Package 'collapse'

January 23, 2022

Title Advanced and Fast Data Transformation

Version 1.7.2

Date 2022-01-21

Description A C/C++ based package for advanced data transformation and statistical computing in R that is extremely fast, flexible and parsimonious to code with, class-agnostic and programmer friendly. It is well integrated with base R, 'dplyr' / (grouped) 'tibble', 'data.table', 'plm' (panel-series and data frames), 'sf' data frames, and non-destructively handles other matrix or data frame based classes (such as 'ts', 'xts' / 'zoo', 'timeSeries', 'tsibble', 'tibbletime' etc.)
--- Key Features: ---
(1) Advanced statistical programming: A full set of fast statistical functions supporting grouped and weighted computations on vectors, matrices and data frames. Fast and programmable grouping, ordering, unique values / rows, factor generation and interactions. Fast and flexible functions for data manipulation, data object conversions, and memory efficient R programming.
(2) Advanced aggregation: Fast and easy multi-data-type, multi-function, weighted, parallelized and fully custom data aggregation.
(3) Advanced transformations: Fast row / column arithmetic, (grouped) replacing and sweeping out of statistics, (grouped, weighted) scaling / standardizing, between (averaging) and (quasi-)within (demeaning) transformations, higher-dimensional centering (i.e. multiple fixed effects or polynomials), linear prediction, model fitting and testing exclusion restrictions.
(4) Advanced time-computations: Fast (sequences of) lags / leads, and (lagged / leaded, iterated, quasi-, log-) differences and (compounded) growth rates on (irregular) 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, splitting, extraction / subsetting, data-apply, and generalized recursive row-binding / unlisting in 2D.
(6) Advanced data exploration: Fast (grouped, weighted, panel-decomposed) summary statistics for complex multilevel / panel data.

URL https://sebkrantz.github.io/collapse/, https://github.com/SebKrantz/collapse,

https://twitter.com/collapse_R

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

License GPL (>= 2) | file LICENSE

Encoding UTF-8

LazyData true

Depends R (>= 2.10)

Imports Rcpp (>= 1.0.1)

LinkingTo Rcpp

Suggests data.table, sf, matrixStats, magrittr, kit, plm, fixest, vars, weights, RcppArmadillo, RcppEigen, dplyr, ggplot2, scales, microbenchmark, testthat, covr, knitr, rmarkdown

SystemRequirements C++11

VignetteBuilder knitr

NeedsCompilation yes

Author Sebastian Krantz [aut, cre],
Matt Dowle [ctb],
Arun Srinivasan [ctb],
Morgan Jacob [ctb],
Dirk Eddelbuettel [ctb],
Laurent Berge [ctb],
Josh Pasek [ctb],
Kevin Tappe [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 2022-01-23 00:32:41 UTC

R topics documented:

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

Description

collapse is a C/C++ based package for data transformation and statistical computing in R. It's aims are:

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

It is made compatible with *dplyr*, *data.table*, *sf* and the *plm* approach to panel data, and non-destructively handles other classes such as *xts*.

Getting Started

See Collapse Documentation & Overview (the most up-to-date documentation for *collapse* 1.7), or read the introductory vignette. All vignettes can be accessed on the package website. A cheatsheet is available at here. A compact introduction for quick-starters is provided in the examples section below.

Details

collapse provides an integrated suite of statistical and data manipulation functions. These improve, complement and extend the capabilities of base R and packages like *dplyr*, *data.table*, *plm*, *matrixStats*, *Rfast* etc.. Key Highlights:

- Fast C/C++ based (grouped, weighted) computations embedded in highly optimized R code.
- More complex statistical, time series / panel data and recursive (list-processing) operations.
- A flexible and generic approach supporting and preserving many R objects.
- Optimized programming in standard and non-standard evaluation.

The statistical functions in *collapse* are S3 generic with core methods for vectors, matrices and data frames, and internally support grouped and weighted computations carried out in C/C++. R code is strongly optimized and inputs are swiftly passed to compiled C/C++ code, where various efficient algorithms are implemented.

To facilitate efficient programming, core S3 methods, grouping and ordering functionality and some C-level helper functions can be accessed by the user.

Additional (hidden) S3 methods and C-level features enable broad based compatibility with *dplyr* (grouped tibble), *data.table*, *sf* and *plm* panel data classes. Functions and core methods also seek to preserve object attributes (including column attributes such as variable labels), ensuring flexibility and effective workflows with a very broad range of R objects (including most time-series classes).

Missing values are efficiently skipped at C/C++ level. The package default is `na.rm = TRUE`, whereas `na.rm = FALSE` also yields efficient checking and early termination. Missing weights are supported. Core functionality and all statistical functions / computations are tested with 13,000 unit tests for Base R equivalence, exempting some improvements (e.g. `fsum(NA,na.rm = TRUE)` evaluates to `NA`, not 0, similarly for `fmin` and `fmax`; no `NaN` values are generated from computations involving `NA` values). Generic functions provide some security against silent swallowing of arguments.

collapse installs with a built-in hierarchical documentation facilitating the use of the package. The vignettes are complimentary and also follow a more structured approach.

The package is coded both in C and C++ and built with *Rcpp*, but also uses C/C++ functions from *data.table* (grouping, ordering, subsetting, row-binding), *kit* (hash-based grouping), *fixest* (centering on multiple factors), *weights* (weighted pairwise correlations), *stats* (ACF and PACF) and *RcppArmadillo / RcppEigen* (fast linear fitting methods). For the moment *collapse* does not utilize low-level parallelism (such as OpenMP).

Author(s)

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

Other contributors from packages *collapse* utilizes:

- Matt Dowle, Arun Srinivasan and contributors worldwide (*data.table*)
- Dirk Eddelbuettel and contributors worldwide (*Rcpp*, *RcppArmadillo*, *RcppEigen*)
- Morgan Jacob (*kit*)
- Laurent Berge (*fixest*)
- Josh Pasek (*weights*)
- R Core Team and contributors worldwide (*stats*)

I thank Ralf Stubner, Joseph Wood and Dirk Eddelbuettel and a host of other quant people from diverse fields for helpful answers on Stackoverflow, Joris Meys for encouraging me and helping to set up the Github repository for *collapse*, Matthieu Stigler, Patrice Kiener, Zhiyi Xu, Kevin Tappe and Grant McDermott for feature requests and helpful suggestions.

Developing / 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 to the 'development' branch of the repository, or e-mail me.
- Please report issues at https://github.com/SebKrantz/collapse/issues.

Examples

```
## Let's start with some statistical programming
v <- iris$Sepal.Length
d <- num_vars(iris)    # Saving numeric variables
f <- iris$Species      # Factor

# Simple statistics
fmean(v)               # vector
fmean(qM(d))           # matrix (qM is a faster as.matrix)
fmean(d)               # data.frame

# Preserving data structure
fmean(qM(d), drop = FALSE)     # Still a matrix
fmean(d, drop = FALSE)         # Still a data.frame

# Weighted statistics, supported by most functions...
w <- abs(rnorm(fnrow(iris)))
fmean(d, w = w)

# Grouped statistics...
fmean(d, f)

# Groupwise-weighted statistics...
fmean(d, f, w)

# Simple Transformations...
head(fmode(d, TRA = "replace"))    # Replacing values with the mode
head(fmedian(d, TRA = "-"))        # Subtracting the median
head(fsum(d, TRA = "%"))           # Computing percentages
head(fsd(d, TRA = "/"))           # Dividing by the standard-deviation (scaling), etc...

# Weighted Transformations...
head(fnth(d, 0.75, w = w, TRA = "replace"))  # Replacing by the weighted 3rd quartile

# Grouped Transformations...
head(fvar(d, f, TRA = "replace"))  # Replacing values with the group variance
head(fsd(d, f, TRA = "/"))         # Grouped scaling
head(fmin(d, f, TRA = "-"))        # Setting the minimum value in each species to 0
head(fsum(d, f, TRA = "/"))        # Dividing by the sum (proportions)
head(fmedian(d, f, TRA = "-"))     # Groupwise de-median
head(ffirst(d, f, TRA = "%%"))     # Taking modulus of first group-value, etc. ...

# Grouped and weighted transformations...
head(fsd(d, f, w, "/"), 3)         # weighted scaling
head(fmedian(d, f, w, "-"), 3)     # subtracting the weighted group-median
head(fmode(d, f, w, "replace"), 3) # replace with weighted statistical mode

## Some more advanced transformations...
head(fbetween(d))                      # Averaging (faster t.: fmean(d, TRA = "replace"))
head(fwithin(d))                          # Centering (faster than: fmean(d, TRA = "-"))
head(fwithin(d, f, w))                # Grouped and weighted (same as fmean(d, f, w, "-"))
head(fwithin(d, f, w, mean = 5))              # Setting a custom mean
```

```
head(fwithin(d, f, w, theta = 0.76))       # Quasi-centering i.e. d - theta*fbetween(d, f, w)
head(fwithin(d, f, w, mean = "overall.mean")) # Preserving the overall mean of the data
head(fscale(d))                            # Scaling and centering
head(fscale(d, mean = 5, sd = 3))          # Custom scaling and centering
head(fscale(d, mean = FALSE, sd = 3))      # Mean preserving scaling
head(fscale(d, f, w))                     # Grouped and weighted scaling and centering
head(fscale(d, f, w, mean = 5, sd = 3))    # Custom grouped and weighted scaling and centering
head(fscale(d, f, w, mean = FALSE,         # Preserving group means
        sd = "within.sd"))           # and setting group-sd to fsd(fwithin(d, f, w), w = w)
head(fscale(d, f, w, mean = "overall.mean",  # Full harmonization of group means and variances,
         sd = "within.sd"))           # while preserving the level and scale of the data.

head(get_vars(iris, 1:2))                # Use get_vars for fast selecting, gv is shortcut
head(fhdbetween(gv(iris, 1:2), gv(iris, 3:5))) # Linear prediction with factors and covariates
head(fhdwithin(gv(iris, 1:2), gv(iris, 3:5)))  # Linear partialling out factors and covariates
ss(iris, 1:10, 1:2)                       # Similarly fsubset/ss for fast subsetting rows

# Simple Time-Computations..
head(flag(AirPassengers, -1:3))               # One lead and three lags
head(fdiff(EuStockMarkets,                  # Suitably lagged first and second differences
      c(1, frequency(EuStockMarkets)), diff = 1:2))
head(fdiff(EuStockMarkets, rho = 0.87))       # Quasi-differences (x_t - rho*x_t-1)
head(fdiff(EuStockMarkets, log = TRUE))       # Log-differences
head(fgrowth(EuStockMarkets))                 # Exact growth rates (percentage change)
head(fgrowth(EuStockMarkets, logdiff = TRUE))  # Log-difference growth rates (percentage change)
# Note that it is not necessary to use factors for grouping.
fmean(gv(mtcars, -c(2,8:9)), mtcars$cyl) # Can also use vector (internally converted using qF())
fmean(gv(mtcars, -c(2,8:9)),
     gv(mtcars, c(2,8:9)))          # or a list of vector (internally grouped using GRP())
g <- GRP(mtcars, ~ cyl + vs + am)       # It is also possible to create grouping objects
print(g)                               # These are instructive to learn about the grouping,
plot(g)                                    # and are directly handed down to C++ code
fmean(gv(mtcars, -c(2,8:9)), g)       # This can speed up multiple computations over same groups
fsd(gv(mtcars, -c(2,8:9)), g)

# Factors can efficiently be created using qF()
f1 <- qF(mtcars$cyl)                    # Unlike GRP objects, factors are checked for NA's
f2 <- qF(mtcars$cyl, na.exclude = FALSE) # This can however be avoided through this option
class(f2)                                  # Note the added class

library(microbenchmark)
microbenchmark(fmean(mtcars, f1), fmean(mtcars, f2)) # A minor difference, larger on larger data

with(mtcars, finteraction(cyl, vs, am))  # Efficient interactions of vectors and/or factors
finteraction(gv(mtcars, c(2,8:9)))       # .. or lists of vectors/factors

# Simple row- or column-wise computations on matrices or data frames with dapply()
dapply(mtcars, quantile)                 # column quantiles
dapply(mtcars, quantile, MARGIN = 1)     # Row-quantiles
  # dapply preserves the data structure of any matrices / data frames passed
  # Some fast matrix row/column functions are also provided by the matrixStats package
# Similarly, BY performs grouped comptations
BY(mtcars, f2, quantile)
```

```
BY(mtcars, f2, quantile, expand.wide = TRUE)
# For efficient (grouped) replacing and sweeping out computed statistics, use TRA()
sds <- fsd(mtcars)
head(TRA(mtcars, sds, "/"))    # Simple scaling (if sd's not needed, use fsd(mtcars, TRA = "/"))

microbenchmark(TRA(mtcars, sds, "/"), sweep(mtcars, 2, sds, "/")) # A remarkable performance gain..

sds <- fsd(mtcars, f2)
head(TRA(mtcars, sds, "/", f2)) # Groupd scaling (if sd's not needed: fsd(mtcars, f2, TRA = "/"))

# All functions above perserve the structure of matrices / data frames
# If conversions are required, use these efficient functions:
mtcarsM <- qM(mtcars)                      # Matrix from data.frame
head(qDF(mtcarsM))                         # data.frame from matrix columns
head(mrtl(mtcarsM, TRUE, "data.frame"))    # data.frame from matrix rows, etc..
head(qDT(mtcarsM, "cars"))           # Saving row.names when converting matrix to data.table
head(qDT(mtcars, "cars"))                  # Same use a data.frame

## Now let's get some real data and see how we can use this power for data manipulation
library(magrittr)
head(wlddev) # World Bank World Development Data: 216 countries, 61 years, 5 series (columns 9-13)

# Starting with some discriptive tools...
namlab(wlddev, class = TRUE)           # Show variable names, labels and classes
fnobs(wlddev)                          # Observation count
pwnobs(wlddev)                         # Pairwise observation count
head(fnobs(wlddev, wlddev$country))    # Grouped observation count
fndistinct(wlddev)                     # Distinct values
descr(wlddev)                          # Describe data
varying(wlddev, ~ country)             # Show which variables vary within countries
qsu(wlddev, pid = ~ country,          # Panel-summarize columns 9 though 12 of this data
    cols = 9:12, vlabels = TRUE)       # (between and within countries)
qsu(wlddev, ~ region, ~ country,     # Do all of that by region and also compute higher moments
    cols = 9:12, higher = TRUE)        # -> returns a 4D array
qsu(wlddev, ~ region, ~ country, cols = 9:12,
   higher = TRUE, array = FALSE) %>%                     # Return as a list of matrices..
unlist2d(c("Variable","Trans"), row.names = "Region") %>% head  # and turn into a tidy data.frame
pwcor(num_vars(wlddev), P = TRUE)                     # Pairwise correlations with p-value
pwcor(fmean(num_vars(wlddev), wlddev$country), P = TRUE)    # Correlating country means
pwcor(fwithin(num_vars(wlddev), wlddev$country), P = TRUE)  # Within-country correlations
psacf(wlddev, ~country, ~year, cols = 9:12)           # Panel-data Autocorrelation function
pspacf(wlddev, ~country, ~year, cols = 9:12)             # Partial panel-autocorrelations
psmat(wlddev, ~iso3c, ~year, cols = 9:12) %>% plot       # Convert panel to 3D array and plot

## collapse offers a few very efficent functions for data manipulation:
# Fast selecting and replacing columns
series <- get_vars(wlddev, 9:12)     # Same as wlddev[9:12] but 2x faster
series <- fselect(wlddev, PCGDP:ODA) # Same thing: > 100x faster than dplyr::select
get_vars(wlddev, 9:12) <- series    # Replace, 8x faster wlddev[9:12] <- series + replaces names
fselect(wlddev, PCGDP:ODA) <- series # Same thing

# 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))
settransform(wlddev, ODA_GDP = ODA / PCGDP, ODA_LIFEEX = sqrt(ODA) / LIFEEX) # by reference
head(ftransform(wlddev, PCGDP = NULL, ODA = NULL, GINI_sum = fsum(GINI)))
head(ftransformv(wlddev, 9:12, log))          # Can also transform with lists of columns
head(ftransformv(wlddev, 9:12, fscale, apply = FALSE)) # apply = FALSE invokes fscale.data.frame
settransformv(wlddev, 9:12, fscale, apply = FALSE)     # Changing the data by reference
ftransform(wlddev) <- fscale(gv(wlddev, 9:12))      # Same thing (using replacement method)

wlddev %<>% ftransformv(9:12, fscale, apply = FALSE) # Same thing, using magrittr
wlddev %>% ftransform(gv(., 9:12) %>%           # With compound pipes: Scaling and lagging
                        fscale %>% flag(0:2, iso3c, year)) %>% head

# Fast reordering
head(roworder(wlddev, -country, year))
head(colorder(wlddev, country, year))

# Fast renaming
head(frename(wlddev, country = Ctry, year = Yr))
setrename(wlddev, country = Ctry, year = Yr)     # By reference
head(frename(wlddev, tolower, cols = 9:12))

# Fast grouping
fgroup_by(wlddev, Ctry, decade) %>% fgroup_vars %>% head # fgroup_by is faster than dplyr::group_by

rm(wlddev)                                    # .. but only works with collapse functions

## Now lets start putting things together

wlddev %>% fsubset(year > 1990, region, income, PCGDP:ODA) %>%
  fgroup_by(region, income) %>% fmean            # Fast aggregation using the mean

# Same thing using dplyr manipulation verbs
library(dplyr)
wlddev %>% filter(year > 1990) %>% select(region, income, PCGDP:ODA) %>%
 group_by(region,income) %>% fmean     # This is already a lot faster than summarize_all(mean)

wlddev %>% fsubset(year > 1990, region, income, PCGDP:POP) %>%
  fgroup_by(region, income) %>% fmean(POP)     # Weighted group means

wlddev %>% fsubset(year > 1990, region, income, PCGDP:POP) %>%
  fgroup_by(region, income) %>% fsd(POP)       # Weighted group standard deviations

wlddev %>% na_omit(cols = "POP") %>% fgroup_by(region, income) %>%
  fselect(PCGDP:POP) %>% fnth(0.75, POP)       # Weighted group third quartile

wlddev %>% fgroup_by(country) %>% fselect(PCGDP:ODA) %>%
fwithin %>% head                               # Within transformation
wlddev %>% fgroup_by(country) %>% fselect(PCGDP:ODA) %>%
fmedian(TRA = "-") %>% head                    # Grouped centering using the median
```

```
# Replacing data points by the weighted first quartile:
wlddev %>% na_omit(cols = "POP") %>% fgroup_by(country) %>%
  fselect(country, year, PCGDP:POP) %>%
  ftransform(fselect(., -country, -year) %>%
  fnth(0.25, POP, "replace_fill")) %>% head

wlddev %>% fgroup_by(country) %>% fselect(PCGDP:ODA) %>% fscale %>% head   # Standardizing
wlddev %>% fgroup_by(country) %>% fselect(PCGDP:POP) %>%
   fscale(POP) %>% head  # Weigted..

wlddev %>% fselect(country, year, PCGDP:ODA) %>%  # Adding 1 lead and 2 lags of each variable
  fgroup_by(country) %>% flag(-1:2, year) %>% head
wlddev %>% fselect(country, year, PCGDP:ODA) %>%  # Adding 1 lead and 10-year growth rates
  fgroup_by(country) %>% fgrowth(c(0:1,10), 1, year) %>% head

# etc...

# Aggregation with multiple functions
wlddev %>% fsubset(year > 1990, region, income, PCGDP:ODA) %>%
  fgroup_by(region, income) %>% {
    add_vars(fgroup_vars(., "unique"),
             fmedian(., keep.group_vars = FALSE) %>% add_stub("median_"),
             fmean(., keep.group_vars = FALSE) %>% add_stub("mean_"),
             fsd(., keep.group_vars = FALSE) %>% add_stub("sd_"))
  } %>% head

# Transformation with multiple functions
wlddev %>% fselect(country, year, PCGDP:ODA) %>%
  fgroup_by(country) %>% {
   add_vars(fdiff(., c(1,10), 1, year) %>% flag(0:2, year),  # Sequence of lagged differences
             ftransform(., fselect(., PCGDP:ODA) %>% fwithin %>% add_stub("W.")) %>%
            flag(0:2, year, keep.ids = FALSE))          # Sequence of lagged demeaned vars
  } %>% head

# With ftransform, can also easily do one or more grouped mutations on the fly..
settransform(wlddev, median_ODA = fmedian(ODA, list(region, income), TRA = "replace_fill"))

settransform(wlddev, sd_ODA = fsd(ODA, list(region, income), TRA = "replace_fill"),
                     mean_GDP = fmean(PCGDP, country, TRA = "replace_fill"))

wlddev %<>% ftransform(fmedian(list(median_ODA = ODA, median_GDP = PCGDP),
                               list(region, income), TRA = "replace_fill"))

# On a groped data frame it is also possible to grouped transform certain columns
# but perform aggregate operatins on others:
wlddev %>% fgroup_by(region, income) %>%
    ftransform(gmedian_GDP = fmedian(PCGDP, GRP(.), TRA = "replace"),
               omedian_GDP = fmedian(PCGDP, TRA = "replace"),  # "replace" preserves NA's
               omedian_GDP_fill = fmedian(PCGDP)) %>% tail

rm(wlddev)

## For multi-type data aggregation, the function collap offers ease and flexibility
```

