying the order by which to pass through x.
del	integer. The integer deliminating two consecutive points in a sequence. del = 1 means seqid tracks sequences of the form c(1,2,3,..), del = 2 tracks sequences c(1,3,5,..) etc.
start	integer. The starting value of the resulting sequence id. Default is starting from 1. For C++ programmers, starting from 0 could be a better choice.
na.skip	logical. Skip missing values in the sequence. The default behavior is skipping such that seqid(c(1,NA,2)) is regarded as one sequence and coded as c(1,NA,1).
skip.seq	logical. If na.skip = TRUE, this changes the behavior such that missing values are viewed as part of the sequence, i.e. seqid(c(1,NA,3)) is regarded as one sequence and coded as c(1,NA,1).
check.o	logical. Programmers option: FALSE prevents checking that each element of o is in the range [1,length(x)], it only checks the length of o. This gives some extra speed, but will terminate R if any element of o is too large or too small.

Details

seqid was created primarily to deal with problems of computing lagged values, differences and growth rates on irregularly spaced time-series and panels (#26). flag, fdiff and fgrowth do not natively support such panels because they do not pre-compute an ordering of the data but directly

compute the ordering from the supplied id and time variables while providing errors for gaps and repeated time values. see flag for computational details.

However fortunately any irregular time-series or panel-series can be expressed as a regular panel-series with a group-id created such that the time-periods within each group are consecutive.

A simple solution to applying existing functionality (flag, fdiff and fgrowth) to irregular time-series and panels is thus to create a group-id that fully identifies the data together with the time variable. seqid makes this very easy: For an irregular panel with some arbitrary gaps or repeated values in the time variable, an appropriate id variable can be generated using settransform(data,newid = seqid(time,radixorder(id,time))). Lags can then be computed using L(data,1,~newid,~time) etc. This way *collapse* maintains a balance between offering very fast computations on 99% of time series and panels (which may be unbalanced but where observations for each entity are consecutive in time), and flexibility of application.

In general, for any regularly spaced panel the identity given by identical(groupid(id,order(id,time)),seqid(time,or should hold.

I note that regularly spaced panels with gaps in time (such as a panel-survey) can be handled either by seqid(...,del = gap) or, in most cases, simply by converting the time variable to factor using qF, which will make observations consecutive.

There are potentially other more analytical applications for seqid...

For the opposite operation of creating a new time-variable that is consecutive in each group, see data.table::rowid.

Value

An integer vector of class 'qG'. See qG.

See Also

groupid, qG, Fast (Ordered) Grouping, Collapse Overview

Examples

```
## This creates an irregularly spaced panel, with a gap in time for id = 2
data <- data.frame(id = rep(1:3, each = 4),
                   time = c(1:4, 1:2, 4:5, 1:4),
                   value = rnorm(12))
data
## Not run:
## Gaps in time error
L(data, 1, value ~ id, ~time)

## End(Not run)
## Generating new id variable (here seqid(time) would suffice as data is sorted)
settransform(data, newid = seqid(time, order(id, time)))
data

## Lag the panel
L(data, 1, value ~ newid, ~time)

## A different solution: Simply creating a consecutive time variable
```

```
settransform(data, newtime = data.table::rowid(id))
data
L(data, 1, value ~ id, ~newtime)

## With sorted data we could of course also omit the time variable alltogether...
L(data, 1, value ~ id)
```

TRA *Transform Data by (Grouped) Replacing or Sweeping out Statistics*

Description

TRA is an S3 generic that efficiently transforms data by either (column-wise) replacing data values with supplied statistics or sweeping the statistics out of the data. TRA supports grouped sweeping and replacing operations, and is thus a generalization of sweep.

Usage

```
TRA(x, STATS, FUN = "-", ...)

## Default S3 method:
TRA(x, STATS, FUN = "-", g = NULL, ...)

## S3 method for class 'matrix'
TRA(x, STATS, FUN = "-", g = NULL, ...)

## S3 method for class 'data.frame'
TRA(x, STATS, FUN = "-", g = NULL, ...)

## S3 method for class 'grouped_df'
TRA(x, STATS, FUN = "-", keep.group_vars = TRUE, ...)
```

Arguments

x	a atomic vector, matrix, data frame or grouped tibble (dplyr::grouped_df).
STATS	a matching set of summary statistics computed on x. If g = NULL (no groups), all methods support an atomic vector of statistics of length NCOL(x). The matrix and data.frame methods also support a 1-row matrix or 1-row data.frame/list, respectively. If groups are supplied to g, STATS needs to be of the same type as x and of appropriate dimensions (such that NCOL(x) == NCOL(STATS) and NROW(STATS) matches the number of groups supplied to g i.e. the number of levels if g is a factor, with the first row of STATS corresponding to the first level of g etc...)
FUN	an integer or character string indicating the operation to perform. There are 10 supported operations:

Int.	*String*	*Description*
1	"replace_fill"	replace and overwrite missing values
2	"replace"	replace but preserve missing values
3	"-"	subtract (i.e. center)
4	"-+"	subtract group-statistics but add group-frequency weighted average of group statistics (i.e. center
5	"/"	divide (i.e. scale, but also changes mean. `fscale` can scale and keep mean)
6	"%"	compute percentages (i.e. divide and multiply by 100)
7	"+"	add
8	"*"	multiply
9	"%%"	modulus (i.e. remainder from division by STATS)
10	"-%%"	subtract modulus (i.e. floor data by STATS)

g — a factor, GRP object, atomic vector (internally converted to ordered factor) or a list of vectors / factors (internally converted to a GRP object) used to group x. Number of groups must match rows of STATS. See STATS and Details.

keep.group_vars — *grouped_df method:* Logical. Remove grouping variables after computation. In contrast to the other methods, TRA.grouped_df matches column names exactly, thus STATS can be any subset of aggregated columns in x in any order, with or without grouping columns. TRA.grouped_df will transform the columns in x with their aggregated versions matched from STATS (ignoring grouping columns found in x or STATS and columns in x not found in STATS), and return x again. If keep.group_vars = FALSE, x is returned again without grouping columns. See Details and Examples.

... — arguments to be passed to or from other methods.

Details

Without groups (g = NULL), TRA is nothing more than a column based version of base::sweep, albeit 4-times more efficient on matrices and many times more efficient on data frames. TRA always preserves all attributes of x.

With groups passed to g, TRA expects (and checks for) a set of statistics such that NROW(STATS) equals the number of groups. If this condition is satisfied, TRA will assume that the first row of STATS is the set of statistics computed on the first group of g, the second row on the second group etc. and do groupwise replacing or sweeping out accordingly.

For example Let x = c(1.2,4.6,2.5,9.1,8.7,3.3), g is an integer vector in 3 groups g = c(1,3,3,2,1,2) and STATS = fmean(x,g) = c(4.95,6.20,3.55). Then out = TRA(x,fmean(x,g),"-",g) = c(-3.75,1.05,-1.05,2.90,
(same as fmean(x,g,TRA = "-")) does the equivalent to the following for-loop: for(i in 1:6) out[i] = x[i] -fmean(x,g)[g[i]].

Correct computation requires that g as used in fmean and g passed to TRA are exactly the same vector. Using g = c(1,3,3,2,1,2) for fmean and g = c(3,1,1,2,3,2) for TRA will not give the right result. The safest way of programming with TRA is thus to repeatedly employ the same factor or GRP object for all grouped computations. Atomic vectors passed to g will be converted to ordered factors (see qF) and lists will be converted to ordered GRP objects. This is also done by all Fast Statistical Functions and by default by BY, thus together with these functions, TRA can also safely be used with atomic- or list-groups. Problems may arise if other functions internally convert atomic

vectors or lists to groups in a non-sorted way. *Note*: `as.factor` conversions are ok as this also involves sorting.

If `x` is a grouped tibble (`grouped_df`), `TRA` matches the columns of `x` and `STATS` and also checks for grouping columns in `x` and `STATS`. `TRA.grouped_df` will then only transform those columns in `x` for which matching counterparts were found in `STATS`, exempting grouping columns, and returns `x` again (with columns in the same order). If `keep.group_vars = FALSE`, the grouping columns are dropped after computation, however the "groups" attribute is not dropped (it can be removed using `dplyr::ungroup()`).

Value

`x` with columns replaced or swept out using `STATS`, grouped by `g`.

Note

I have tried to make `TRA` as redundant as possible by adding a TRA-argument to all Fast Statistical Functions (ensuring that the exact same grouping vector is used for aggregation and transformation), and by creating the `fbetween` / `B` (between-transformation) and `fwithin` / `W` (within-transform) as well as `fscale` / `STD` functions for frequent scaling, centering and averaging tasks.