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

# taking weighted mean and weighted mode:
head(collap(wlddev, ~ country + decade, fmean, fmode, w = ~ POP, wFUN = fsum))

# 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)))

# For grouped data frames use collapg
wlddev %>% fsubset(year > 1990, country, region, income, PCGDP:ODA) %>%
  fgroup_by(country) %>% collapg(fmean, ffirst) %>%
  ftransform(AMGDP = PCGDP > fmedian(PCGDP, list(region, income), TRA = "replace_fill"),
             AMODA = ODA > fmedian(ODA, income, TRA = "replace_fill")) %>% head

## Additional flexibility for data transformation tasks is offerend by tidy transformation operators
# Within-transformation (centering on overall mean)
head(W(wlddev, ~ country, cols = 9:12, mean = "overall.mean"))

# Partialling out country and year fixed effects
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
head(L(wlddev, -1:3, ~ country, ~year, cols = 9:12))
# Computing the 1- and 10-year first differences
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
# Computing the 1- and 10-year growth rates
head(G(wlddev, c(1,10), 1, ~ country, ~year, cols = 9:12))
# Adding growth rate variables to dataset
add_vars(wlddev) <- G(wlddev, c(1, 10), 1, ~ country, ~year, cols = 9:12, keep.ids = FALSE)
get_vars(wlddev, "G1.", regex = TRUE) <- NULL # Deleting again

# These operators can conveniently be used in regression formulas:
# Using a Mundlak (1978) procedure to estimate the effect of OECD on LIFEEX, controlling for PCGDP
lm(LIFEEX ~ log(PCGDP) + OECD + B(log(PCGDP), country),
   wlddev %>% fselect(country, OECD, PCGDP, LIFEEX) %>% na_omit)

# Adding 10-year lagged life-expectancy to allow for some convergence effects (dynamic panel model)
lm(LIFEEX ~ L(LIFEEX, 10, country) + log(PCGDP) + OECD + B(log(PCGDP), country),
   wlddev %>% fselect(country, OECD, PCGDP, LIFEEX) %>% na_omit)

# Tranformation functions and operators also support plm panel data classes:
```

```
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))
rm(pwlddev)

# Remove all objects used in this example section
rm(v, d, w, f, f1, f2, g, mtcarsM, sds, series, wlddev)
```

`across` *Apply Functions Across Multiple Columns*

Description

`across()` can be used inside `fmutate` and `fsummarise` to apply one or more functions to a selection of columns. It is overall very similar to `dplyr::across`, but does not support some `rlang` features, has some additional features (arguments), and is optimized to work with *collapse*'s, `.FAST_FUN`, yielding much faster computations.

Usage

```
across(.cols = NULL, .fns, ..., .names = NULL,
       .apply = "auto", .transpose = "auto")

# acr(...) can be used to abbreviate across(...)
```

Arguments

`.cols`	select columns using column names and expressions (e.g. `a:b` or `c(a,b,c:f)`), column indices, logical vectors, or functions yielding a logical value e.g. `is.numeric`. `NULL` applies functions to all columns except for grouping columns.
`.fns`	A function, character vector of functions or list of functions. Vectors / lists can be named to yield alternative names in the result (see `.names`). This argument is evaluated inside `substitute()`, and the content (not the names of vectors/lists) is checked against `.FAST_FUN` and `.OPERATOR_FUN`. Matching functions receive vectorized execution, other functions are applied to the data in a standard way.
`...`	further arguments to `.fns`. Arguments are evaluated in the data environment and split by groups as well (for non-vectorized functions, if of the same length as the data).

`.names` controls the naming of computed columns. `NULL` generates names of the form `coli_funj` if multiple functions are used. `.names = TRUE` enables this for a single function, `.names = FALSE` disables it for multiple functions (sensible for functions such as `.OPERATOR_FUN` that rename columns (if `.apply = FALSE`)). It is also possible to supply a function with two arguments for column and function names e.g. `function(c,f) paste0(f,"_",c)`. Finally, you can supply a custom vector of names which must match `length(.cols) * length(.fns)`.

`.apply` controls whether functions are applied column-by-column (`TRUE`) or to multiple columns at once (`FALSE`). The default, `"auto"`, does the latter for vectorized functions, which have an efficient data frame method. It can also be sensible to use `.apply = FALSE` for non-vectorized functions, especially multivariate functions like `lm` or `pwcor`, or functions renaming the data. See Examples.

`.transpose` with multiple `.fns`, `.transpose` controls whether the result is ordered first by column, then by function (`TRUE`), or vice-versa (`FALSE`). `"auto"` does the former if all functions yield results of the same dimensions (dimensions may differ if `.apply = FALSE`). See Examples.

Note

`across` does not support *purr*-style lambdas, and does not support `dplyr`-style predicate functions e.g. `across(where(is.numeric),sum)`, simply use `across(is.numeric,sum)`. In contrast to `dplyr`, you can also compute on grouping columns.

In general, my mission with `collapse` is not to create a `dplyr`-clone, but to take some of the useful features and make them robust and fast using base R and C/C++, with the aim of having a stable API. So don't ask me to implement the latest *dplyr* feature, unless you firmly believe it is very useful and will be around 10 years from now.

See Also

`fsummarise`, `fmutate`, Fast Data Manipulation, Collapse Overview

Examples

```
# Basic (Weighted) Summaries
fsummarise(wlddev, across(PCGDP:GINI, fmean, w = POP))

library(magrittr) # Note: Used because |> is not available on older R versions
wlddev %>% fgroup_by(region, income) %>%
    fsummarise(across(PCGDP:GINI, fmean, w = POP))

# Note that for these we don't actually need across...
fselect(wlddev, PCGDP:GINI) %>% fmean(w = wlddev$POP, drop = FALSE)
wlddev %>% fgroup_by(region, income) %>%
    fselect(PCGDP:GINI, POP) %>% fmean(POP, keep.w = FALSE)
collap(wlddev, PCGDP + LIFEEX + GINI ~ region + income, w = ~ POP, keep.w = FALSE)

# But if we want to use some base R function that reguires argument splitting...
wlddev %>% na_omit(cols = "POP") %>% fgroup_by(region, income) %>%
    fsummarise(across(PCGDP:GINI, weighted.mean, w = POP, na.rm = TRUE))
```

```
# Or if we want to apply different functions...
wlddev %>% fgroup_by(region, income) %>%
    fsummarise(across(PCGDP:GINI, list(mu = fmean, sd = fsd), w = POP),
               POP_sum = fsum(POP), OECD = fmean(OECD))
# Note that the above still detects fmean as a fast function, the names of the list
# are irrelevant, but the function name must be typed or passed as a character vector,
# Otherwise functions will be executed by groups e.g. function(x) fmean(x) won't vectorize

# Or we want to do more advanced things..
# Such as nesting data frames..
qTBL(wlddev) %>% fgroup_by(region, income) %>%
    fsummarise(across(c(PCGDP, LIFEEX, ODA),
               function(x) list(Nest = list(x)),
               .apply = FALSE))
# Or linear models..
qTBL(wlddev) %>% fgroup_by(region, income) %>%
    fsummarise(across(c(PCGDP, LIFEEX, ODA),
               function(x) list(Mods = list(lm(PCGDP ~., x))),
               .apply = FALSE))
# Or cumputing grouped correlation matrices
qTBL(wlddev) %>% fgroup_by(region, income) %>%
    fsummarise(across(c(PCGDP, LIFEEX, ODA),
      function(x) qDF(pwcor(x), "Variable"), .apply = FALSE))

# Here calculating 1- and 10-year lags and growth rates of these variables
qTBL(wlddev) %>% fgroup_by(country) %>%
    fmutate(across(c(PCGDP, LIFEEX, ODA), list(L, G),
                   n = c(1, 10), t = year, .names = FALSE))

# Same but variables in different order
qTBL(wlddev) %>% fgroup_by(country) %>%
    fmutate(across(c(PCGDP, LIFEEX, ODA), list(L, G), n = c(1, 10),
                   t = year, .names = FALSE, .transpose = FALSE))
```

`arithmetic` *Fast Row/Column Arithmetic for Matrix-Like Objects*

Description

Fast operators to perform row- or column-wise replacing and sweeping operations of vectors on matrices, data frames, lists.

Usage

```
## Perform the operation with v and each row of X

X %rr% v    # Replace rows of X with v
X %r+% v    # Add v to each row of X
```

```
X %r-% v    # Subtract v from each row of X
X %r*% v    # Multiply each row of X with v
X %r/% v    # Divide each row of X by v

## Perform a column-wise operation between V and X

X %cr% V    # Replace columns of X with V
X %c+% V    # Add V to columns of X
X %c-% V    # Subtract V from columns of X
X %c*% V    # Multiply columns of X with V
X %c/% V    # Divide columns of X by V
```

Arguments

X	a vector, matrix, data frame or list like object (with rows (r) columns (c) matching v / V).
v	for row operations: an atomic vector of matching NCOL(X). If X is a data frame, v can also be a list of scalar atomic elements. It is also possible to sweep lists of vectors v out of lists of matrices or data frames X.
V	for column operations: a suitable scalar, vector, or matrix / data frame matching NROW(X). X can also be a list of vectors / matrices in which case V can be a scalar / vector / matrix or matching list of scalars / vectors / matrices.

Details

With a matrix or data frame X, the default behavior of R when calling X op v (such as multiplication X * v) is to perform the operation of v with each column of X. The equivalent operation is performed by X %cop% V, with the difference that it computes significantly faster if X/V is a data frame / list. A more complex but frequently required task is to perform an operation with v on each row of X. This is provided based on efficient C++ code by the %rop% set of functions, e.g. X %r*% v efficiently multiplies v to each row of X.

Value

X where the operation with v / V was performed on each row or column. All attributes of X are preserved.

Note

Computations and Output: These functions are all quite simple, they only work with X on the LHS i.e. v %op% X will likely fail. The row operations are simple wrappers around TRA which provides more operations including grouped replacing and sweeping (where v would be a matrix or data frame with less rows than X being mapped to the rows of X by grouping vectors). One consequence is that just like TRA, row-wise mathematical operations (+, -, *, /) always yield numeric output, even if both X and v may be integer. This is different for column- operations which depend on base R and may also preserve integer data.

Rules of Arithmetic: Since these operators are defined as simple infix functions, the normal rules of arithmetic are not respected. So a %c+% b %c*% c evaluates as (a %c+% b) %c*% c. As with all chained infix operations, they are just evaluated sequentially from left to right.

Performance Notes: The function setop and a related set of %op=% operators can be used to perform these operations by reference, and are faster if copies of the output are not required!! Furthermore, for Fast Statistical Functions, using fmedian(X,TRA = "-") will be a tiny bit faster than X %r-% fmedian(X). Also use fwithin(X) for fast centering using the mean, and fscale(X) for fast scaling and centering or mean-preserving scaling.

See Also

setop, TRA, dapply, Efficient Programming, Data Transformations, Collapse Overview

Examples

```
## Using data frame's / lists
v <- mtcars$cyl
mtcars %cr% v
mtcars %c-% v
mtcars %r-% seq_col(mtcars)
mtcars %r-% lapply(mtcars, quantile, 0.28)

mtcars %c*% 5       # Significantly faster than mtcars * 5
mtcars %c*% mtcars  # Significantly faster than mtcars * mtcars

## Using matrices
X <- qM(mtcars)
X %cr% v
X %c-% v
X %r-% dapply(X, quantile, 0.28)

## Chained Operations
library(magrittr) # Note: Used because |> is not available on older R versions
mtcars %>% fwithin() %r-% rnorm(11) %c*% 5 %>%
    tfm(mpg = fsum(mpg)) %>% qsu()
```

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. Similar to dapply it seeks to retain the structure and attributes of the data, but can also output to various standard 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", "vector", "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, ..., keep.group_vars = TRUE, use.g.names = FALSE)
```

Arguments

x	a atomic vector, matrix, data frame or alike object.
g	a GRP object, or a factor / atomic vector / list of atomic vectors (internally converted to a GRP object) used to group x.
FUN	a function, can be scalar- or vector-valued. For vector valued functions see expand.wide and the Note.
...	further arguments to FUN, or to BY.data.frame for the 'grouped_df' method.
use.g.names	logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.
sort	logical. Sort the groups? Internally passed to GRP, 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 mclapply instead of lapply.
mc.cores	integer. Argument to mclapply indicating the number of cores to use for parallel execution. Can use detectCores() to select all available cores.
return	an integer or string indicating the type of object to return. The default 1 -"same" returns the same object type (i.e. class and other attributes are retained if the underlying data type is the same, just the names for the dimensions are adjusted). 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. See also the Note.

Details

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

It is principally a wrapper around lapply(gsplit(x,g),FUN,...), that uses gsplit for optimized splitting and also strongly optimizes on the internal code compared to *base* R functions. For more details look at the documentation for dapply which works very similar (apart from the splitting performed in BY). The function is intended for simple usage involving data aggregation or flexible computation of summary statistics across groups. For larger tasks requiring split-apply-combine computing on data frames, the Fast Statistical Functions and the *data.table* package are more appropriate tools.

Value

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

Note

BY can be used with vector-valued functions preserving the length of the data, note however that data are recombined in the order of the groups, not in the order of the original data. It is thus advisable to sort the data by the grouping variable before using BY with such a function.

See Also

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

Examples

```
v <- iris$Sepal.Length   # A numeric vector
f <- GRP(iris$Species)   # A grouping

## default vector method
BY(v, f, sum)                                # Sum by species
head(BY(v, f, scale))                      # Scale by species (please use fscale instead)
head(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
head(BY(m, f, scale))
head(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
head(BY(num_vars(iris), f, scale))
head(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 data frame method
library(magrittr) # Note: Used because |> is not available on older R versions
giris <- fgroup_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(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), multi-function aggregations applying multiple functions to each column, 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 = TRUE, decreasing = FALSE, na.last = TRUE, parallel = FALSE, mc.cores = 2L,
     return = c("wide","list","long","long_dupl"), give.names = "auto", sort.row, ...)

# 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 = TRUE, decreasing = FALSE, na.last = TRUE, parallel = FALSE, mc.cores = 2L,
      return = c("wide","list","long","long_dupl"), give.names = "auto", sort.row, ...)
```

```
# Auxiliary function: for grouped data ('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,
        parallel = FALSE, mc.cores = 2L,
     return = c("wide","list","long","long_dupl"), give.names = "auto", sort.row, ...)
```

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 indicated 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. *Notes*: custom lets collap ignore any inputs passed to FUN, catFUN or cols. Since v1.6.0 you can also rename columns e.g. custom = list(fmean = c(newname = "col1","col2"),fmode = c(newname = 3)).
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, decreasing, na.last	logical. Arguments passed to GRP.default and affecting the row-order in the aggregated data frame.

parallel	logical. Use mclapply instead of lapply to parallelize the computation at the column level. Not available for Windows.
mc.cores	integer. Argument to mclapply setting the number of cores to use, default is 2.
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 frames - 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.var'. 'auto' will automatically create such prefixes whenever multiple functions are applied to a column.
sort.row	depreciated, renamed to sort.
...	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 statistical 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 setting parallel = TRUE on a non-windows computer, aggregations will efficiently be parallelized at the column level using mclapply utilizing mc.cores cores.

Value

X aggregated. If X is not a data frame it is coerced to one using qDF and then aggregated.

Note

(1) Additional arguments passed are not split by groups. Weighted aggregations with user defined functions should be done with fsummarise, or using the *data.table* package.

(2) When the w argument is used, the weights are passed to all Fast Statistical Functions. This may be undesirable in settings like collapse::collap(data,~ id,custom = list(fsum = ...,fmean = ...),w = ~ weights) where some columns are to be aggregated using the weighted mean, and others using a simple sum or another unweighted statistic. Since many Fast Statistical Functions including fsum support weights, the above computes a weighted mean and a weighted sum. A couple of workarounds were outlined here, but *collapse* 1.5.0 incorporates an easy solution into collap: It is now possible to simply append Fast Statistical Functions by _uw to yield an unweighted computation. So for the above example we can write: collapse::collap(data,~ id,custom = list(fsum_uw = ...,fmean = ...),w = ~ weights) to get the weighted mean and

the simple sum. *Note* that the _uw functions are not available for use outside collap. Thus one also needs to quote them when passed to the FUN or catFUN arguments, e.g. use collap(data,~ id,fmean,"fmode_uw",w = ~ weighs), since collap(data,~ id,fmean,fmode_uw,w = ~ weighs) gives an error stating that fmode_uw was not found. *Note* also that it is never necessary for functions passed to wFUN to be appended like this, as the weights are never used to aggregate themselves.

(3) The dispatch between using optimized Fast Statistical Functions performing grouped computations internally or calling BY to perform split-apply-combine computing is done by matching the function name against .FAST_STAT_FUN. Thus code like collapse::collap(data,~ id,collapse::fmedian) does not yield an optimized computation, as "collapse::fmedian" %!in% .FAST_STAT_FUN. It is sufficient to write collapse::collap(data,~ id,"fmedian") to get the desired result when the *collapse* namespace is not attached.

See Also

fsummarise, 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)



## 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 = 9:13))            # Only the 5 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 ---------------------------------------
# We aggregate to region level using population weights
head(collap(wlddev, ~ region + year, w = ~ POP))           # Takes weighted mean for numeric..
# ..and weighted mode for categorical data. The weight vector is aggregated using fsum

head(collap(wlddev, ~ region + year, w = ~ POP,             # Aggregating weights using sum
          wFUN = list(fsum, fmax)))                        # and max (corresponding to mode)


## Multi-Function Aggregation ---------------------------------
head(collap(wlddev, ~ country + decade, list(fmean, fnobs),     # Saving mean and Nobs
            cols = 9:13))

head(collap(wlddev, ~ country + decade,                       # Same using base R -> slower
            list(mean = mean,
                 Nobs = function(x, ...) sum(!is.na(x))),
            cols = 9:13, na.rm = TRUE))

lapply(collap(wlddev, ~ country + decade,                         # List output format
            list(fmean, fnobs), cols = 9:13, return = "list"), head)

head(collap(wlddev, ~ country + decade,                           # Long output format
            list(fmean, fnobs), cols = 9:13, 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 = 9:13, 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 = ~ POP,
            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 = ~ POP,
            wFUN = list(fsum, fmax),
            parallel = TRUE, mc.cores = 1L, keep.col.order = FALSE))


## Piped Use ---------------------------------------------------
library(magrittr) # Note: Used because |> is not available on older R versions
iris %>% fgroup_by(Species) %>% collapg()
wlddev %>% fgroup_by(country, decade) %>% collapg() %>% head()
wlddev %>% fgroup_by(region, year) %>% collapg(w = POP) %>% head()
wlddev %>% fgroup_by(country, decade) %>% collapg(fmedian, flast) %>% head()
wlddev %>% fgroup_by(country, decade) %>%
  collapg(custom = list(fmean = 9:12, fmode = 5:7, flast = 3)) %>% head()
```

---

collapse-depreciated *Depreciated* collapse *Functions*

Description

The functions Recode and replace_non_finite available until *collapse* v1.1.0 will be removed soon. Since v1.2.0, Recode is replaced by recode_num and recode_char and replace_non_finite is replaced by replace_Inf. Since version 1.5.1, is.regular is depreciated - the function is not very useful and clashes with a more important one in the *zoo* package.

Usage

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

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

is.regular(x)
```

Arguments

X	a vector, matrix or data frame.
x	an R object.
...	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 and Replace Values, Collapse Overview

Examples

```
## Not run:
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)

## End(Not run)
```

`collapse-documentation`

Collapse Documentation & Overview

Description

The following table fully summarizes the contents of *collapse*. The documentation is structured hierarchically: 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 gro
Fast Grouping and Ordering	Fast (ordered) groupings from vectors, data frames, lists. 'GRP' objects are extremely
Fast Data Manipulation	Fast and flexible select, subset, summarise, mutate/transform, sort/reorder, rename and
Quick Data Conversion	Quick conversions: data.frame <> data.table <> tibble | matrix <> list, data.frame, data
Advanced Data Aggregation	Fast and easy (weighted and parallelized) aggregation of multi-type data, with (multipl
Data Transformations	Fast row- and column- arithmetic and (object preserving) apply functionality for vecto
Linear Models	Fast (weighted) linear model fitting with 6 different solvers and a fast F-test to test exc
Time Series and Panel Series	Fast (sequences of) lags / leads and (lagged / leaded and iterated) differences, quasi-dif
List Processing	(Recursive) list search and identification, search and extract list-elements / list-subsetti
Summary Statistics	Fast (grouped and weighted), summary statistics for cross-sectional and complex multi
Recode and Replace Values	Recode multiple values (exact or regex matching) and replace `NaN/Inf/-Inf` and outli
(Memory) Efficient Programming	Efficient comparisons of a vector/matrix with a value, and replacing values/rows in vec

Package Options

- `options("collapse_unused_arg_action")` sets the action taken by generic statistical functions when unknown arguments are passed to a method. The default is `"warning"`.
- `options("collapse_mask")` can be used to export copies of functions starting with `"f"` when loading the package, removing the leading `"f"` (e.g. also exporting `subset` as a clone to `fsubset`). This will mask like-named base R or *dplyr* functions.
- `options("collapse_F_to_FALSE")` can also be called before loading the package to set the lead operator `F` in the package to `FALSE`, to avoid problems with `base::F`.
- When manipulating *data.table*'s, you can set how many columns *collapse* functions overallocate with `option("collapse_DT_alloccol")`. The default is `100L`.

Details

The added top-level documentation infrastructure in *collapse* allows you to effectively navigate the package. Calling `?FUN` brings up the documentation page documenting the function, which contains 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` (e.g. calling `help(.COLLAPSE_TOPICS[1])` brings up this page).

Author(s)

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

See Also

collapse-package

`collapse-options` collapse *Package Options*

Description

- `option("collapse_unused_arg_action")` 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.
- `option("collapse_mask")` can be used to create additional functions in the *collapse* namespace when loading the package, which will mask some existing base R and *dplyr* functions. In particular, *collapse* provides a large number of functions that start with 'f' e.g. `fsubset`, `ftransform`, `fdroplevels` etc.. Specifying `options(collapse_mask = c("fsubset","ftransform","fdroplevels` before loading the package will make additional functions `subset`, `transform`, and `droplevels` available to the user, and mask the corresponding base R functions when the package is attached. In general, all functions starting with 'f' can be passed to the option. There are also a couple of keywords that you can specify to add groups of functions:
  - `"manip"` adds data manipulation functions: `fsubset,ftransform,ftransform<-,ftransformv,fcompute,fcor`
  - `"helper"` adds the functions: `fdroplevels`, `finteraction`, `funique`, `fnlevels`, `fnrow` and `fncol`.
  - `"fast-fun"` adds the functions contained in the macro: `.FAST_FUN`.
  - `"fast-stat-fun"` adds the functions contained in the macro: `.FAST_STAT_FUN`.
  - `"fast-trfm-fun"` adds the functions contained in: `setdiff(.FAST_FUN,.FAST_STAT_FUN)`.
  - `"all"` turns on all of the above.

  Note that none of these options will impact internal *collapse* code, but they may change the way your programs run. `"manip"` is probably the safest option to start with. Specifying `"fast-fun"`, `"fast-stat-fun"`, `"fast-trfm-fun"` or `"all"` are ambitious as they replace basic R functions like `sum` and `max`, introducing *collapse*'s `na.rm = TRUE` default and different behavior for matrices and data frames, and these options also changes some internal macros so that base R functions like `sum` or `max` called inside `fsummarise`, `fmutate` or `collap` will also receive vectorized execution. In other words, if you put `options(collapse_mask = "all")` before loading the package, and you have a collapse-compatible line of dplyr code like `wlddev |> group_by(region,income) |> summarise(across(PCGDP:POP,sum))`, this will now receive fully optimized execution. Note however that because of `collapse`'s `na.rm = TRUE` default, the result will be different unless you add `na.rm = FALSE`. In General, this option is for your convenience, if you want to write visually more appealing code or you want to translate existing dplyr codes to *collapse*. Use with care! For production code I generally recommend not using it.
- `option("collapse_F_to_FALSE")`, if set to `TRUE`, replaces the lead operator `F` in the package with a value `FALSE` when loading the package, which solves issues arising from the use of `F` as a shortcut for `FALSE` in R codes when *collapse* is attached. Note that `F` is just a value in the *base* package namespace, and it should NOT be used in production codes, precisely because users can overwrite it by assignment. An alternative solution to invoking this option would also just be assigning a value `F <-FALSE` in your global environment.
- `option("collapse_DT_alloccol")` sets how many empty columns *collapse* data manipulation functions like `ftransform` allocate when taking a shallow copy of *data.table*'s. The default is `100L`. Note that the *data.table* default is `getOption("datatable.alloccol") = 1024L`. I chose a lower default because shallow copies are taken by each data manipulation function if you manipulate *data.table*'s with collapse, and the cost increases with the number of overallocated columns. With 100 columns, the cost is 2-5 microseconds per copy.

See Also

Collapse Overview, collapse-package

`collapse-renamed` *Renamed Functions*

Description

These functions were renamed moving from collapse 1.5.3 to 1.6.0 to make the namespace more consistent. With collapse 1.7.0 I have depreciated all methods to `fNobs`, `fNdistinct`, `fHDbetween` and `fHDwithin`. The S3 generics and the other functions will be depreciated in 2023 for the earliest. These all now give a message reminding you not to use them in fresh code.

Renaming

```
fNobs -> fnobs
fNdistinct -> fndistinct
pwNobs -> pwnobs
fHDwithin -> fhdwithin
fHDbetween -> fhdbetween
as.factor_GRP -> as_factor_GRP
as.factor_qG -> as_factor_qG
is.GRP -> is_GRP
is.qG -> is_qG
is.unlistable -> is_unlistable
is.categorical -> is_categorical
is.Date -> is_date
as.numeric_factor -> as_numeric_factor
as.character_factor -> as_character_factor
Date_vars -> date_vars
`Date_vars<-` -> `date_vars<-`
```

`colorder` *Fast Reordering of Data Frame Columns*

Description

Efficiently reorder columns in a data frame (no copies). To do this by reference see also `data.table::setcolorder`.

Usage

```
colorder(X, ..., pos = "front")

colorderv(X, neworder = radixorder(names(X)),
          pos = "front", regex = FALSE, ...)
```

Arguments

X	a data frame or list.
...	for colorder: Column names of X in the new order (can also use sequences i.e. col1:coln,colk,...). For colorderv: Further arguments to grep if regex = TRUE.
neworder	a vector of column names, positive indices, a suitable logical vector, a function such as is.numeric, or a vector of regular expressions matching column names (if regex = TRUE).
pos	integer or character. Different options regarding column arrangement if ...length() < ncol(X) (or length(neworder) < ncol(X)).

Int.	*String*	*Description*
1	"front"	move specified columns to the front of X (the default).
2	"end"	move specified columns to the end of X.
3	"exchange"	just exchange the positions of selected columns in X, other columns remain in the same position.
4	"after"	place all further selected columns behind the first selected column.

regex	logical. TRUE will do regular expression search on the column names of X using a (vector of) regular expression(s) passed to neworder. Matching is done using grep. *Note* that multiple regular expressions will be matched in the order they are passed, and funique will be applied to the resulting set of indices.

Value

X with columns re-ordered (no deep copy).

See Also

roworder, Data Frame Manipulation, Collapse Overview

Examples

```
head(colorder(mtcars, vs, cyl:hp, am))
head(colorder(mtcars, vs, cyl:hp, am, pos = "end"))
head(colorder(mtcars, vs, cyl:hp, am, pos = "after"))
head(colorder(mtcars, vs, cyl, pos = "exchange"))

## Same in standard evaluation
head(colorderv(mtcars, c(8, 2:4, 9)))
head(colorderv(mtcars, c(8, 2:4, 9), pos = "end"))
head(colorderv(mtcars, c(8, 2:4, 9), pos = "after"))
head(colorderv(mtcars, c(8, 2), pos = "exchange"))
```

dapply *Data Apply*

Description

dapply efficiently applies functions to columns or rows of matrix-like objects and by default returns an object of the same type and with the same attributes. Alternatively it is possible to return the result in a plain matrix or data.frame. 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, data frame or alike object.
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 mclapply instead of lapply.
mc.cores	integer. Argument to mclapply indicating the number of cores to use for parallel execution. Can use detectCores() to select all available cores.
return	an integer or string indicating the type of object to return. The default 1 -"same" returns the same object type (i.e. class and other attributes are retained, just the names for the dimensions are adjusted). 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 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 lapply or mclapply on these plain lists (which is faster than calling lapply on an object with attributes).
- depending on the requested output type, use matrix, 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).

The 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 apply.

Value

X where FUN was applied to every row or column.

See Also

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

Examples

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

# With classed objects, we have to be a bit careful
## Not run:
dapply(EuStockMarkets, quantile)  # This gives an error because the tsp attribute is misspecified

## End(Not run)
dapply(EuStockMarkets, quantile, return = "matrix")    # These both work fine..
dapply(EuStockMarkets, quantile, return = "data.frame")

# Similarly for grouped tibbles and other data frame based classes
library(dplyr)
gmtcars <- group_by(mtcars,cyl,vs,am)
head(dapply(gmtcars, log))                # Still gives a grouped tibble back
dapply(gmtcars, quantile, MARGIN = 1)    # Here it makes sense to keep the groups attribute
dapply(gmtcars, quantile)                 # This does not make much sense, ...
dapply(gmtcars, quantile,                 # better convert to plain data.frame:
```

```
          return = "data.frame")
```

`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 frames, preserving the data format.
- `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 the Fast Statistical Functions).
- A set of arithmetic operators facilitates **row-wise** `%rr%`, `%r+%`, `%r-%`, `%r*%`, `%r/%` and **column-wise** `%cr%`, `%c+%`, `%c-%`, `%c*%`, `%c/%` **replacing and sweeping operations** involving a vector and a matrix or data frame / list. Since v1.7, the operators `%+=%`, `%-=%`, `%*=%` and `%/=%` do column- and element- wise math by reference, and the function `setop` can also perform sweeping out rows by reference.
- `TRA` is a more advanced S3 generic to efficiently perform **(groupwise) replacing 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 `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** (also a lot faster than `ave`).
- `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). 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**.

- 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. fcumsum flexibly computes cumulative sums. 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*
dapply	No methods, works with matrices and data frames
BY	default, matrix, data.frame, grouped_df
%(r/c)(r/+/-/*//)%	No methods, works with matrices and data frames / lists
TRA	default, matrix, data.frame, grouped_df
fscale/STD	default, matrix, data.frame, pseries, pdata.frame, grouped_d
fwithin/W	default, matrix, data.frame, pseries, pdata.frame, grouped_d
fbetween/B	default, matrix, data.frame, pseries, pdata.frame, grouped_d
fhdwithin/HDW	default, matrix, data.frame, pseries, pdata.frame
fhdbetween/HDB	default, matrix, data.frame, pseries, pdata.frame
flag/L/F, fdiff/D/Dlog, fgrowth/G, fcumsum	default, matrix, data.frame, pseries, pdata.frame, grouped_d

See Also

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

---

descr *Detailed Statistical Description of Data Frame*

Description

descr offers a 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 = 7, perc = TRUE, digits = 2, t.table = 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	double. 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, a logical vector or a function (e.g. 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. If the number of distinct elements is < 2*n, the whole table is printed.
perc	logical. TRUE (default) adds percentages to the frequencies in the table for categorical variables.
digits	integer. The number of decimals to print in statistics and percentage tables.
t.table	logical. TRUE (default) prints a transposed table.
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 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 descr 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, and an attribute 'table' indicating whether table = TRUE (i.e. the object could contain tables for categorical variables).

See Also

qsu, pwcor, Summary Statistics, 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)
```

---

efficient-programming *Small Functions to Make R Programming More Efficient*

Description

A small set of functions to addresses some common inefficiencies in R, such as the creation of logical vectors to compare quantities, unnecessary copies of objects in elementary mathematical or subsetting operations, obtaining information about objects (esp. data frames), or dealing with missing values.

Usage

```
anyv(x, value)              # Faster than any(x == value)
allv(x, value)              # Faster than all(x == value)
allNA(x)                    # Faster than all(is.na(x))
whichv(x, value,            # Faster than which(x == value)
       invert = FALSE)      # or which(x != value)
whichNA(x, invert = FALSE)  # Faster than which((!)is.na(x))
x %==% value             # Infix for whichv(v, value, FALSE), use e.g. in fsubset
x %!=% value                # Infix for whichv(v, value, TRUE)
alloc(value, n)             # Faster than rep_len(value, n)
copyv(X, v, R, ..., invert # Faster than replace(x, x == v, r) or replace(x, v, r[v])
  = FALSE, vind1 = FALSE)   # or replace(x, x != v, r) or replace(x, !v, r[!v])
setv(X, v, R, ..., invert   # Same for x[x (!/=)= v] <- r or x[(!)v] <- r[(!)v]
  = FALSE, vind1 = FALSE)   # modifies x by reference, fastest
setop(X, op, V, ...,       # Faster than X <- X +\-\*\/ V (modifies by reference)
      rowwise = FALSE)      # optionally can also add v to rows of a matrix
X %+=% V                    # Infix for setop(X, "+", V)
X %-=% V                    # Infix for setop(X, "-", V)
X %*=% V                    # Infix for setop(X, "*", V)
X %/=% V                    # Infix for setop(X, "/", V)
na_rm(x)                    # Fast: if(anyNA(x)) x[!is.na(x)] else x,
                            # also removes NULL / empty elements from list
na_omit(X, cols = NULL,     # Faster na.omit for matrices and data frames,
        na.attr = FALSE)    # can use selected columns and attach indices
na_insert(X, prop = 0.1,    # Insert missing values at random
          value = NA)
missing_cases(X,            # The oposite of complete.cases(), faster for
              cols = NULL)  # data frames
vlengths(X, use.names=TRUE) # Faster version of lengths() (in C, no method dispatch)
vtypes(X, use.names = TRUE) # Get data storage types (faster vapply(X, typeof, ...))
fnlevels(x)                 # Faster version of nlevels(x) (for factors)
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)
seq_row(X)                  # Fast integer sequences along rows of X
seq_col(X)                  # Fast integer sequences along columns of X
cinv(x)                # Choleski (fast) inverse of symmetric PD matrix, e.g. X'X
```

Arguments

X, V, R	a vector, matrix or data frame.
x, v	a (atomic) vector or matrix (na_rm also supports lists).
value	a single value of any (atomic) vector type.
invert	logical. TRUE considers elements x != value.
vind1	logical. If length(v) == 1L, setting vind1 = TRUE will interpret v as an index of X and R, rather than a value to search and replace.
op	an integer or character string indicating the operation to perform.

Int.	*String*	*Description*
1	"+"	add v
2	"-"	subtract v
3	"*"	multiply by v
4	"/"	divide by v

rowwise	logical. TRUE performs the operation between v and each row of x. Only applicable if x is a matrix and v a vector such that length(v) == ncol(x).
cols	select columns to check for missing values using column names, indices, a logical vector or a function (e.g. is.numeric). The default is to check all columns, which could be inefficient.
n	integer. The length of the vector to allocate with value.
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".
prop	double. Specify the proportion of observations randomly replaced with NA.
use.names	logical. Preserve names if X is a list.
...	not used, reserved for possible future arguments.

Details

copyv and setv are designed to optimize operations that require replacing a single value in an object e.g. X[X == value] <-r or X[X == value] <-R[R == value] or simply copying parts of an existing object into another object e.g. X[v] <-R[v]. Thus they only cover cases where base R is inefficient by either creating a logical vector or materializing a subset to do some replacement. No alternative is provided in cases where base R is efficient e.g. x[v] <-r or cases provided by set and copy from the *data.table* package. Both functions work equivalently, with the difference that copyv creates a deep copy of the data before making the replacements and returns the copy, whereas setv modifies the data directly without creating a copy and returns the modified object invisibly. Thus setv is considerably more efficient.

copyv and setv perform different tasks, depending on the input. If v is a scalar, the elements of X are compared to v, and the matching ones (or non-matching ones if invert = TRUE) are replaced with R, where R can be either a scalar or an object of the same dimensions as X. If X is a data frame, R can also be a column-vector matching fnrow(X). The second option is if v is either a logical or integer vector of indices with length(v) > 1L, indicating the elements of a vector / matrix (or rows if X is a data frame) to replace with corresponding elements from R. Thus R has to be of equal dimensions as X, but could also be a column-vector if X is a data frame. Setting vind1 = TRUE ensures that v is always interpreted as an index, even if length(v) == 1L.

Note

(1) None of these functions currently support complex vectors. (2) It is possible to compare factors by the levels (e.g. iris$Species %==% "setosa") or using integers (iris$Species %==% 1L). The latter is slightly more efficient. (3) Nothing special is implemented for other objects apart from basic types, e.g. for dates (which are stored as doubles) you need to generate a date object e.g. wlddev$date %==% as.Date("2019-01-01"), wlddev$date %==% "2019-01-01" will give integer(0).