See Also

sweep, Fast Statistical Functions, Data Transformations, Collapse Overview

Examples

```
v <- iris$Sepal.Length          # A numeric vector
f <- iris$Species               # A factor
dat <- num_vars(iris)           # Numeric columns
m <- qM(dat)                    # Matrix of numeric data

head(TRA(v, fmean(v)))            # Simple centering [same as fmean(v, TRA = "-") or W(v)]
head(TRA(m, fmean(m)))          # [same as sweep(m, 2, fmean(m)), fmean(m, TRA = "-") or W(m)]
head(TRA(dat, fmean(dat)))           # [same as fmean(dat, TRA = "-") or W(dat)]
head(TRA(v, fmean(v), "replace"))   # Simple replacing [same as fmean(v, TRA = "replace") or B(v)]
head(TRA(m, fmean(m), "replace"))   # [same as sweep(m, 2, fmean(m)), fmean(m, TRA = 1L) or B(m)]
head(TRA(dat, fmean(dat), "replace")) # [same as fmean(dat, TRA = "replace") or B(dat)]
head(TRA(m, fsd(m), "/"))           # Simple scaling... [same as fsd(m, TRA = "/")]...

# Note: All grouped examples also apply for v and dat...
head(TRA(m, fmean(m, f), "-", f))     # Centering [same as fmean(m, f, TRA = "-") or W(m, f)]
head(TRA(m, fmean(m, f), "replace", f)) # Replacing [same fmean(m, f, TRA = "replace") or B(m, f)]
head(TRA(m, fsd(m, f), "/", f))         # Scaling [same as fsd(m, f, TRA = "/")]

head(TRA(m, fmean(m, f), "-+", f))      # Centering on the overall mean ...
                                        # [same as fmean(m, f, TRA = "-+") or
                                        #           W(m, f, mean = "overall.mean")]
head(TRA(TRA(m, fmean(m, f), "-", f),   # Also the same thing done manually !!
     fmean(m), "+"))

# grouped tibble method
```

```
library(dplyr)
iris %>%  group_by(Species) %>% TRA(fmean(.))
iris %>%  group_by(Species) %>% fmean(TRA = "-")        # Same thing
iris %>%  group_by(Species) %>% TRA(fmean(.)[c(2,4)])   # Only transforming 2 columns
iris %>%  group_by(Species) %>% TRA(fmean(.)[c(2,4)],   # Dropping species column
                            keep.group_vars = FALSE)
```

`unlist2d` *Recursive Row-Binding / Unlisting in 2D - to Data Frame*

Description

`unlist2d` efficiently unlists lists of regular R objects (objects built up from atomic elements) and creates a data frame representation of the list. It is a full 2-dimensional generalization of `base::unlist`, but is best understood and used as a recursive generalization of `do.call(rbind,l)`, for lists of vectors, data frames, arrays or heterogeneous objects (i.e. unlisting happens via recursive flattening and intelligent row-binding of objects, see Details and Examples).

Usage

```
unlist2d(l, idcols = ".id", row.names = FALSE, recursive = TRUE,
         id.factor = FALSE, DT = FALSE)
```

Arguments

`l`	a unlistable list (with atomic elements in all final nodes, see `is.unlistable`).
`idcols`	a character stub or a vector of names for id-columns automatically added - one for each level of nesting in `l`. By default the stub is `".id"`, so columns will be of the form `".id.1"`,`".id.2"`, etc... . if `idcols = TRUE`, the stub is also set to `".id"`. If `idcols = FALSE`, id-columns are omitted. The content of the id columns are the list names, or (if missing) integers for the list elements. Missing elements in asymmetric nested structures are filled up with `NA`. See Examples.
`row.names`	`TRUE` extracts row names from all the objects in `l` (where available) and adds them to the output in a column named `"row.names"`. Alternatively, a column name i.e. `row.names = "file"` can be supplied.
`recursive`	if `FALSE`, only process the lowest (deepest) level of `l`.
`id.factor`	if `TRUE` and `idcols != FALSE`, create id columns as ordered factors instead of character or integer vectors. This is useful if id's are used for further analysis e.g. as inputs to `ggplot2`.
`DT`	if `TRUE`, return a *data.table*, not a data frame.

Details

The data frame representation created by `unlist2d` is built as follows:

- Recurse down to the lowest level of the list-tree, data frames are exempted and treated as a final elements.

- Check out the objects, if they are vectors, matrices or arrays convert them to data frame (in the case of atomic vectors each element becomes a column).
- Row-bind these data frame's using data.table's `rbindlist` function. Columns are matched by name. If the number of columns differ, fill empty spaces with NA's. If `idcols != FALSE`, create a id-columns on the left, filled with the object names or indices (if unnamed). If `row.names = TRUE`, store row-names of the objects (if available) in a separate column.
- Move up to the next higher level of the list-tree and repeat: Convert atomic objects to data frame and row-bind while matching all columns and filling unmatched ones with NA's. Create another id-column for each level of nesting passed through. If the list-tree is asymmetric, fill empty spaces in lower-level id columns with NA's.