See Also

Data Transformations, Small (Helper) Functions, Collapse Overview

Examples

```
## Which value
whichNA(wlddev$PCGDP)                # Same as which(is.na(wlddev$PCGDP))
whichNA(wlddev$PCGDP, invert = TRUE) # Same as which(!is.na(wlddev$PCGDP))
whichv(wlddev$country, "Chad")       # Same as which(wlddev$county == "Chad")
wlddev$country %==% "Chad"           # Same thing
whichv(wlddev$country, "Chad", TRUE) # Same as which(wlddev$county != "Chad")
wlddev$country %!=% "Chad"           # Same thing
lvec <- wlddev$country == "Chad"     # If we already have a logical vector...
whichv(lvec, FALSE)                  # is fastver than which(!lvec)
rm(lvec)

# Using the %==% operator can yield tangible performance gains
fsubset(wlddev, iso3c %==% "DEU") # 3x faster than:
fsubset(wlddev, iso3c == "DEU")

## 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(mtc_na)
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)
```

`fast-data-manipulation`

Fast Data 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 subset to 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.
- fsummarise is a much faster version of dplyr::summarise when used together with the Fast Statistical Functions and fgroup_by, with whom it also supports super fast weighted aggregation.
- fmutate is a much faster version of dplyr::mutate when used together with the Fast Statistical Functions as well as fast Data Transformation Functions and fgroup_by.
- ftransform is a much faster version of transform, which also supports list input and nested pipelines. 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.
- roworder is a fast substitute for dplyr::arrange, but the syntax is inspired by data.table::setorder.
- colorder efficiently reorders columns in a data frame, see also data.table::setcolorder.
- frename is a fast substitute for dplyr::rename, to efficiently rename various objects. setrename renames objects by reference. relabel and setrelabel do the same thing for variable labels (see also vlabels).

Table of Functions

Function / S3 Generic	*M*
fselect(<-)	No
get_vars(<-), num_vars(<-), cat_vars(<-), char_vars(<-), fact_vars(<-), logi_vars(<-), date_vars(<-)	No
add_vars(<-)	No
fsubset	de
ss	No
fsummarise	No
fmutate, (f/set)ftransform(<-)	No
fcompute(v)	No
roworder(v)	No
colorder(v)	No
(f/set)rename, (set)relabel	No

See Also

Collapse Overview, Quick Data Conversion, Recode and Replace Values

fast-grouping-ordering *Fast Grouping and Ordering*

Description

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

- radixorder, provides fast radix-ordering through direct access to the method order(...,method = "radix"), as well as the possibility to return some attributes very useful for grouping data and finding unique elements. radixorderv exists as a programmers alternative. The function roworder(v) efficiently reorders a data frame based on an ordering computed by radixorderv.
- group provides fast grouping in first-appearance order of rows, based on a hashing algorithm in C. Objects have class 'qG', see below.
- GRP creates *collapse* grouping objects of class 'GRP' based on radixorderv or group. '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 data frame (or data.table / tibble etc.) with a 'GRP' object attached. This grouped frame can be used for grouped operations using *collapse*'s fast functions.
- funique is a faster version of unique. The data frame method also allows selecting unique rows according to a subset of the columns.
- qF, shorthand for 'quick-factor' implements very fast factor generation from atomic vectors using either radix ordering method = "radix" or 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.
- fdroplevels is a substantially faster replacement for droplevels.
- finteraction is a fast alternative to interaction implemented as a wrapper around as_factor_GRP(GRP(...)). It can be 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 regular 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.

Table of Functions

Function / S3 Generic	*Methods*	*Description*
radixorder(v)	No methods, for data frames and vectors	Radix-based ordering -
roworder(v)	No methods, for data frames	Row sorting/reordering
group	No methods, for data frames and vectors	Hash-based grouping +
GRP	default, GRP, factor, qG, grouped_df, pseries, pdata.frame	Fast grouping and a fle
fgroup_by	No methods, for data frames	Fast grouped data fram
funique	default, data.frame	Fast unique values/row
qF	No methods, for vectors	Quick factor generatio
qG	No methods, for vectors	Quick grouping of vec
fdroplevels	factor, data.frame, list	Fast removal of unused
finteraction	No methods, for data frames and vectors	Fast interactions
groupid	No methods, for vectors	Run-length type group
seqid	No methods, for vectors	Run-length type intege

See Also

Collapse Overview, Data Frame Manipulation, Fast Statistical Functions

---

fast-statistical-functions

Fast (Grouped, Weighted) Statistical Functions for Matrix-Like Objects

---

Description

With fsum, fprod, fmean, fmedian, fmode, fvar, fsd, fmin, fmax, fnth, 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 grouped data frames / tibbles (*dplyr*) and *data.table*'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 data frame (class '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	logical. Make group-names and add to the result as names (default method) or row-names (matrix and
drop	*matrix and data.frame methods:* Logical. TRUE drops dimensions and returns an atomic vector if g = 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 suitably aggregated or transformed. Data frame column-attributes and overall attributes are preserved.

Notes

- Panel-decomposed (i.e. between and within) statistics as well as grouped and weighted skewness and kurtosis are implemented in qsu.
- The vector-valued functions and operators fcumsum, 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.

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 data frames - created with dplyr::group_by or fgroup_by
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)
```

Benchmark

```
## This compares fsum with data.table (2 threads) 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")

                    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
 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")

                     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
```

See Also

Collapse Overview, Data Transformations, Time Series and Panel Series

---

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

Description

`fbetween` and `fwithin` are S3 generics to efficiently obtain between-transformed (averaged) or (quasi-)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.)

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, theta = 1, ...)
## 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, theta = 1, ...)

## 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, theta = 1, ...)
## 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, theta = 1, 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, theta = 1, ...)
## 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, theta = 1, 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, theta = 1, ...)
## 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, theta = 1, ...)

## 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, theta = 1, ...)
## 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, theta = 1, stub = "W.", keep.ids = TRUE, keep.w = TRUE, ...)

# Methods for grouped data frame / 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, theta = 1,
        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, theta = 1,
  stub = "W.", keep.group_vars = TRUE, keep.w = TRUE, ...)
```

Arguments

x	a numeric vector, matrix, data frame, panel series (class pseries of package plm), panel data frame (plm::pdata.frame) or grouped data frame (class '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, indices or a logical vector. 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 and w 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 in the output vector will also be NA.
effect	*plm* methods: Select which panel identifier should be used as grouping variable. 1L takes the first variable in the plm::index, 2L the second etc. Index variables

can also be called by name using a character string. If more than one variable is supplied, the corresponding index-factors 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.

`theta` *option to* `fwithin/W`: Double. An optional scalar parameter for quasi-demeaning i.e. `x -theta * xi..` This is useful for variance components ('random-effects') estimators. see Details.

`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 (weighted) 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. In terms of panel data notation: If `x` is a vector in such a panel 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. In regression analysis, centering with `mean = "overall.mean"` will only change the constant term. See Examples.

If `theta != 1`, `fwithin/W` performs quasi-demeaning `x -theta * xi..` If `mean = "overall.mean"`, `x -theta * xi. + theta * x..` is returned, so that the mean of the partially demeaned data is still

equal to the overall data mean `x...` A numeric value passed to `mean` will simply be added back to the quasi-demeaned data i.e. `x - theta * xi. + mean`.

Now in the case of a linear panel model $y_{it} = \beta_0 + \beta_1 X_{it} + u_{it}$ with $u_{it} = \alpha_i + \epsilon_{it}$. If $\alpha_i \neq \alpha = const.$ (there exists individual heterogeneity), then pooled OLS is at least inefficient and inference on β_1 is invalid. If $E[\alpha_i|X_{it}] = 0$ (mean independence of individual heterogeneity α_i), the variance components or 'random-effects' estimator provides an asymptotically efficient FGLS solution by estimating a transformed model $y_{it} - \theta y_{i.} = \beta_0 + \beta_1(X_{it} - \theta X_{i.}) + (u_{it} - \theta u_{i.})$, where $\theta = 1 - \frac{\sigma_\alpha}{\sqrt{(\sigma_\alpha^2 + T\sigma_\epsilon^2)}}$. An estimate of θ can be obtained from the an estimate of $\hat{u}_{it}$ (the residuals from the pooled model). If $E[\alpha_i|X_{it}] \neq 0$, pooled OLS is biased and inconsistent, and taking $\theta = 1$ gives an unbiased and consistent fixed-effects estimator of β_1. See Examples.

Value

`fbetween`/`B` returns `x` with every element replaced by its (groupwise) mean (`xi.`). Missing values are preserved if `fill = FALSE` (the default). `fwithin`/`W` returns `x` where every element was subtracted its (groupwise) mean (`x - theta * xi. + mean` or, if `mean = "overall.mean"`, `x - theta * xi. + theta * x..`). See Details.

References

Mundlak, Yair. 1978. On the Pooling of Time Series and Cross Section Data. *Econometrica* 46 (1): 69-85.

See Also

`fhdbetween/HDB and fhdwithin/HDW`, `fscale/STD`, `TRA`, Data Transformations, Collapse Overview

Examples

```
## Simple centering and averaging
head(fbetween(mtcars))
head(B(mtcars))
head(fwithin(mtcars))
head(W(mtcars))
all.equal(fbetween(mtcars) + fwithin(mtcars), mtcars)

## Groupwise centering and averaging
head(fbetween(mtcars, mtcars$cyl))
head(fwithin(mtcars, mtcars$cyl))
all.equal(fbetween(mtcars, mtcars$cyl) + fwithin(mtcars, mtcars$cyl), mtcars)

head(W(wlddev, ~ iso3c, cols = 9:13))    # Center the 5 series in this dataset by country
head(cbind(get_vars(wlddev,"iso3c"),     # Same thing done manually using fwithin..
      add_stub(fwithin(get_vars(wlddev,9:13), wlddev$iso3c), "W.")))

## Using B() and W() for fixed-effects 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

plm::plm(mpg ~ carb, mtcars, index = "cyl", model = "within")   # "Proof"..

# This takes the interaction of cyl, vs and am as 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 fixed effects weighted by hp:
lm(W(mpg,cyl,hp) ~ W(carb,cyl,hp), data = mtcars)
lm(mpg ~ carb, data = W(mtcars, ~ cyl, ~ hp, stub = FALSE))
lm(mpg ~ carb + B(carb,cyl,hp), data = mtcars)     # WRONG ! Gives a different coefficient!!

## Manual variance components (random-effects) estimation
res <- HDW(mtcars, mpg ~ carb)[[1]]  # Get residuals from pooled OLS
sig2_u <- fvar(res)
sig2_e <- fvar(fwithin(res, mtcars$cyl))
T <- length(res) / fndistinct(mtcars$cyl)
sig2_alpha <- sig2_u - sig2_e
theta <- 1 - sqrt(sig2_alpha) / sqrt(sig2_alpha + T * sig2_e)
lm(mpg ~ carb, data = W(mtcars, ~ cyl, theta = theta, mean = "overall.mean", stub = FALSE))

# A slightly different method to obtain theta...
plm::plm(mpg ~ carb, mtcars, index = "cyl", model = "random")
```

`fcumsum` *Fast (Grouped, Ordered) Cumulative Sum for Matrix-Like Objects*

Description

`fcumsum` is a generic function that computes the (column-wise) cumulative sum of `x`, (optionally) grouped by `g` and/or ordered by `o`. Several options to deal with missing values are provided.

Usage

```
fcumsum(x, ...)

## Default S3 method:
fcumsum(x, g = NULL, o = NULL, na.rm = TRUE, fill = FALSE, check.o = TRUE, ...)

## S3 method for class 'matrix'
fcumsum(x, g = NULL, o = NULL, na.rm = TRUE, fill = FALSE, check.o = TRUE, ...)

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

```
fcumsum(x, g = NULL, o = NULL, na.rm = TRUE, fill = FALSE, check.o = TRUE, ...)

# Methods for compatibility with plm:

## S3 method for class 'pseries'
fcumsum(x, na.rm = TRUE, fill = FALSE, ...)

## S3 method for class 'pdata.frame'
fcumsum(x, na.rm = TRUE, fill = FALSE, ...)

# Methods for grouped data frame / compatibility with dplyr:

## S3 method for class 'grouped_df'
fcumsum(x, o = NULL, na.rm = TRUE, fill = FALSE, check.o = TRUE, keep.ids = TRUE, ...)
```

Arguments

x	a vector / time series, matrix, data frame, panel series (plm::pseries), panel data frame (plm::pdata.frame) or grouped data frame (class '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.
o	a vector or list of vectors providing the order in which the elements of x are cumulatively summed. Will be passed to radixorderv unless check.o = FALSE.
na.rm	logical. Skip missing values in x. Defaults to TRUE and implemented at very little computational cost.
fill	if na.rm = TRUE, setting fill = TRUE will overwrite missing values with the previous value of the cumulative sum, starting from 0.
check.o	logical. Programmers option: FALSE prevents passing o to radixorderv, requiring o to be a valid ordering vector that is integer typed with each element in the range [1,length(x)]. This gives some extra speed, but will terminate R if any element of o is too large or too small.
keep.ids	*pdata.frame / grouped_df methods*: Logical. Drop all identifiers from the output (which includes all grouping variables and variables passed too). *Note*: For grouped / panel data frames identifiers are dropped, but the 'groups' / 'index' attributes are kept.
...	arguments to be passed to or from other methods.

Details

If na.rm = FALSE, fcumsum works like cumsum and propagates missing values. The default na.rm = TRUE skips missing values and computes the cumulative sum on the non-missing values. Missing values are kept. If fill = TRUE, missing values are replaced with the previous value of the cumulative sum (starting from 0), computed on the non-missing values.

By default the cumulative sum is computed in the order in which elements appear in x. If o is provided, the cumulative sum is computed in the order given by radixorderv(o), without the need to first sort x. This applies as well if groups are used (g), in which the cumulative sum is computed separately in each group.

The *pseries* and *pdata.frame* methods assume that the last factor in the `plm::index` is the time-variable and the rest are grouping variables. The time-variable is passed to `radixorderv` and used for ordered computation, so that cumulative sums are accurately computed regardless of whether the panel-data is ordered or balanced.

`fcumsum` explicitly supports integers. Integers in R are bounded at bounded at +-2,147,483,647, and an integer overflow error will be provided if the cumulative sum (within any group) exceeds +-2,147,483,647.

Value

the cumulative sum of values in `x`, (optionally) grouped by `g` and/or ordered by `o`. See Details and Examples.

See Also

`fdiff`, `fgrowth`, Time Series and Panel Series, Collapse Overview

Examples

```
## Non-grouped
fcumsum(AirPassengers)
head(fcumsum(EuStockMarkets))
fcumsum(mtcars)

# Non-grouped but ordered
o <- order(rnorm(nrow(EuStockMarkets)))
all.equal(copyAttrib(fcumsum(EuStockMarkets[o, ], o = o)[order(o), ], EuStockMarkets),
          fcumsum(EuStockMarkets))

## Grouped
head(with(wlddev, fcumsum(PCGDP, iso3c)))

## Grouped and ordered
head(with(wlddev, fcumsum(PCGDP, iso3c, year)))
head(with(wlddev, fcumsum(PCGDP, iso3c, year, fill = TRUE)))
```

`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`, providing 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, log = 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, log = FALSE, rho = 1,
      stubs = length(n) + length(diff) > 2L, ...)
## 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, log = FALSE, rho = 1,
      stubs = length(n) + length(diff) > 2L, ...)
## 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, log = 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, log = FALSE, rho = 1,
      stubs = length(n) + length(diff) > 2L, ...)
## 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 grouped data frame / compatibility with dplyr:

## S3 method for class 'grouped_df'
fdiff(x, n = 1, diff = 1, t = NULL, fill = NA, log = FALSE, rho = 1,
      stubs = length(n) + length(diff) > 2L, 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 / time series, (time series) matrix, data frame, panel series (plm::pseries), panel data frame (plm::pdata.frame) or grouped data frame (class 'grouped_df').
n	integer. A vector indicating the number of lags or leads.
diff	integer. 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/by, to indicate the time-variable(s). For safe computation of differences on unordered time series and panels. Data Frame method also allows one-sided formula i.e. ~time. grouped_df method supports lazy-evaluation i.e. time (no quotes).
cols	*data.frame method*: Select columns to difference using a function, column names, indices or a logical vector. 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.
log	logical. TRUE computes log-differences instead. See Details.
rho	double. Autocorrelation parameter. Set to a value between 0 and 1 for quasi-differencing. 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 grouped / panel data frames identifiers are dropped, but the 'groups' / 'index' 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 unordered.

fdiff/D/Dlog supports balanced panels and unbalanced panels where various individuals are observed for different time-sequences.

For computational details and efficiency considerations see the help page for flag.

It is also possible to compute differences on unordered vectors or irregular 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 log arguments or the Dlog operator. 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.

Prais, S. J. & Winsten, C. B. (1954). Trend Estimators and Serial Correlation. *Cowles Commission Discussion Paper No. 383.* Chicago.

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)))

head(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
head(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!!:
# -> a pseries is only created when subsetting the pdata.frame using $ or [[
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
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) %>%                              # Dropping id's
             select(year,PCGDP,LIFEEX) %>% D(0:1,1:2,year, keep.ids = FALSE)
```

`fdroplevels` *Fast Removal of Unused Factor Levels*

Description

A substantially faster replacement for `droplevels`.

Usage

```
fdroplevels(x, ...)

## S3 method for class 'factor'
fdroplevels(x, ...)

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

Arguments

x	a factor, or data frame / list containing one or more factors.
...	not used.

Details

droplevels passes a factor from which levels are to be dropped to factor, which first calls unique and then match to drop unused levels. Both functions internally use a hash table, which is highly inefficient. fdroplevels does not require mapping values at all, but uses a super fast boolean vector method to determine which levels are unused and remove those levels. In addition, if no unused levels are found, x is simply returned. Any missing values found in x are efficiently skipped in the process of checking and replacing levels. All other attributes of x are preserved.

Value

x will any unused factor levels removed.

Note

If x is malformed i.e. has too few levels, this function can cause a segmentation fault, thus only use with ordinary / proper factors.

See Also

qF, funique, Fast Grouping and Ordering, Collapse Overview

Examples

```
f <- iris$Species[1:100]
fdroplevels(f)
identical(fdroplevels(f), droplevels(f))

fNA <- na_insert(f)
fdroplevels(fNA)
identical(fdroplevels(fNA), droplevels(fNA))

identical(fdroplevels(ss(iris, 1:100)), droplevels(ss(iris, 1:100)))
```

---

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 data frame (class '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 skips missing values and returns 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	logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.

drop — *matrix and data.frame method:* Logical. TRUE drops dimensions and returns 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 data frames - created with dplyr::group_by or fgroup_by
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 (Weighted) F-test for 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, w = 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.
w	numeric. A vector of (frequency) weights.
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 fhdwithin. Sensible options might be the lm.method argument or further control parameters to fixest::demean, the workhorse function underlying fhdwithin for higher-order centering tasks.

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 (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.

Note that an intercept is always added by fhdwithin, so it is not necessary to include an intercept in data supplied to exc / X.

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

flm, fhdwithin, 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))
fFtest(fdiff(AirPassengers), qF(cycle(AirPassengers)))  # Seasonality in first-difference

## 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, and also provides more flexibility 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, power = 1, stubs = TRUE, ...)
## Default S3 method:
G(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA, logdiff = FALSE,
  scale = 100, power = 1, stubs = TRUE, ...)

## S3 method for class 'matrix'
fgrowth(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA,
        logdiff = FALSE, scale = 100, power = 1,
        stubs = length(n) + length(diff) > 2L, ...)
## S3 method for class 'matrix'
G(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA, logdiff = FALSE,
  scale = 100, power = 1, stubs = TRUE, ...)

## S3 method for class 'data.frame'
fgrowth(x, n = 1, diff = 1, g = NULL, t = NULL, fill = NA,
        logdiff = FALSE, scale = 100, power = 1,
        stubs = length(n) + length(diff) > 2L, ...)
## 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, power = 1, 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,
        power = 1, stubs = TRUE, ...)
## S3 method for class 'pseries'
G(x, n = 1, diff = 1, fill = NA, logdiff = FALSE, scale = 100,
  power = 1, stubs = TRUE, ...)

## S3 method for class 'pdata.frame'
fgrowth(x, n = 1, diff = 1, fill = NA, logdiff = FALSE, scale = 100,
        power = 1, stubs = length(n) + length(diff) > 2L, ...)
## S3 method for class 'pdata.frame'
G(x, n = 1, diff = 1, cols = is.numeric, fill = NA, logdiff = FALSE,
  scale = 100, power = 1, stubs = TRUE, keep.ids = TRUE, ...)

# Methods for grouped data frame / compatibility with dplyr:

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

```
fgrowth(x, n = 1, diff = 1, t = NULL, fill = NA, logdiff = FALSE,
        scale = 100, power = 1, stubs = length(n) + length(diff) > 2L,
        keep.ids = TRUE, ...)
## S3 method for class 'grouped_df'
G(x, n = 1, diff = 1, t = NULL, fill = NA, logdiff = FALSE,
  scale = 100, power = 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 data frame (class 'grouped_df').
n	integer. A vector indicating the number of lags or leads.
diff	integer. A vector of integers > 1 indicating the order of taking growth rates, e.g. diff = 2 means computing the growth rate of the growth rate.
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/by, to indicate the time-variable(s). For safe computation of growth rates on unordered time series and panels. Data Frame method also allows one-sided formula i.e. ~time. grouped_df method supports lazy-evaluation i.e. time (no quotes).
cols	*data.frame method*: Select columns to compute growth rates using a function, column names, indices or a logical vector. 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-difference growth rates 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.
power	numeric. Apply a power to annualize or compound growth rates e.g. fgrowth(AirPassengers,12,power = 1/12) is equivalent to ((AirPassengers/flag(AirPassengers,12))^(1/12)-1)*100.
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 grouped / panel data frames identifiers are dropped, but the 'groups' / 'index' 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])^power -1)*scale, and, if logdiff = TRUE approximate growth rates using repeat(diff) log(x[i]/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)

head(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 external 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
head(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!!:
# -> a pseries is only created when subsetting the pdata.frame using $ or [[
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
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 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, lm.method = "qr", ...)
## Default S3 method:
fhdwithin(x, fl, w = NULL, na.rm = TRUE, fill = FALSE, lm.method = "qr", ...)
```

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

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

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

# Methods for compatibility with plm:

## S3 method for class 'pseries'
fhdbetween(x, effect = seq_col(attr(x, "index")), w = NULL, na.rm = TRUE,
           fill = TRUE, ...)
## S3 method for class 'pseries'
fhdwithin(x, effect = seq_col(attr(x, "index")), w = NULL, na.rm = TRUE,
          fill = TRUE, ...)
## S3 method for class 'pseries'
HDB(x, effect = seq_col(attr(x, "index")), w = NULL, na.rm = TRUE, fill = TRUE, ...)
## S3 method for class 'pseries'
HDW(x, effect = seq_col(attr(x, "index")), w = NULL, na.rm = TRUE, fill = TRUE, ...)

## S3 method for class 'pdata.frame'
fhdbetween(x, effect = seq_col(attr(x, "index")), w = NULL, na.rm = TRUE, fill = TRUE,
           variable.wise = TRUE, ...)
## S3 method for class 'pdata.frame'
fhdwithin(x, effect = seq_col(attr(x, "index")), w = NULL, na.rm = TRUE, fill = TRUE,
          variable.wise = TRUE, ...)
## S3 method for class 'pdata.frame'
HDB(x, effect = seq_col(attr(x, "index")), w = NULL, cols = is.numeric, na.rm = TRUE,
```

```
    fill = TRUE, variable.wise = TRUE, stub = "HDB.", ...)
## S3 method for class 'pdata.frame'
HDW(x, effect = seq_col(attr(x, "index")), 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 and the Note.
w	a vector of (non-negative) weights.
cols	*data.frame methods*: Select columns to center (partial-out) or predict using column names, indices, a logical vector 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	*(p)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.
effect	*plm* methods: Select which panel identifiers should be used for centering. 1L takes the first variable in the plm::index, 2L the second etc. Index variables can also be called by name using a character vector.
stub	a prefix or stub to rename all transformed columns. FALSE will not rename columns.
lm.method	character. The linear fitting method. Supported are "chol" and "qr". See flm.
...	further arguments passed to fixest::demean and chol / qr. Possible choices are tol to set a uniform numerical tolerance for the entire fitting process, or nthreads and iter to govern the higher-order centering process.

Details

fhdbetween/HDB and fhdwithin/HDW are powerful functions for high-dimensional linear prediction problems involving large factors and datasets, but can just as well handle ordinary regression problems. They are implemented as efficient wrappers around fbetween / fwithin, flm and fixest::demean (imported for higher-order centering tasks).

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 either fbetween / fwithin for a single factor or fixest::demean for multiple factors. 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 model.matrix and expanded to matrices of dummies but projected beforehand.

The formula interface to the data.frame method (only supported by the operators HDW | HDB) provides ease of use and allows for additional modeling complexity. For example it is possible to project out formulas like HDW(data,~ fct1*var1 + fct2:fct3 + var2:fct2:fct3 + 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

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

- fhdwithin/HDW can center data on multiple factors and also partial out continuous variables and factor-continuous interactions while fwithin/W only centers on one factor or the interaction of a set of factors, and 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, fwithin, fscale, TRA, flm, 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]))
head(HDW(qM(mtcars[3:4]), mtcars[1:2]))
head(HDW(iris[1:2], iris[3:4]))           # Partialling columns 3 and 4 out of colums 1 and 2
head(HDW(iris[1:2], iris[3:5]))               # Adding the Species factor -> fixed effect

head(HDW(wlddev, PCGDP + LIFEEX ~ iso3c + qF(year))) # Partialling out 2 fixed effects
head(HDW(wlddev, PCGDP + LIFEEX ~ iso3c + qF(year), variable.wise = TRUE)) # Variable-wise
head(HDW(wlddev, PCGDP + LIFEEX ~ iso3c + qF(year) + ODA)) # Adding ODA as a continuous regressor
head(HDW(wlddev, PCGDP + LIFEEX ~ iso3c:qF(decade) + qF(year) + ODA)) # Country-decade and year FE's

head(HDW(wlddev, PCGDP + LIFEEX ~ iso3c*year))          # Country specific time trends
head(HDW(wlddev, PCGDP + LIFEEX ~ iso3c*poly(year, 3))) # Country specific cubic trends

# More complex examples
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 = length(n) > 1L, ...)
## 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 = length(n) > 1L, ...)
## 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 = length(n) > 1L, ...)
## 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 grouped data frame / compatibility with dplyr:
```

```
## S3 method for class 'grouped_df'
flag(x, n = 1, t = NULL, fill = NA, stubs = length(n) > 1L, 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 / time series, (time series) matrix, data frame, panel series (plm::pseries), panel data frame (plm::pdata.frame) or grouped data frame (class 'grouped_df'). Data must not be numeric i.e you can also lag a date variable, character data etc...
n	integer. A vector indicating the lags / leads to compute (passing negative integers to flag or L computes leads, passing negative integers to F computes lags).
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/by, to indicate the time-variable(s). For safe computation of differences on unordered time series and panels. Data Frame method also allows one-sided formula i.e. ~time. grouped_df method supports lazy-evaluation i.e. time (no quotes).
cols	*data.frame method*: Select columns to difference using a function, column names, indices or a logical vector. 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 grouped / panel data frames identifiers are dropped, but the 'groups' / 'index' 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 (time series), flag/L/F returns a (time series) 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/lead 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 unordered.

It is also possible to lag unordered or irregular time series utilizing only the `t` argument to identify the temporal dimension of the data.

Since v1.5.0 `flag/L/F` provide full built-in support for irregular time series and unbalanced panels. The suggested workaround using the `seqid` function is therefore no longer necessary.

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 minimum and maximum 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 `t` is not factor or integer but instead `is.double(t) && !is.object(t)`, it is assumed to be integer represented by double and converted using `as.integer(t)`. For other objects such as dates, `t` is grouped using `qG` or `GRP` (for multiple time identifiers). Similarly, if `g/by` is not factor or 'GRP' object, `qG` or `GRP` will be called to group the respective identifier. Since grouping is more expensive than computing lags, prepare the data for optimal performance (or use *plm* classes). See also the Note.

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. 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.

Note

Since v1.7.0, if `is.double(t) && !is.object(t)`, it is coerced to integer using `as.integer(t)`. This is to avoid the inefficiency of ordered grouping, and owes to the fact that in most data imported into R, the time (year) variables are coded as double although they should be integer.

See Also

`fdiff`, `fgrowth`, 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))
```

```
head(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
head(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
head(L(PCGDP))                                                 # Lagging the panel series
summary(lm(PCGDP ~ ., L(pwlddev, 0:2, 9:10, keep.ids = FALSE))) # Running the lm again
# THIS DOES NOT WORK: -> a pseries is only created when subsetting the pdata.frame using $ or [[
summary(lm(PCGDP ~ L(PCGDP,1:2) + L(LIFEEX,0:2), pwlddev))    # ..so L.default is used here..
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)
```

`flm` *Fast (Weighted) Linear Model Fitting*

Description

`flm` is a fast linear model command that takes matrices as input and (by default) only returns a coefficient matrix. 6 different efficient fitting methods are implemented: 4 using base R linear

algebra, and 2 utilizing the *RcppArmadillo* and *RcppEigen* packages. The function itself only has an overhead of 5-10 microseconds, and is thus well suited as a bootstrap workhorse.

Usage

```
flm(y, X, w = NULL, add.icpt = FALSE, return.raw = FALSE,
    method = c("lm", "solve", "qr", "arma", "chol", "eigen"),
    eigen.method = 3L, ...)
```

Arguments

y	a response vector or matrix. Multiple dependent variables are only supported by methods "lm", "solve", "qr" and "chol".
X	a matrix of regressors.
w	a weight vector.
add.icpt	logical. TRUE adds an intercept column named '(Intercept)' to X.
return.raw	logical. TRUE returns the original output from the different methods. For 'lm', 'arma' and 'eigen', this includes additional statistics such as residuals, fitted values or standard errors. The other methods just return coefficients but in different formats.
method	an integer or character string specifying the method of computation:

Int.	*String*	*Description*
1	"lm"	uses .lm.fit.
2	"solve"	solve(crossprod(X), crossprod(X, y)).
3	"qr"	qr.coef(qr(X), y).
4	"arma"	uses RcppArmadillo::fastLmPure.
5	"chol"	chol2inv(chol(crossprod(X))) %*% crossprod(X, y) (quite fast but requires crossprod(X) to be
6	"eigen"	uses RcppEigen::fastLmPure (very fast but potentially unstable, depending on the method).

eigen.method	integer. Select the method of computation used by RcppEigen::fastLmPure:

Int.	*Description*
0	column-pivoted QR decomposition.
1	unpivoted QR decomposition.
2	LLT Cholesky.
3	LDLT Cholesky.
4	Jacobi singular value decomposition (SVD).
5	method based on the eigenvalue-eigenvector decomposition of X'X.

	See vignette("RcppEigen-Introduction",package = "RcppEigen") for details on these methods and benchmark results. Run source(system.file("examples","lmBenchmark.R = "RcppEigen")) to re-run the benchmark on your machine.
...	further arguments passed to other methods. Sensible choices are tol = value - a numerical tolerance for the solution - applicable with methods "lm", "solve" and

"qr" (default is 1e-7), or LAPACK = TRUE with method "qr" to use LAPACK routines to for the qr decomposition (typically faster than LINPACK (the default)).

Value

If return.raw = FALSE, a matrix of coefficients with the rows corresponding to the columns of X, otherwise the raw results from the various methods are returned.

Note

Method "qr" supports sparse matrices, so for an X matrix with many dummy variables consider method "qr" passing as(X,"dgCMatrix") instead of just X.

See Also

fhdwithin/HDW, fFtest, Data Transformations, Collapse Overview

Examples

```
coef <- flm(mtcars$mpg, qM(mtcars[c("hp","carb")]),
            mtcars$wt, add.icpt = TRUE)
coef

lmcoef <- coef(lm(mpg ~ hp + carb, weights = wt, mtcars))
lmcoef

all.equal(drop(coef), lmcoef)


all_obj_equal(lapply(1:6, function(i)
  flm(mtcars$mpg, qM(mtcars[c("hp","carb")]),
      mtcars$wt, add.icpt = TRUE, method = i)))
```

---

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 data frame (class '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	logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.
drop	*matrix and data.frame method:* Logical. TRUE drops dimensions and returns 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 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 frames 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..
head(fmean(mtcars, g, TRA = "-"))      # etc..

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

## method for grouped data frames - created with dplyr::group_by or fgroup_by
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 and faster !
```

```
fselect(mpg,hp) %>% fmean(hp, "-")
```

---

fmedian *Fast (Grouped, Weighted) 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 and/or weighted by w. The TRA argument can further be used to transform x using its (grouped, weighted) median value.

Usage

```
fmedian(x, ...)

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

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

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

## S3 method for class 'grouped_df'
fmedian(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 data frame (class '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, but only if x is also missing.
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	logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.
drop	*matrix and data.frame method:* Logical. TRUE drops dimensions and returns 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

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 vector (x or a column of x).

Grouped computations are currently performed by mapping the data to a sparse-array and then partially sorting each row (group) of that array. Because of compiler optimizations this requires less memory than a full deep copy done with no groups.

The weighted median is defined as the element k from a set of sorted elements, such that the sum of weights of all elements larger and all elements smaller than k is <= sum(w)/2. If the half-sum of weights (sum(w)/2) is reached exactly for some element k, then (summing from the lower end) both k and k+1 would qualify as the weighted median (and some possible additional elements with zero weights following k would also qualify). fmedian solves these ties by taking a simple arithmetic mean of all elements qualifying as the weighted median.

The weighted median is computed using radixorder to first obtain an ordering of all elements, so it is considerably more computationally expensive than the unweighted version. With groups, the entire vector is also ordered, and the weighted median is computed in a single ordered pass through the data (after group-summing the weights, skipping weights for which x is missing).

If x is a matrix or data frame, these computations are performed independently for each column. When applied to data frames 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 (w weighted) median value of x, grouped by g, or (if TRA is used) x transformed by its median, grouped by g.

See Also

fnth, fmean, fmode, Fast Statistical Functions, Collapse Overview

Examples

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

## data.frame method
fmedian(mtcars)
head(fmedian(mtcars, TRA = "-"))
fmedian(mtcars, g)
fmedian(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)
head(fmedian(m, TRA = "-"))
fmedian(m, g) # etc..

library(dplyr)
# grouped_df method
mtcars %>% group_by(cyl,vs,am) %>% fmedian()
mtcars %>% group_by(cyl,vs,am) %>% fmedian(hp)             # Weighted
mtcars %>% fgroup_by(cyl,vs,am) %>% fmedian()              # Faster grouping!
mtcars %>% fgroup_by(cyl,vs,am) %>% fmedian(TRA = "-")     # De-median
mtcars %>% fgroup_by(cyl,vs,am) %>% fselect(mpg, hp) %>%    # Faster selecting
      fmedian(hp, "-")  # Weighted de-median mpg, using hp as weights
```

`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 data frame (class '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	logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.
drop	*matrix and data.frame method:* Logical. TRUE drops dimensions and returns 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 max and 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 frames 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)
head(fmax(mtcars, TRA = "%"))
fmax(mtcars, g)
fmax(mtcars, g, use.g.names = FALSE) # No row-names generated

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

## method for grouped data frames - created with dplyr::group_by or fgroup_by
```

```
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. Ties between multiple possible modes can be resolved by taking the minimum, maximum, (default) first or last occurring mode.

Usage

```
fmode(x, ...)

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

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

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

## 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,
      ties = "first", ...)
```

Arguments

x	a vector, matrix, data frame or grouped data frame (class '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` logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.

`ties` an integer or character string specifying the method to resolve ties between multiple possible modes i.e. multiple values with the maximum frequency or sum of weights:

Int.	*String*	*Description*
1	"first"	take the first occurring mode.
2	"min"	take the smallest of the possible modes.
3	"max"	take the largest of the possible modes.
4	"last"	take the last occurring mode.

Note: `"min"/"max"` don't work with character data. For logical data `TRUE` will be chosen unless `ties = "min"`. See Details.

`drop` *matrix and data.frame method:* Logical. `TRUE` drops dimensions and returns 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 `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)]`.

It is possible that multiple values have the same mode that is the maximum frequency or sum of weights. Typical cases are simply when all values are either all the same or all distinct. In such cases, the default option `ties = "first"` returns the first occurring value in the data reaching the maximum frequency count or sum of weights. For example in a sample `x = c(1,3,2,2,4,4,1,7)`, the first mode is 2 as `fmode` goes through the data from left to right. `ties = "last"` on the other hand gives 1. It is also possible to take the minimum or maximum mode, i.e. `fmode(x,ties = "min")` returns 1, and `fmode(x,ties = "max")` returns 4. It should be noted that options `ties = "min"` and `ties = "max"` work well for numeric/integer/factor data as well as date/time variables, but give unintuitive results for character data (no strict alphabetic sorting, similar to using < and > to compare character values in R). These options are also best avoided if missing values are counted

(na.rm = FALSE) since no proper logical comparison with missing values is possible: With numeric data it depends, since in C++ any comparison with NA_real_ evaluates to FALSE, NA_real_ is chosen as the min or max mode only if it is also the first mode, and never otherwise. For integer data, NA_integer_ is stored as the smallest integer in C++, so it will always be chosen as the min mode and never as the max mode. For character data, NA_character_ is stored as the string "NA" in C++ and thus the behavior depends on the other character content. fmode also implements a fast method for logical values which does not support the options "first"/"last" i.e. TRUE is returned unless ties = "min".

This all seamlessly generalizes to grouped computations, which are performed by mapping the data to a sparse-array 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 in grouped operations). If a data frame is passed to fmode and drop = TRUE (the default), unlist will be called on the result, which might not be sensible depending on the data at hand.

Value

The (w weighted) 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

```
x <- c(1, 3, 2, 2, 4, 4, 1, 7, NA, NA, NA)
fmode(x)                            # Default is ties = "first"
fmode(x, ties = "last")
fmode(x, ties = "min")
fmode(x, ties = "max")
fmode(x, na.rm = FALSE)             # Here NA is the mode, regardless of ties option
fmode(x[-length(x)], na.rm = FALSE) # Not anymore..

## World Development Data
attach(wlddev)
## default vector method
fmode(PCGDP)                       # Numeric mode
head(fmode(PCGDP, iso3c))          # Grouped numeric mode
head(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)                      # Calling unlist -> coerce to character vector
fmode(wlddev, drop = FALSE)        # Gives one row
head(fmode(wlddev, iso3c))         # Grouped mode
head(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 data frame (class '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	logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.
drop	*matrix and data.frame method:* Logical. TRUE drops dimensions and returns 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 performed by mapping the data to a sparse-array 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 data frames - created with dplyr::group_by or fgroup_by
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 data frame (class '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	logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.

drop — *matrix and data.frame method:* Logical. TRUE drops dimensions and returns 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 data frames - created with dplyr::group_by or fgroup_by
library(dplyr)
airquality %>% group_by(Month) %>% fnobs()
wlddev %>% group_by(country) %>%
           select(PCGDP,LIFEEX,GINI,ODA) %>% fnobs()
```

fnth *Fast (Grouped, Weighted) N'th Element/Quantile for Matrix-Like Objects*

Description

fnth (column-wise) returns the n'th smallest element from a set of unsorted elements x corresponding to an integer index (n), or to a probability between 0 and 1. If n is passed as a probability, ties can be resolved using the lower, upper, or (default) average of the possible elements. These are discontinuous and fast methods to estimate a sample quantile.

Usage

```
fnth(x, n = 0.5, ...)