The result of this iterative procedure is a single data frame containing on the left side id-columns for each level of nesting (from higher to lower level), followed by a column containing all the row.names of the objects if `row.names = TRUE`, followed by the object columns, matched at each level of recursion. Optimal results are of course obtained with symmetric lists of arrays, matrices or data frames, which `unlist2d` nicely converts to a beautiful data frame ready for plotting or further analysis. See examples below.

Value

A data frame or (if `DT = TRUE`) a *data.table*.

Note

For lists of data frames `unlist2d` works just like `data.table::rbindlist(l,use.names = TRUE,fill = TRUE,idcol = ".id")` (also the same speed), however for lists of lists `unlist2d` does not produce the same output as `data.table::rbindlist`.

See Also

`rapply2d`, List Processing, Collapse Overview

Examples

```
## basic examples:
l <- list(mtcars, list(mtcars, mtcars))
unlist2d(l)
unlist2d(rapply2d(l, fmean))
l = list(a = qM(mtcars[1:8]),
         b = list(c = mtcars[4:11], d = list(e = mtcars[2:10], f = mtcars)))
unlist2d(l, row.names = TRUE)
unlist2d(rapply2d(l, fmean))
unlist2d(rapply2d(l, fmean), recursive = FALSE)

## Groningen Growth and Development Center 10-Sector Database
head(GGDC10S) # See ?GGDC10S
namlab(GGDC10S, class = TRUE)

# Panel-Summarize this data by Variable (Emloyment and Value Added)
l <- qsu(GGDC10S, by = ~ Variable,             # Output as list (instead of 4D array)
         pid = ~ Variable + Country,
```

```
        cols = 6:16, array = FALSE)
str(l)                                  # A list of 2-levels with matrices of statistics
head(unlist2d(l))                      # Default output, missing the variables (row-names)
head(unlist2d(l, row.names = TRUE))          # Here we go, but this is still not very nice
head(unlist2d(l, idcols = c("Sector","Trans"), # Now this is looking pretty good
              row.names = "Variable"))

dat <- unlist2d(l, c("Sector","Trans"),     # Id-columns can also be generated as ordered factors
                "Variable", id.factor = TRUE)
str(dat)

# Split this sectoral data, first by Variable (Emloyment and Value Added), then by Country
sdat <- rapply2d(split(GGDC10S[c(1,6:16)], GGDC10S$Variable), function(x) split(x[-1],x[[1]]))

# Compute pairwise correlations between sectors and recombine:
dat <- unlist2d(rapply2d(sdat, pwcor),
                idcols = c("Variable","Country"),
                row.names = "Sector")
head(dat)
plot(hclust(as.dist(1-pwcor(dat[-(1:3)]))))   # Using corrs. as distance metric to cluster sectors

# Together with other functions like psmat, unlist2d can also effectively help reshape data:
head(unlist2d(psmat(subset(GGDC10S, Variable == "VA"), ~Country, ~Year, cols = 6:16, array = FALSE),
              idcols = "Sector", row.names = "Country"))
```

`varying` *Fast Check of Variation in Data*

Description

`varying` is a generic function that (column-wise) checks for variation in the values of `x`, (optionally) within the groups `g` (i.e. a panel-identifier).

Usage

```
varying(x, ...)

## Default S3 method:
varying(x, g = NULL, any_group = TRUE, use.g.names = TRUE, ...)

## S3 method for class 'matrix'
varying(x, g = NULL, any_group = TRUE, use.g.names = TRUE, drop = TRUE, ...)

## S3 method for class 'data.frame'
varying(x, by = NULL, cols = NULL, any_group = TRUE, use.g.names = TRUE, drop = TRUE, ...)

# Methods for compatibility with plm:
```

```
## S3 method for class 'pseries'
varying(x, effect = 1L, any_group = TRUE, use.g.names = TRUE, ...)

## S3 method for class 'pdata.frame'
varying(x, effect = 1L, cols = NULL, any_group = TRUE, use.g.names = TRUE,
        drop = TRUE, ...)

# Methods for compatibility with dplyr:

## S3 method for class 'grouped_df'
varying(x, any_group = TRUE, use.g.names = FALSE, drop = TRUE,
        keep.group_vars = TRUE, ...)
```