## Default S3 method:
fnth(x, n = 0.5, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, ties = "mean", ...)

## S3 method for class 'matrix'
fnth(x, n = 0.5, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, drop = TRUE, ties = "mean", ...)

## S3 method for class 'data.frame'
fnth(x, n = 0.5, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = TRUE, drop = TRUE, ties = "mean", ...)

## S3 method for class 'grouped_df'
fnth(x, n = 0.5, w = NULL, TRA = NULL, na.rm = TRUE,
     use.g.names = FALSE, keep.group_vars = TRUE, keep.w = TRUE,
     ties = "mean", ...)
```

Arguments

x	a numeric vector, matrix, data frame or grouped data frame (class 'grouped_df').
n	the element to return using a single integer index such that 1 < n < NROW(x), or a probability 0 < n < 1. See Details.
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 — logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.

ties — an integer or character string specifying the method to resolve ties between adjacent qualifying elements:

Int.	*String*	*Description*
1	"mean"	take the arithmetic mean of all qualifying elements.
2	"min"	take the smallest of the elements.
3	"max"	take the largest of the elements.

drop — *matrix and data.frame method:* Logical. TRUE drops dimensions and returns 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

This is an R port to std::nth_element, an efficient partial sorting algorithm in C++. It is also used to calculated the median (in fact the default fnth(x,n = 0.5) is identical to fmedian(x), so see also the details for fmedian).

fnth generalizes the principles of median value calculation to find arbitrary elements. It offers considerable flexibility by providing both simple order statistics and simple discontinuous quantile estimation. Regarding the former, setting n to an index between 1 and NROW(x) will return the n'th smallest element of x, about 2x faster than sort(x,partial = n)[n]. As to the latter, setting n to a probability between 0 and 1 will return the corresponding element of x, and resolve ties between multiple qualifying elements (such as when n = 0.5 and x is even) using the arithmetic average ties = "mean", or the smallest ties = "min" or largest ties = "max" of those elements.

If n > 1 is used and x contains missing values (and na.rm = TRUE, otherwise NA is returned), n is internally converted to a probability using p = (n-1)/(NROW(x)-1), and that probability is applied to the set of complete elements (of each column if x is a matrix or data frame) to find the as.integer(p*(fnobs(x)-1))+1L'th element (which corresponds to option ties = "min"). Note that it is necessary to subtract and add 1 so that n = 1 corresponds to p = 0 and n = NROW(x) to p = 1.

When using grouped computations (supplying a vector or list to g subdividing x) and n > 1 is used, it is transformed to a probability p = (n-1)/(NROW(x)/ng-1) (where ng contains the number of unique groups in g) and ties = "min" is used to sort out clashes. This could be useful for example to return the n'th smallest element of each group in a balanced panel, but with unequal group sizes it more intuitive to pass a probability to n.

If weights are used, the same principles apply as for weighted median calculation: A target partial sum of weights p*sum(w) is calculated, and the weighted n'th element is the element k such that all

elements smaller than k have a sum of weights <= p*sum(w), and all elements larger than k have a sum of weights <= (1 -p)*sum(w). If the partial-sum of weights (p*sum(w)) is reached exactly for some element k, then (summing from the lower end) both k and k+1 would qualify as the weighted n'th element (and some possible additional elements with zero weights following k would also qualify). If n > 1, the lowest of those elements is chosen (congruent with the unweighted behavior), but if 0 < n < 1, the ties option regulates how to resolve such conflicts, yielding lower-weighted, upper-weighted or (default) average weighted n'th elements.

The weighted n'th element is computed using radixorder to first obtain an ordering of all elements, so it is considerably more computationally expensive than the unweighted version. With groups, the entire vector is also ordered, and the weighted n'th element is computed in a single ordered pass through the data (after calculating partial-group sums of the weights, skipping weights for which x is missing).

If x is a matrix or data frame, these computations are performed independently for each column. Column-attributes and overall attributes of a data frame are preserved (if g is used or drop = FALSE).

Value

The (w weighted) n'th element of x, grouped by g, or (if TRA is used) x transformed by its n'th element, grouped by g.

See Also

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

Examples

```
## default vector method
mpg <- mtcars$mpg
fnth(mpg)                         # Simple nth element: Median (same as fmedian(mpg))
fnth(mpg, 5)                      # 5th smallest element
sort(mpg, partial = 5)[5]         # Same using base R, fnth is 2x faster.
fnth(mpg, 0.75)                   # Third quartile
fnth(mpg, 0.75, w = mtcars$hp)    # Weighted third quartile: Weighted by hp
fnth(mpg, 0.75, TRA = "-")        # Simple transformation: Subtract third quartile
fnth(mpg, 0.75, mtcars$cyl)             # Grouped third quartile
fnth(mpg, 0.75, mtcars[c(2,8:9)])       # More groups..
g <- GRP(mtcars, ~ cyl + vs + am)       # Precomputing groups gives more speed !
fnth(mpg, 0.75, g)
fnth(mpg, 0.75, g, mtcars$hp)           # Grouped weighted third quartile
fnth(mpg, 0.75, g, TRA = "-")           # Groupwise subtract third quartile
fnth(mpg, 0.75, g, mtcars$hp, "-")      # Groupwise subtract weighted third quartile

## data.frame method
fnth(mtcars, 0.75)
head(fnth(mtcars, 0.75, TRA = "-"))
fnth(mtcars, 0.75, g)
fnth(fgroup_by(mtcars, cyl, vs, am), 0.75)   # Another way of doing it..
fnth(mtcars, 0.75, g, use.g.names = FALSE)   # No row-names generated

## matrix method
m <- qM(mtcars)
```

```
fnth(m, 0.75)
head(fnth(m, 0.75, TRA = "-"))
fnth(m, 0.75, g) # etc..

library(dplyr)
## grouped_df method
mtcars %>% group_by(cyl,vs,am) %>% fnth(0.75)
mtcars %>% group_by(cyl,vs,am) %>% fnth(0.75, hp)           # Weighted
mtcars %>% fgroup_by(cyl,vs,am) %>% fnth(0.75)              # Faster grouping!
mtcars %>% fgroup_by(cyl,vs,am) %>% fnth(0.75, TRA = "/")   # Divide by third quartile
mtcars %>% fgroup_by(cyl,vs,am) %>% fselect(mpg, hp) %>%     # Faster selecting
    fnth(0.75, hp, "/")  # Divide mpg by its third weighted group-quartile, using hp as weights
```

`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, weighted) 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 data frame (class '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	logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.
drop	*matrix and data.frame method:* Logical. TRUE drops dimensions and returns 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 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 frames 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 (w weighted) 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)
head(fprod(mtcars, TRA = "/"))
fprod(mtcars, g)
fprod(mtcars, g, use.g.names = FALSE) # No row-names generated

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

## method for grouped data frames - created with dplyr::group_by or fgroup_by
library(dplyr)
mtcars %>% group_by(cyl,vs,am) %>% fprod(hp)   # Weighted grouped product
mtcars %>% fgroup_by(cyl,vs,am) %>% fprod(hp)  # Equivalent and faster
mtcars %>% fgroup_by(cyl,vs,am) %>% fprod(TRA = "/")
mtcars %>% fgroup_by(cyl,vs,am) %>% fselect(mpg) %>% fprod()
```

---

frename *Fast Renaming and Relabelling Objects*

Description

A fast substitute for `dplyr::rename`. `setrename` renames objects by reference. These functions also work with objects other than data frames that have a 'names' attribute. `relabel` and `setrelabel` do that same for labels attached to list elements / data frame columns.

Usage

```
frename(.x, ..., cols = NULL)
rnm(.x, ..., cols = NULL)     # Shortcut for frename()

setrename(.x, ..., cols = NULL)
```

```
relabel(.x, ..., cols = NULL, attrn = "label")

setrelabel(.x, ..., cols = NULL, attrn = "label")
```

Arguments

.x	an R object with a 'names' attribute.
...	either tagged vector expressions of the form name = newname / name = newlabel, or a single function (+ optional arguments to the function) applied to all names/labels (of columns/elements selected in cols).
cols	If ... is a function, select a subset of columns/elements to rename/relabel using names, indices, a logical vector or a function applied to the columns if .x is a data frame (e.g. is.numeric).
attrn	character. Name of attribute to store labels or retrieve labels from.

Value

.x renamed / relabelled. setrename and setrelabel return .x invisibly.

Note

Note that both relabel and setrelabel modify .x by reference. This is because labels are attached to columns themselves, making it impossible to avoid permanent modification by taking a shallow copy of the encompassing list / data.frame. On the other hand frename makes a shallow copy whereas setrename also modifies by reference.

See Also

Data Frame Manipulation, Collapse Overview

Examples

```
## Using tagged expressions
head(frename(iris, Sepal.Length = SL, Sepal.Width = SW,
                   Petal.Length = PL, Petal.Width = PW))
head(frename(iris, Sepal.Length = "S L", Sepal.Width = "S W",
                   Petal.Length = "P L", Petal.Width = "P W"))

## Using a function
head(frename(iris, tolower))
head(frename(iris, tolower, cols = 1:2))
head(frename(iris, tolower, cols = is.numeric))
head(frename(iris, paste, "new", sep = "_", cols = 1:2))

## Renaming by reference
# setrename(iris, tolower)
# head(iris)
# rm(iris)
```

```
## Relabelling (by reference)
# namlab(relabel(wlddev, PCGDP = "GDP per Capita", LIFEEX = "Life Expectancy"))
# namlab(relabel(wlddev, toupper))
```

`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. Arbitrary target means and standard deviations can be set, with special options for grouped scaling and centering. It is also possible to scale data without centering i.e. perform mean-preserving scaling.

Note: For centering without scaling see `fwithin/W`. For simple not mean-preserving scaling use `fsd(...,TRA = "/")`. To sweep pre-computed means and scale-factors out of data see `TRA`.

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 grouped data frame / 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 data frame (class '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, indices or a logical vector. 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 group-id. 1L takes the first variable in the plm::index, 2L the second etc.. Index variables can also be called by name using a character string. More than one variable can be supplied.
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, 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, fsd, TRA, Fast Statistical Functions, Data Transformations, Collapse Overview

Examples

```
## Simple Scaling & Centering / Standardizing
head(fscale(mtcars))               # Doesn't rename columns
head(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$iso3c)) # This calculates the within standard deviations
head(qsu(fscale(get_vars(wlddev, 9:12),         # This group-centers on the overall mean and
   wlddev$iso3c,                          # group-scales to the within standard deviation
   mean = "overall.mean", sd = "within.sd"),     # -> data harmonized in the first 2 moments
    by = wlddev$iso3c))


## 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,POP) %>% STD(POP) # weighting by POP ->
# ..keeps the weight column unless keep.w = FALSE
```

fselect-get_vars-add_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, ...)
      gv(x, vars, return = "data", ...)     # Shortcut for get_vars
   gvr(x, vars, return = "data", ...)    # Shortcut for get_vars(\dots, regex = TRUE)

get_vars(x, vars, regex = FALSE, ...) <- value
      gv(x, vars, ...) <- value             # Shortcut for get_vars<-
   gvr(x, vars, ...) <- value         # Shortcut for get_vars<-(\dots, regex = TRUE)

## 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, or 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 (if available).

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 e.g. fselect(mtcars,newname = mpg,hp,carb:vs). 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 (ab-

stracting 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

The functions here only check the length of the first column, which is one of the reasons why they are so fast. 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).

See Also

fsubset, ftransform, Data Frame Manipulation, Collapse Overview

Examples

```
## Wold Development Data
head(fselect(wlddev, Country = country, Year = year, ODA)) # 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
head(gvr(wlddev, "[[:upper:]]"))                             # Same thing

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, 14:17) <- 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
```

`fsubset` *Fast Subsetting Matrix-Like Objects*

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 frames it is significantly (4-5 times) faster than `subset` or `dplyr::filter`. The methods also provide enhanced functionality compared to `subset`. The function `ss` provides an (internal generic) programmers alternative to `[` that does not drop dimensions and is significantly faster than `[` for data frames.

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 (replaces `[` with drop = FALSE, programmers choice)
ss(x, i, j)
```

Arguments

x	object to be subsetted.
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 or 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 x.
j	a vector of column names, positive or negative indices or a suitable logical vector to subset the columns of x. *Note:* Negative indices are converted to positive ones using `j <-seq_along(x)[j]`.

Details

fsubset is a generic function, with methods supplied for vectors, matrices, and data frames (including lists). It represents an improvement in both speed and functionality over subset. The function ss is an improvement of [to subset (vectors) matrices and data frames without dropping dimensions. It is significantly faster than [.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 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, and new column names may be assigned e.g. `fsubset(data,col1 > 20,newname = col2,col3:col6)` (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 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 fdroplevels 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).

Note

No replacement method fsubset<- or ss<- is offered in *collapse*. For efficient subset replacement (without copying) use data.table::set, which can also be used with data frames and tibbles. To search and replace certain elements without copying, and to efficiently copy elements / rows from an equally sized vector / data frame, see setv.

See Also

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

Examples

```
fsubset(airquality, Temp > 90, Ozone, Temp)
fsubset(airquality, Temp > 90, OZ = Ozone, Temp) # With renaming
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)

ss(airquality, 1:10, 2:3)         # Significantly faster than airquality[1:10, 2:3]
fsubset(airquality, 1:10, 2:3)    # This is possible but not advised
```

---

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 (e.g., 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 data frame (class '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	logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.
drop	*matrix and data.frame method:* Logical. TRUE drops dimensions and returns 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 sum which just runs through without any checks).

The weighted sum (e.g., survey total) is computed as sum(x * w), but in one pass and about twice as efficient. 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 are therefore extremely fast. See Benchmark and Examples below.

When applied to data frames with groups or drop = FALSE, fsum preserves all column attributes (such as variable labels), unless columns have a class (checked using is.object). The attributes of the data frame itself are also preserved.

Since v1.6.0 fsum explicitly supports integers. Integers are summed using the long long type in C which is bounded at +-9,223,372,036,854,775,807 (so ~4.3 billion times greater than the minimum/maximum R integer bounded at +-2,147,483,647). If the value of the sum is outside +-2,147,483,647, a double containing the result is returned, otherwise an integer is returned. With groups, an integer overflow error is provided if the sum in any group is outside +-2,147,483,647.

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, fmean, 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 data frames - created with dplyr::group_by or fgroup_by
library(dplyr)
mtcars %>% group_by(cyl,vs,am) %>% fsum(hp)  # Weighted grouped sum (total)
mtcars %>% fgroup_by(cyl,vs,am) %>% fsum(hp) # Equivalent and faster !!
mtcars %>% fgroup_by(cyl,vs,am) %>% fsum(TRA = "%")
mtcars %>% fgroup_by(cyl,vs,am) %>% fselect(mpg) %>% fsum()
```

Benchmark

```
## This compares fsum with data.table (2 threads) 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")

                    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
 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")

                     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
```

`fsummarise` *Fast Summarise*

Description

`fsummarize` is a much faster version of `dplyr::summarise`, when used together with the Fast Statistical Functions.

Usage

```
fsummarise(.data, ..., keep.group_vars = TRUE)
smr(.data, ..., keep.group_vars = TRUE)        # Shortcut
```

Arguments

.data	a (grouped) data frame or named list of columns. Grouped data can be created with fgroup_by or dplyr::group_by.
...	name-value pairs of summary functions, or across statements. For fast performance use the Fast Statistical Functions.
keep.group_vars	logical. FALSE removes grouping variables after computation.

Value

If .data is grouped by fgroup_by or dplyr::group_by, the result is a data frame of the same class and attributes with rows reduced to the number of groups. If .data is not grouped, the result is a data frame of the same class and attributes with 1 row.

Note

Since v1.7, fsummarise is fully featured, allowing expressions using functions and columns of the data as well as external scalar values (just like dplyr::summarise). **NOTE** however that once a Fast Statistical Function is used, the execution will be vectorized instead of split-apply-combine computing over groups. Please see the first Example.

See Also

across, collap, Data Frame Manipulation, Fast Statistical Functions, Collapse Overview

Examples

```
library(magrittr) # Note: Used because |> is not available on older R versions
## Since v1.7, fsummarise supports arbitrary expressions, and expressions
## containing fast statistical functions receive vectorized execution:

# (a) This is an expression using base R functions which is executed by groups
mtcars %>% fgroup_by(cyl) %>% fsummarise(res = mean(mpg) + min(qsec))

# (b) Here, the use of fmean causes the whole expression to be executed
# in a vectorized way i.e. the expression is translated to something like
# fmean(mpg, g = cyl) + min(mpg) and executed, thus the result is different
# from (a), because the minimum is calculated over the entire sample
mtcars %>% fgroup_by(cyl) %>% fsummarise(mpg = fmean(mpg) + min(qsec))

# (c) For fully vectorized execution, use fmin. This yields the same as (a)
mtcars %>% fgroup_by(cyl) %>% fsummarise(mpg = fmean(mpg) + fmin(qsec))

# In across() statements it is fine to mix different functions, each will
# be executed on its own terms (i.e. vectorized for fmean and standard for sum)
mtcars %>% fgroup_by(cyl) %>% fsummarise(across(mpg:hp, list(fmean, sum)))

# Note that this still detects fmean as a fast function, the names of the list
# are irrelevant, but the function name must be typed or passed as a character vector,
# Otherwise functions will be executed by groups e.g. function(x) fmean(x) won't vectorize
```

```
mtcars %>% fgroup_by(cyl) %>% fsummarise(across(mpg:hp, list(mu = fmean, sum = sum)))

# We can force none-vectorized execution by setting .apply = TRUE
mtcars %>% fgroup_by(cyl) %>% fsummarise(across(mpg:hp, list(mu = fmean, sum = sum), .apply = TRUE))

# Another argument of across(): Order the result first by function, then by column
mtcars %>% fgroup_by(cyl) %>%
     fsummarise(across(mpg:hp, list(mu = fmean, sum = sum), .transpose = FALSE))

#--------------------------------------------------------------------------
# Examples that also work for pre 1.7 versions

# Simple use
fsummarise(mtcars, mean_mpg = fmean(mpg),
                   sd_mpg = fsd(mpg))

# Using base functions (not a big difference without groups)
fsummarise(mtcars, mean_mpg = mean(mpg),
                   sd_mpg = sd(mpg))

# Grouped use
mtcars %>% fgroup_by(cyl) %>%
  fsummarise(mean_mpg = fmean(mpg),
             sd_mpg = fsd(mpg))

# This is still efficient but quite a bit slower on large data (many groups)
mtcars %>% fgroup_by(cyl) %>%
  fsummarise(mean_mpg = mean(mpg),
             sd_mpg = sd(mpg))

# Weighted aggregation
mtcars %>% fgroup_by(cyl) %>%
  fsummarise(w_mean_mpg = fmean(mpg, wt),
             w_sd_mpg = fsd(mpg, wt))


## Can also group with dplyr::group_by, but at a conversion cost, see ?GRP
library(dplyr)
mtcars %>% group_by(cyl) %>%
  fsummarise(mean_mpg = fmean(mpg),
             sd_mpg = fsd(mpg))

# Again less efficient...
mtcars %>% group_by(cyl) %>%
  fsummarise(mean_mpg = mean(mpg),
             sd_mpg = sd(mpg))
```

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

Description

ftransform is a much faster version of 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. fcompute computes and returns new columns. These functions evaluate all arguments simultaneously, allow list-input (nested pipelines) and natively disregard grouped data.

Catering to the *tidyverse* user, v1.7.0 introduced fmutate, providing familiar functionality i.e. arguments are evaluated sequentially, computation on grouped data is done by groups, and functions can be applied to multiple columns using across. See also the Details.

Usage

```
# Modify and return data frame
ftransform(.data, ...)
ftransformv(.data, vars, FUN, ..., apply = TRUE)
tfm(.data, ...)              # Shortcut for ftransform
tfmv(.data, vars, FUN, ..., apply = TRUE)

# Modify data frame by reference
settransform(.data, ...)
settransformv(.data, vars, FUN, ..., apply = TRUE)
settfm(.data, ...)           # Shortcut for settransform
settfmv(.data, vars, FUN, ..., apply = TRUE)

# Replace/add modified columns in/to a data frame
ftransform(.data) <- value
tfm(.data) <- value          # Shortcut for ftransform<-

# Compute columns, returned as a new data frame
fcompute(.data, ..., keep = NULL)
fcomputev(.data, vars, FUN, ..., apply = TRUE, keep = NULL)

# New: dplyr-style mutate (sequential evaluation + across() feature)
fmutate(.data, ..., .keep = "all")
mtt(.data, ..., .keep = "all") # Shortcut for fmutate
```

Arguments

.data	a data frame or named list of columns.
...	further arguments of the form column = value. The value can be a combination of other columns, a scalar value, or NULL, which deletes column. Alternatively it is also possible to place a single list here, which will be treated like a list of column = value arguments. For ftransformv and fcomputev, ... can be used

to pass further arguments to FUN. *Note:* The ellipsis (...) is always evaluated within the data frame (.data) environment. See Examples. fmutate supports across statements, and evaluates tagged vector expressions sequentially.

vars — variables to be transformed by applying FUN to them: select using names, indices, a logical vector or a selector function (e.g. is.numeric). Since v1.7 vars is evaluated within the .data environment, permitting expressions on columns e.g. c(col1,col3:coln).

FUN — a single function yielding a result of length NROW(.data) or 1. See also apply.

apply — logical. TRUE (default) will apply FUN to each column selected in vars; FALSE will apply FUN to the subsetted data frame i.e. FUN(get_vars(.data,vars),...). The latter is useful for *collapse* functions with data frame or grouped / panel data frame methods, yielding performance gains and enabling grouped transformations. See Examples.

value — a named list of replacements, it will be treated like an evaluated list of column = value arguments.

keep — select columns to preserve using column names, indices or a function (e.g. is.numeric). By default computed columns are added after the preserved ones, unless they are assigned the same name in which case the preserved columns will be replaced in order.

.keep — Either one of "all","used","unused" or "none" (see mutate), or columns names/indices/function as keep.

Details

The ... arguments to ftransform are tagged vector expressions, which are evaluated in the data frame .data. The tags are matched against names(.data), and for those that match, the values replace the corresponding variable in .data, whereas the others are appended to .data. It is also possible to delete columns by assigning NULL to them, i.e. ftransform(data,colk = NULL) removes colk from the data. *Note* that names(.data) and the names of the ... arguments are checked for uniqueness beforehand, yielding an error if this is not the case.

Since *collapse* v1.3.0, is is also possible to pass a single named list to ..., i.e. ftransform(data,newdata). This list will be treated like a list of tagged vector expressions. *Note* the different behavior: ftransform(data,list(newcol = col1)) is the same as ftransform(data,newcol = col1), whereas ftransform(data,newcol = as.list(col1)) creates a list column. Something like ftransform(data,as.list(col1)) gives an error because the list is not named. See Examples.

The function ftransformv added in v1.3.2 provides a fast replacement for the functions dplyr::mutate_at and dplyr::mutate_if (without the grouping feature) facilitating mutations of groups of columns (dplyr::mutate_all is already accounted for by dapply). See Examples.

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 afterwards).

The function fcompute(v) works just like ftransform(v), but returns only the changed / computed columns without modifying or appending the data in .data. See Examples.

The function fmutate added in v1.7.0, provides functionality familiar from *dplyr* 1.0.0 and higher. It evaluates tagged vector expressions sequentially and does operations by groups on a grouped

frame (thus it is slower than ftransform if you have many tagged expressions or a grouped data frame). Note however that *collapse* does not depend on *rlang*, so fancy things like data masking or lambda expressions are not available. *Note also* that fmutate operates differently on grouped data whether you use .FAST_FUN or base R functions / functions from other packages. With .FAST_FUN (including .OPERATOR_FUN, excluding fhdbetween / fhdwithin / HDW / HDB), fmutate performs an efficient vectorized execution, i.e. the grouping object from the grouped data frame is passed to the g argument of these functions, and for .FAST_STAT_FUN also TRA = "replace_fill" is set (if not overwritten by the user), yielding internal grouped computation by these functions without the need for splitting the data by groups. For base R and other functions, fmutate performs classical split-apply combine computing i.e. the relevant columns of the data are selected and split into groups, the expression is evaluated for each group, and the result is re-combined and suitably expanded to match the original data frame. **Note** that it is not possible to mix vectorized and standard execution in the same expression!! Vectorized execution is performed if **any** .FAST_FUN or .OPERATOR_FUN is part of the expression, thus a code like mtcars |> gby(cyl) |> fmutate(new = fmin(mpg) / min(mpg)) will be expanded to something like mtcars %>% gby(cyl) %>% ftransform(new = fmin(mpg,g = GRP(.),TRA = "replace_fill") / min(mpg)) and then executed, i.e. fmin(mpg) will be executed in a vectorized way, and min(mpg) will not be executed by groups at all.

Value

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

Note

ftransform does not do anything per se with a grouped data frame. This is on purpose as it affords greater flexibility and performance in programming with the .FAST_FUN, through which *collapse* supports *various kinds* of fully vectorized grouped transformations (see TRA for a list of available transformations). In particular, you can run a nested pipeline inside ftransform, and decide which expressions should be grouped, and you can use the ad-hoc grouping functionality of the .FAST_FUN, allowing operations where different groupings are applied simultaneously in an expression. See Examples or the answer provided here.

fmutate on the other hand supports grouped operations just like dplyr::mutate, but works in two different ways depending on whether you use .FAST_FUN in an expression or other functions. See the Examples section of fsummarise for an illustration.

See Also

across, fsummarise, Data Frame Manipulation, Collapse Overview

Examples

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

# With collapse's grouped and weighted functions, complex operations are done on the fly
```

```
head(ftransform(airquality, # Grouped operations by month:
                Ozone_Month_median = fmedian(Ozone, Month, TRA = "replace_fill"),
                Ozone_Month_sd = fsd(Ozone, Month, TRA = "replace"),
                Ozone_Month_centered = fwithin(Ozone, Month)))

# Grouping by month and above/below average temperature in each month
head(ftransform(airquality, Ozone_Month_high_median =
            fmedian(Ozone, list(Month, Temp > fbetween(Temp, Month)), TRA = "replace_fill")))

## ftransformv can be used to modify multiple columns using a function
head(ftransformv(airquality, 1:3, log))
head(`[<-`(airquality, 1:3, value = lapply(airquality[1:3], log))) # Same thing in base R

head(ftransformv(airquality, 1:3, log, apply = FALSE))
head(`[<-`(airquality, 1:3, value = log(airquality[1:3])))        # Same thing in base R

# Using apply = FALSE yields meaningful performance gains with collapse functions
# This calls fwithin.default, and repeates the grouping by month 3 times:
head(ftransformv(airquality, 1:3, fwithin, Month))

# This calls fwithin.data.frame, and only groups one time -> 5x faster!
head(ftransformv(airquality, 1:3, fwithin, Month, apply = FALSE))

library(magrittr) # Pipe operators
# This also works for grouped and panel data frames (calling fwithin.grouped_df)
airquality %>% fgroup_by(Month) %>%
  ftransformv(1:3, fwithin, apply = FALSE) %>% head

# But this gives the WRONG result (calling fwithin.default). Need option apply = FALSE!!
airquality %>% fgroup_by(Month) %>%
  ftransformv(1:3, fwithin) %>% head

# For grouped modification of single columns in a grouped dataset, we can use GRP():
airquality %>% fgroup_by(Month) %>%
  ftransform(W_Ozone = fwithin(Ozone, GRP(.)),                 # Grouped centering
           sd_Ozone_m = fsd(Ozone, GRP(.), TRA = "replace"), # In-Month standard deviation
            sd_Ozone = fsd(Ozone, TRA = "replace"),          # Overall standard deviation
              sd_Ozone2 = fsd(Ozone, TRA = "replace_fill"),     # Same, overwriting NA's
            sd_Ozone3 = fsd(Ozone)) %>% head                # Same thing (calling alloc())

rm(airquality)

## For more complex mutations we can use ftransform with compound pipes
airquality %>% fgroup_by(Month) %>%
  ftransform(get_vars(., 1:3) %>% fwithin %>% flag(0:2)) %>% head

airquality %>% ftransform(STD(., cols = 1:3) %>% replace_NA(0)) %>% head

# The list argument feature also allows flexible operations creating multiple new columns
airquality %>% # The variance of Wind and Ozone, by month, weighted by temperature:
 ftransform(fvar(list(Wind_var = Wind, Ozone_var = Ozone), Month, Temp, "replace")) %>% head

# Same as above using a grouped data frame (a bit more complex)
```

```
airquality %>% fgroup_by(Month) %>%
 ftransform(fselect(., Wind, Ozone) %>% fvar(Temp, "replace") %>% add_stub("_var", FALSE)) %>%
  fungroup %>% head

# This performs 2 different multi-column grouped operations (need c() to make it one list)
ftransform(airquality, c(fmedian(list(Wind_Day_median = Wind,
                                      Ozone_Day_median = Ozone), Day, TRA = "replace"),
                         fsd(list(Wind_Month_sd = Wind,
                              Ozone_Month_sd = Ozone), Month, TRA = "replace"))) %>% head

## settransform(v) works like ftransform(v) but modifies a data frame in the global environment..
settransform(airquality, Ratio = Ozone / Temp, Ozone = NULL, Temp = NULL)
head(airquality)
rm(airquality)

# Grouped and weighted centering
settransformv(airquality, 1:3, fwithin, Month, Temp, apply = FALSE)
head(airquality)
rm(airquality)

# Suitably lagged first-differences
settransform(airquality, get_vars(airquality, 1:3) %>% fdiff %>% flag(0:2))
head(airquality)
rm(airquality)

# Same as above using magrittr::`%<>%`
airquality %<>% ftransform(get_vars(., 1:3) %>% fdiff %>% flag(0:2))
head(airquality)
rm(airquality)

# It is also possible to achieve the same thing via a replacement method (if needed)
ftransform(airquality) <- get_vars(airquality, 1:3) %>% fdiff %>% flag(0:2)
head(airquality)
rm(airquality)

## fcompute only returns the modified / computed columns
head(fcompute(airquality, Ozone = -Ozone))
head(fcompute(airquality, new = -Ozone, Temp = (Temp-32)/1.8))
head(fcompute(airquality, new = -Ozone, new2 = 1))

# Can preserve existing columns, computed ones are added to the right if names are different
head(fcompute(airquality, new = -Ozone, new2 = 1, keep = 1:3))

# If given same name as preserved columns, preserved columns are replaced in order...
head(fcompute(airquality, Ozone = -Ozone, new = 1, keep = 1:3))

# Same holds for fcomputev
head(fcomputev(iris, is.numeric, log)) # Same as:
iris %>% get_vars(is.numeric) %>% dapply(log) %>% head()

head(fcomputev(iris, is.numeric, log, keep = "Species"))   # Adds in front
head(fcomputev(iris, is.numeric, log, keep = names(iris))) # Preserve order
```

```
# Keep a subset of the data, add standardized columns
head(fcomputev(iris, 3:4, STD, apply = FALSE, keep = names(iris)[3:5]))
```

---

funique *Fast Unique Elements / Rows*

Description

funique is a substantially faster alternative to unique. It is generic with a default vector and a data frame methods.

Usage

```
funique(x, ...)

## Default S3 method:
funique(x, sort = FALSE, method = "auto", ...)

## S3 method for class 'data.frame'
funique(x, cols = NULL, sort = FALSE, method = "auto", ...)

## S3 method for class 'sf'
funique(x, cols = NULL, sort = FALSE, method = "auto", ...)
```

Arguments

x	a atomic vector or data frame / list of equal-length columns.
sort	logical. TRUE orders the unique elements / rows. FALSE returns unique values in order of first occurrence.
method	an integer or character string specifying the method of computation:

Int.	*String*	*Description*
1	"auto"	automatic selection: hash if sort = FALSE else radix.
2	"radix"	use radix ordering to determine unique values. Supports sort = FALSE but only for character data.
3	"hash"	use index hashing to determine unique values. Supports sort = TRUE but only for atomic vectors (defau

cols	compute unique rows according to a subset of columns. Columns can be selected using column names, indices, a logical vector or a selector function (i.e. is.character). *Note:* All columns are returned.
...	arguments passed to radixorderv, e.g. decreasing or na.last. Only applicable if method = "radix".

Details

If x is a data frame / list and all rows are already unique, then x is returned. Otherwise a copy of x with duplicate rows removed is returned. See group for some additional computational details.

The *sf* method simply ignores the geometry column when determining unique values.

Value

x with duplicate elements/rows removed.

See Also

group, Fast Grouping and Ordering, Collapse Overview.

Examples

```
funique(mtcars$cyl)
funique(gv(mtcars, c(2,8,9)))
funique(mtcars, cols = c(2,8,9))
```

---

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 data frame (class '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	logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.
drop	*matrix and data.frame method:* Logical. TRUE drops dimensions and returns 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 frames 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). 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.

References

Welford, B. P. (1962). Note on a method for calculating corrected sums of squares and products. *Technometrics*. 4 (3): 419-420. doi:10.2307/1266577.

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
head(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 data frames - created with dplyr::group_by or fgroup_by
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
```

---

get_elem *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. `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 = FALSE, keep.tree = FALSE,
         keep.class = FALSE, regex = FALSE, ...)

## Check for the existence of elements
has_elem(l, elem, recursive = TRUE, DF.as.list = FALSE, 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. `TRUE` treats data frames like (sub-)lists; `FALSE` 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

For a lack of better terminology, *collapse* defines 'regular' R objects as objects that are either atomic or a list. `reg_elem` with `recursive = TRUE` 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

```
m <- qM(mtcars)
get_elem(list(list(list(m))), is.matrix)
get_elem(list(list(list(m))), is.matrix, keep.tree = TRUE)

l <- list(list(2,list("a",1)),list(1,list("b",2)))
has_elem(l, is.logical)
has_elem(l, is.numeric)
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")
```

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)
# aperm(qsu(GGDC10S, ~ Variable, ~ Variable + Country, vlabels = TRUE))


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")
```

group *Fast Hash-Based Grouping*

Description

group() scans the rows of a data frame (or atomic vector / list of atomic vectors), assigning to each unique row an integer id - starting with 1 and proceeding in first-appearance order of the rows. The function is written in C and optimized for R's data structures. It is the workhorse behind functions like GRP / fgroup_by, collap, qF, qG, finteraction and funique, when called with argument sort = FALSE.

Usage

```
group(x, starts = FALSE, group.sizes = FALSE)
```

Arguments

x	an atomic vector or data frame / list of equal-length atomic vectors.
starts	logical. If TRUE, an additional attribute 'starts' is attached giving a vector of group starts (index of first-occurrence of unique rows).
group.sizes	logical. If TRUE, an additional attribute 'group.sizes' is attached giving the size of each group.

Details

A data frame is grouped on a column-by-column basis, starting from the leftmost column. For each new column the grouping vector obtained after the previous column is also fed back into the hash function so that unique values are determined on a running basis. The algorithm terminates as soon as the number of unique rows reaches the size of the data frame. Missing values are also grouped just like any other values. Invoking arguments starts and/or group.sizes requires an additional pass through the final grouping vector.

Value

An object is of class 'qG' see qG.

Author(s)

The Hash Function was taken from the excellent *kit* package by Morgan Jacob.

See Also

Fast Grouping and Ordering, Collapse Overview

Examples

```
# Let's replicate what funique does
g <- group(wlddev, starts = TRUE)
if(attr(g, "N.groups") == fnrow(wlddev)) wlddev else
   ss(wlddev, attr(g, "starts"))
```

---

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`	an 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 Grouping and Ordering, Collapse Overview

Examples

```
groupid(airquality$Month)
groupid(airquality$Month, start = 0)
groupid(wlddev$country)[1:100]

## 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 or group. 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 statistical and transformation functions / operators, as well as to collap, BY and TRA.

fgroup_by is similar to dplyr::group_by but faster. It creates a grouped data frame 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. Notable among these is GRP.grouped_df, which returns a 'GRP' object from a grouped data frame created with dplyr::group_by (or fgroup_by), and the duo GRP.factor and as_factor_GRP.

gsplit efficiently splits a vector based on a grouping object.