Arguments

x	a vector, matrix, data.frame or grouped tibble (dplyr::grouped_df).
g	a factor, GRP object, atomic vector (internally converted to factor) or a list of vectors / factors (internally converted to a GRP object) used to group x.
by	same as g, but also allows one- or two-sided formulas i.e. ~ group1 + group2 or var1 + var2 ~ group1 + group2. See Examples
any_group	logical. If !is.null(g), FALSE will check and report variation in all groups, whereas the default TRUE only checks if there is variation within any group. See Examples.
cols	select columns using column names, indices or a function (i.e. is.numeric). Two-sided formulas passed to by overwrite cols.
use.g.names	make group-names and add to the result as names (vector method) or row-names (matrix and data.frame method). No row-names are generated for data.tables and (default) grouped tibbles.
drop	*matrix and data.frame methods:* drop dimensions and return an atomic vector if the result is 1-dimensional.
effect	*plm* methods: Select which panel identifier should be used for between and within transformations of the data. 1L means first variable in the plm::index, 2L the second etc.. Index variables can also be called by name. More than one variable can be supplied.
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation.
...	arguments to be passed to or from other methods.

Details

Without groups passed to g, varying simply checks if there is any variation in the columns of x and returns TRUE for each column where this is the case and FALSE otherwise. A set of data points is defined as varying if it contains at least 2 distinct non-missing values (such that a non-0 standard deviation can be computed on numeric data). varying checks for variation in both numeric and non-numeric data.

If groups are supplied to g (or alternatively a *grouped_df* to x), varying can operate in one of 2 modes:

- If any_group = TRUE (the default), varying checks each column for variation in any of the groups defined by g, and returns TRUE if such within-variation was detected and FALSE otherwise. Thus only one logical value is returned for each column and the computation on each column is terminated as soon as any variation within any group was found.
- If any_group = FALSE, varying runs through the entire data checking each group for variation and returns, for each column in x, a logical vector reporting the variation check for all groups. If a group contains only missing values, a NA is returned for that group.

Value

A logical vector or (if !is.null(g) and any_group = FALSE), a matrix or data.frame of logical vectors indicating whether the data vary (over the dimension supplied by g).

See Also

Data Transformations, Collapse Overview

Examples

```
## Checks overall variation in all columns
varying(wlddev)

## Checks whether data are time-variant i.e. vary within country
varying(wlddev, wlddev$country)

## Same as above but done for each country individually, countries wothout data are coded NA
varying(wlddev, wlddev$country, any_group = FALSE)
```

---

wlddev *World Development Dataset*

Description

This dataset contains 4 indicators from the World Bank's World Development Indicators (WDI) database: (1) GDP per capita, (2) Life expectancy at birth, (3) GINI index and (4) Net ODA received. The panel-data is balanced and covers 216 present and historic countries from 1960-2018 (World Bank aggregates and regional entities are excluded).

Apart from the indicators the data contains a number of identifiers (character country name, factor ISO3 country code, World Bank region and income level, numeric year and decade) and 2 generated variables: A logical variable indicating whether the country is an OECD member, and a fictitious variable stating the date the data was recorded. These variables were added so that all common data-types are represented in this dataset, making it an ideal test-dataset for certain *collapse* functions.

Usage

```
data("wlddev")
```

Format

A data frame with 12744 observations on the following 12 variables. All variables are labelled e.g. have a 'label' attribute.

`country` *chr* Country Name

`iso3c` *fct* Country Code

`date` *date* Date Recorded (Fictitious)

`year` *num* Year

`decade` *num* Decade

`region` *fct* World Bank Region

`income` *fct* World Bank Income Level

`OECD` *log* Is OECD Member Country?

`PCGDP` *num* GDP per capita (constant 2010 US$)

`LIFEEX` *num* Life expectancy at birth, total (years)

`GINI` *num* GINI index (World Bank estimate)

`ODA` *num* Net ODA received (constant 2015 US$)

Source

https://data.worldbank.org/. Search `vlabels(wlddev)[9:12]` to find the right series.

See Also

GGDC10S, Collapse Overview

Examples

```
data(wlddev)

# Panel-summarizing the 4 series
qsu(wlddev, pid = ~iso3c, cols = 9:12, vlabels = TRUE)

# By Region
qsu(wlddev, by = ~region, cols = 9:12, vlabels = TRUE)

# Panel-summary by region
qsu(wlddev, by = ~region, pid = ~iso3c, cols = 9:12, vlabels = TRUE)

# Pairwise correlations: Ovarall
print(pwcor(get_vars(wlddev, 9:12), N = TRUE, P = TRUE), show = "lower.tri")

# Pairwise correlations: Between Countries
print(pwcor(fmean(get_vars(wlddev, 9:12), wlddev$iso3c), N = TRUE, P = TRUE), show = "lower.tri")

# Pairwise correlations: Within Countries
print(pwcor(fwithin(get_vars(wlddev, 9:12), wlddev$iso3c), N = TRUE, P = TRUE), show = "lower.tri")
```

Index