Usage

```
GRP(X, ...)

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

## S3 method for class 'factor'
GRP(X, ..., group.sizes = TRUE, drop = FALSE, return.groups = TRUE,
    call = TRUE)

## S3 method for class 'qG'
GRP(X, ..., group.sizes = TRUE, return.groups = TRUE, call = TRUE)
```

```
## S3 method for class 'pseries'
GRP(X, effect = 1L, ..., group.sizes = TRUE, return.groups = TRUE,
    call = TRUE)

## S3 method for class 'pdata.frame'
GRP(X, effect = 1L, ..., group.sizes = TRUE, return.groups = TRUE,
    call = TRUE)

## S3 method for class 'grouped_df'
GRP(X, ..., return.groups = TRUE, call = TRUE)

# Identify, get length, group names, and convert GRP object to factor
is_GRP(x)
## S3 method for class 'GRP'
length(x)
GRPnames(x, force.char = TRUE)
as_factor_GRP(x, ordered = FALSE)

# Efficiently split a vector using a grouping object
gsplit(x, g, use.g.names = FALSE, ...)

# Fast, class-agnostic 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, method = "auto")
# Shortcut for fgroup_by
      gby(X, ..., sort = TRUE, decreasing = FALSE, na.last = TRUE,
          return.order = FALSE, method = "auto")

# Get grouping columns from a grouped data frame created with dplyr::group_by or fgroup_by
fgroup_vars(X, return = "data")

# Ungroup grouped data frame created with dplyr::group_by or fgroup_by
fungroup(X, ...)

## 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, g	a 'GRP' object. For gsplit, x can be a vector of any type, or NULL to return the integer indices of the groups.
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. If `FALSE`, groups are not ordered but simply grouped in the order of first appearance of unique elements / rows. This often provides a performance gain if the data was not sorted beforehand. See also `method`.

`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.

`method` character. The algorithm to use for grouping: either `"radix"`, `"hash"` or `"auto"`. `"auto"` will chose `"radix"` when `sort = TRUE`, yielding ordered grouping via `radixorderv`, and `"hash"`-based grouping in first-appearance order via `group` otherwise. It is possibly to put `method = "radix"` and `sort = FALSE`, which will group character data in first appearance order but sort numeric data (a good hybrid option). `method = "hash"` currently does not support any sorting, thus putting `sort = TRUE` will simply be ignored.

`group.sizes` logical. `TRUE` tabulates factor levels using `tabulate` to create a vector of group sizes; `FALSE` leaves that slot empty when converting from factors.

`drop` logical. `TRUE` efficiently drops unused factor levels beforehand using `fdroplevels`.

`call` logical. `TRUE` calls `match.call` and saves it in the final slot of the 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 takes the first variable in the `plm::index`, 2L the second etc., identifiers can also be passed as a character string. 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

`use.g.names` logical. `TRUE` returns a named list, like `split`. `FALSE` is slightly more efficient.

`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. For gsplit: further arguments passed to GRP (if g is not already a 'GRP' object).

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.

Next to GRP, there is the function fgroup_by as a significantly faster alternative to dplyr::group_by. It creates a grouped data frame 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::summarize or dplyr::mutate (the grouping object and method of computing results is different). The converse is not true, you can group data with dplyr::group_by and then apply *collapse* functions (which call a C function that converts the grouping object). Note also the fgroup_by is class-agnostic, i.e. the classes of the data frame or list passed are preserved, and all standard methods (like subsetting with `[` or print methods) apply to the grouped object. Apart from the class 'grouped_df' which is added behind any classes the object might inherit (apart from 'data.frame'), a class 'GRP_df' is added in front. This class responds to print method and subset (`[`) methods. Both first call the corresponding method for the object and then print / attach the grouping information. print.GRP_df prints one line below the object indicating the grouping variables, followed, in square brackets, by some statistics on the group sizes: [N | Mean (SD) Min-Max]. The mean is rounded to a full number and the standard deviation (SD) to one digit. Minimum and maximum are only displayed if the SD is non-zero.

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 <-fnlevels(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 interacting multiple columns using the paste function to produce unique factor levels (if multiple grouping columns were used).

The method GRP.grouped_df takes the 'groups' attribute from a grouped data frame and converts it to a 'GRP' object. If the grouped data frame 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()` or `NULL`	The `GRP()` call, obtained from `match.call()`, or `NUL`

See Also

radixorder, qF, Fast Grouping and Ordering, Collapse Overview

Examples

```
## default method
GRP(mtcars$cyl)
GRP(mtcars, ~ cyl + vs + am)     # Or GRP(mtcars, c("cyl","vs","am")) or GRP(mtcars, c(2,8:9))
g <- GRP(mtcars, ~ cyl + vs + am)  # Saving the object
print(g)                           # Printing it
plot(g)                            # Plotting it
GRPnames(g)                        # Retain group names
fsum(mtcars, g)             # Compute the sum of mtcars, grouped by variables cyl, vs and am
gsplit(mtcars$mpg, g)              # Use the object to split a vector
gsplit(NULL, g)                    # The indices of the groups

## Convert factor to GRP object and vice-versa
GRP(iris$Species)
as_factor_GRP(g)

## 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

mtcars %>% fgroup_by(cyl,vs,am)            # Print method for grouped data frame
```

`is_unlistable` *Unlistable Lists*

Description

A (nested) list with atomic objects in all final nodes of the list-tree is unlistable - checked with `is_unlistable`.

Usage

```
is_unlistable(l, DF.as.list = FALSE)
```

Arguments

`l`	a list.
`DF.as.list`	logical. `TRUE` treats data frames like (sub-)lists; `FALSE` like atomic elements.

Details

`is_unlistable` with `DF.as.list = TRUE` is defined as `all(rapply(l,is.atomic))`, whereas `DF.as.list = FALSE` yields checking using `all(unlist(rapply2d(l,function(x) is.atomic(x) || is.list(x)),use.names = FALSE))`, assuming that data frames are lists composed of atomic elements. If `l` contains data frames, the latter can be a lot faster than applying `is.atomic` to every data frame column.

Value

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

See Also

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

Examples

```
l <- list(1, 2, list(3, 4, "b", FALSE))
is_unlistable(l)
l <- list(1, 2, list(3, 4, "b", FALSE, e ~ b))
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 = FALSE)
```

Arguments

l	a list.
DF.as.list	logical. TRUE treats data frames like (sub-)lists; FALSE like atomic elements.

Details

The depth or level or nesting of a list or list-like structure (e.g. 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 = FALSE, data frames 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)
```

`list-processing` *List Processing*

Description

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

- **Search and Identification**
  - `is_unlistable` checks whether a (nested) list is composed of atomic objects in all final nodes, and thus unlistable to an atomic vector using `unlist`.
  - `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. (*Tip*: 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.
- **Splitting and Transposition**
  - `rsplit` recursively splits a vector or data frame into subsets according to combinations of (multiple) vectors / factors - by default returning a (nested) list. If `flatten = TRUE`, the list is flattened yielding the same result as `split`. `rsplit` is also faster than `split`, particularly for data frames.
  - `t_list` efficiently transposes nested lists of lists, such as those obtained from splitting a data frame by multiple variables using `rsplit`.
- **Apply Functions**
  - `rapply2d` is a recursive version of `lapply` with two key differences to `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 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_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
atomic_elem	Top-level subset atomic elements
list_elem	Top-level subset list/list-like elements
reg_elem	Recursive version of atomic_elem: Subset / extract 'regular' part of list
irreg_elem	Subset / extract non-regular part of list
rsplit	Recursively split vectors or data frames / lists
t_list	Transpose lists of lists
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

---

pad *Pad Matrix-Like Objects with a Value*

Description

The pad function inserts elements / rows filled with value into a vector matrix or data frame X at positions given by i. It is particularly useful to expand objects returned by statistical procedures which remove missing values to the original data dimensions.

Usage

```
pad(X, i, value = NA, method = c("auto", "xpos", "vpos"))
```

Arguments

X	a vector, matrix, data frame or list of equal-length columns.
i	either an integer (positive or negative) or logical vector giving positions / rows of X into which value's should be inserted, or, alternatively, a positive integer vector with length(i) == NROW(X), but with some gaps in the indices into which value's can be inserted, or a logical vector with sum(i) == NROW(X) such that value's can be inserted for FALSE values in the logical vector. See also method and Examples.
value	a scalar value to be replicated and inserted into X at positions / rows given by i. Default is NA.
method	an integer or string specifying what the use of i. The options are:

Int.	*String*	*Description*
1	"auto"	automatic method selection: If i is positive integer and length(i) == NROW(X) or if i is logical and su
1	"xpos"	i is a vector of positive integers or a logical vector giving the positions of the the elements / rows of X.
2	"vpos"	i is a vector of positive / negative integers or a logical vector giving the positions at which values's / r

Value

X with elements / rows filled with value inserted at positions given by i.

See Also

append, Recode and Replace Values, Small (Helper) Functions, Collapse Overview

Examples

```
v <- 1:3

pad(v, 1:2)       # Automatic selection of method "vpos"
pad(v, -(1:2))    # Same thing
pad(v, c(TRUE, TRUE, FALSE, FALSE, FALSE)) # Same thing

pad(v, c(1, 3:4)) # Automatic selection of method "xpos"
pad(v, c(TRUE, FALSE, TRUE, TRUE, FALSE))  # Same thing

head(pad(wlddev, 1:3)) # Insert 3 missing rows at the beginning of the data
head(pad(wlddev, 2:4)) # ... at rows positions 2-4

# pad() is mostly useful for statistical models which only use the complete cases:
mod <- lm(LIFEEX ~ PCGDP, wlddev)
# Generating a residual column in the original data (automatic selection of method "vpos")
settfm(wlddev, resid = pad(resid(mod), mod$na.action))
# Another way to do it:
r <- resid(mod)
```

```
i <- as.integer(names(r))
resid2 <- pad(r, i)        # automatic selection of method "xpos"
# here we need to add some elements as flast(i) < nrow(wlddev)
resid2 <- c(resid2, rep(NA, nrow(wlddev)-length(resid2)))
# See that these are identical:
identical(unattrib(wlddev$resid), resid2)

# Can also easily get a model matrix at the dimensions of the original data
mm <- pad(model.matrix(mod), mod$na.action)
```

---

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 acf, pacf and 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(s). For secure computations on unordered panel-vectors. Data frame method also allows one-sided formula i.e. ~time.
cols	*data.frame method*: Select columns using a function, column names, indices or a logical vector. *Note*: cols is ignored if a two-sided formula is passed to by.
lag.max	integer. 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 (default) the acf is plotted.
gscale	logical. Do a groupwise scaling / standardization of x,y (using fscale and the groups supplied to g) before computing panel-autocovariances / correlations. See Details.
...	further arguments to be passed to plot.acf.

Details

If gscale = TRUE data are standardized within each group (using 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 flag and then correlating this matrix with the series (x,y) using 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 flag and thus calling flag and cor one time is much faster than calling them lag.max times. The partial ACF is computed from the ACF using a Yule-Walker decomposition, in the same way as in pacf.

Value

An object of class 'acf', see acf. The result is returned invisibly if plot = TRUE.

Note

For plm::pseries and plm::pdata.frame, the first index variable is assumed 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' will not 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 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, grid = FALSE, ...)
```

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(s) 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 for the output matrix.
cols	*data.frame method*: Select columns using a function, column names, indices or a logical vector. *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). FALSE returns 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	either TRUE to automatically colour by panel-groups using rainbow or a character vector of colours matching the number of panel-groups (series).
labs	character. Provide a character-vector of variable labels / series titles when plotting an array.
grid	logical. Calls grid to draw gridlines on the plot.

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.

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),2) # 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
head(psmat(PCGDP), 2)                                    # 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, Weighted) Correlations, Covariances and Observation Counts*

Description

Computes (pairwise, weighted) 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. pwcor and pwcov offer an elaborate print method.

Notes: weights::wtd.cors is imported for weighted pairwise correlations (written in C for speed). For weighted correlations with bootstrap SE's see weights::wtd.cor (but bootstrap can be slow). Weighted correlations for complex surveys are implemented in jtools::svycor. An equivalent and faster implementation of pwcor (without weights) is provided in Hmisc::rcorr (written in Fortran)

Usage

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

pwcov(X, ..., w = NULL, 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, return = FALSE, ...)

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

Arguments

X	a matrix or data.frame, for pwcor and pwcov all columns must be numeric. All functions are faster on matrices, so converting is advised for large data (see qM).
x	an object of class 'pwcor' / 'pwcov'.
w	numeric. A vector of (frequency) weights.
N	logical. TRUE also computes pairwise observation counts.
P	logical. TRUE also computes pairwise p-values (same as cor.test and Hmisc::rcorr).
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. If use != "pairwise.complete.obs", sum(complete.cases(X)) is used for N, and p-values are computed accordingly.
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.
return	logical. TRUE returns the formatted object from the print method for exporting. The default is to return x invisibly.
...	other arguments passed to cor or cov. Only sensible if P = FALSE.

Value

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

See Also

qsu, Summary Statistics, 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-qG-finteraction *Fast Factor Generation, Interactions and Vector Grouping*

Description

qF, shorthand for 'quick-factor' implements very fast 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'. A multivariate version is provided by the function group.

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, drop = FALSE,
   keep.attr = TRUE, method = "auto")

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

is_qG(x)

as_factor_qG(x, ordered = FALSE, na.exclude = TRUE)

finteraction(..., ordered = FALSE, sort = TRUE, method = "auto")
```

Arguments

x	a atomic vector, factor or quick-group.
ordered	logical. Adds a class 'ordered'.
na.exclude	logical. TRUE preserves missing values (i.e. no level is generated for NA).
sort	logical. TRUE sorts the levels in ascending order (like factor); FALSE provides the levels in order of first appearance, which can be significantly faster. Note that if a factor is passed, only sort = FALSE takes effect (as factors usually have sorted levels and checking sortedness can be expensive).
drop	logical. If x is a factor, TRUE efficiently drops unused factor levels beforehand using fdroplevels.
keep.attr	logical. If TRUE and x has additional attributes apart from 'levels' and 'class', these are preserved in the conversion to factor.
method	an integer or character string specifying the method of computation:

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

Note that for finteraction, method = "hash" is always unsorted.

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 an 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. Therefore factors used in grouped operations should preferably be generated using `qF(x,na.exclude = FALSE)`. Setting `sort = FALSE` gathers the levels in first-appearance order (unless `method = "radix"` and `x` is numeric, in which case the levels are always sorted). This can provide a speed improvement, particularly for character data.

There are 3 methods of computation: radix ordering, index hashing, and hashing based on `group`. 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. If `sort = FALSE`, `group` is used which is also very fast. 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 slightly 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`). Since v1.7, you can also call a C-level function `group` directly, which works for multivariate data as well, but does not sort the data and does not preserve missing values.

`finteraction` is simply a wrapper around `as_factor_GRP(GRP.default(X))`, where X is replaced by the arguments in '...' combined in a list (so it's not really an interaction function but just a multivariate grouping converted to factor, 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` and `finteraction` return 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` reorder groups / factor levels. An exception was added in v1.7, when calling `qF(f,sort = FALSE)` on a factor `f`, the levels are recast in first appearance order. These objects can however be converted into one another using `qF/qG` or the direct method `as_factor_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

group, groupid, GRP, Fast Grouping and Ordering, 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)
head(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

```
qsu(x, ...)

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

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

## S3 method for class 'data.frame'
qsu(x, by = NULL, pid = NULL, w = NULL, cols = NULL, higher = FALSE,
    array = TRUE, vlabels = FALSE, stable.algo = TRUE, ...)
```

```
# Methods for compatibility with plm:

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

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

# Methods for compatibility with sf:

## S3 method for class 'sf'
qsu(x, by = NULL, pid = NULL, w = NULL, cols = NULL, higher = FALSE,
    array = TRUE, vlabels = FALSE, stable.algo = TRUE, ...)


## S3 method for class 'qsu'
print(x, digits = 4, nonsci.digits = 9, na.print = "-",
      return = FALSE, print.gap = 2, ...)
```

Arguments

x	a 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 if g/by is also used). 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, a logical vector or a function (e.g. 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.

stable.algo	logical. FALSE uses a faster but less stable method to calculate the standard deviation (see Details of fsd). Only available if w = NULL and higher = FALSE.
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 takes the first variable in the plm::index, 2L the second etc.. Index variables can also be called by name using a character string. 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 of 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(s) '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 4 decimal places. Larger numbers are printed in scientific format. for numbers between 7 and 9 digits, an apostrophe (') is placed after the 6th digit to designate the millions. Missing values are printed using '-'.

The *sf* method simply ignores the geometry column.

Value

A vector, matrix, array or list of matrices of summary statistics. All matrices and arrays have a class 'qsu' and a class 'table' attached.

Note

In weighted summaries, observations with missing or zero weights are skipped, and thus do not affect any of the calculated statistics, including the observation count. This also implies that a logical vector passed to `w` can be used to efficiently summarize a subset of the data.

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.

References

Welford, B. P. (1962). Note on a method for calculating corrected sums of squares and products. *Technometrics*. 4 (3): 419-420. doi:10.2307/1266577.

See Also

`descr`, Summary Statistics, 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
```

---

quick-conversion *Quick Data Conversion*

Description

Fast, flexible and precise conversion of common data objects, without method dispatch and extensive checks:

- qDF, qDT and qTBL convert vectors, matrices, higher-dimensional arrays and suitable lists to data frame, *data.table* and *tibble*, 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 data frame / list, to numeric or character (by converting the levels).

Usage

```
# Converting between matrices, data frames / tables / tibbles

 qDF(X, row.names.col = FALSE, keep.attr = FALSE, class = "data.frame")
 qDT(X, row.names.col = FALSE, keep.attr = FALSE, class = c("data.table", "data.frame"))
qTBL(X, row.names.col = FALSE, keep.attr = FALSE, class = c("tbl_df","tbl","data.frame"))
  qM(X,                        keep.attr = FALSE, class = NULL)

# Programmer functions: matrix rows or columns to list / DF / DT - fully in C++

mctl(X, names = FALSE, return = "list")
mrtl(X, names = FALSE, return = "list")

# Converting factors or factor columns

  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 accept matrices, as_numeric_factor and as_character_factor only accept factors, data frames or lists.
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".
keep.attr	logical. FALSE (default) yields a *hard* / *thorough* object conversion: All unnecessary attributes are removed from the object yielding a plain matrix / data.frame / *data.table*. FALSE yields a *soft* / *minimal* object conversion: Only the attributes 'names', 'row.names', 'dim', 'dimnames' and 'levels' are modified in the conversion. Other attributes are preserved. See also class.

`class` if a vector of classes is passed here, the converted object will be assigned these classes. If `NULL` is passed, the default classes are assigned: `qM` assigns no class, `qDF` a class `"data.frame"`, and `qDT` a class `c("data.table","data.frame")`. If `keep.attr = TRUE` and `class = NULL` and the object already inherits the default classes, further inherited classes are preserved. See Details and the Example.

`names` logical. Should the list be named using row/column names from the matrix?

`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 plain data.frame
3	"data.table"	returns a plain *data.table*

Details

Object conversions using these functions are maximally efficient and involve 3 consecutive steps: (1) Converting the storage mode / dimensions / data of the object, (2) converting / modifying the attributes and (3) modifying the class of the object:

(1) is determined by the choice of function and the optional `row.names.col` argument to `qDF` and `qDT`. Higher-dimensional arrays are converted by expanding the second dimension (adding columns, same as `as.matrix,as.data.frame,as.data.table`).

(2) is determined by the `keep.attr` argument: `keep.attr = TRUE` seeks to preserve the attributes of the object. It's effect is like copying `attributes(converted) <-attributes(original)`, and then modifying the `"dim","dimnames","names","row.names"` and `"levels"` attributes as necessitated by the conversion task. `keep.attr = FALSE` only converts / assigns / removes these attributes and drops all others.

(3) is determined by the `class` argument: Setting `class = "myclass"` will yield a converted object of class `"myclass"`, with any other / prior classes being removed by this replacement. Setting `class = NULL` does NOT mean that a class `NULL` is assigned (which would remove the class attribute), but rather that the default classes are assigned: `qM` assigns no class, `qDF` a class `"data.frame"`, and `qDT` a class `c("data.table","data.frame")`. At this point there is an interaction with `keep.attr`: If `keep.attr = TRUE` and `class = NULL` and the object converted already inherits the respective default classes, then any other inherited classes will also be preserved (with `qM(x,keep.attr = TRUE,class = NULL)` any class will be preserved if `is.matrix(x)` evaluated to `TRUE`.)

The default `keep.attr = FALSE` ensures *hard* conversions so that all unnecessary attributes are dropped. Furthermore in `qDF` and `qDT` the default classes were explicitly assigned, thus any other classes (like 'tbl_df', 'tbl', 'pdata.frame', 'sf', 'tsibble' etc.) will be removed when these objects are passed, regardless of the `keep.attr` setting. This is to ensure that the default methods for 'data.frame' and 'data.table' can be assumed to work, even if the user chooses to preserve further attributes. For `qM` a more lenient default setup was chosen to enable the full preservation of time series matrices with `keep.attr = TRUE`. If the user wants to keep attributes attached to a matrix but make sure that all default methods work properly, either one of `qM(x,keep.attr = TRUE,class = "matrix")` or `unclass(qM(x,keep.attr = TRUE))` should be employed.

Value

qDF - returns a data.frame
qDT - returns a *data.table*
qTBL - returns a *tibble*
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

qF, Collapse Overview

Examples

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

---

radixorder *Fast Radix-Based Ordering*

Description

A slight modification of 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.

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	logical. for controlling the treatment of NA's. 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. See also group.

Value

An integer ordering vector with attributes: Unless na.last = NA an attribute 'sorted' indicating whether the input data was already sorted is attached. If starts = TRUE, 'starts' giving a vector of group starts in the ordered data, and if group.sizes = TRUE, 'group.sizes' giving the vector of group sizes are attached. In either case an attribute 'maxgrpn' providing the size of the largest group is also attached.

Author(s)

The C code was taken - with slight modifications, from base R source code, and is originally due to *data.table* authors Matt Dowle and Arun Srinivasan.

See Also

Fast Grouping and Ordering, Collapse Overview

Examples

```
radixorder(mtcars$mpg)
head(mtcars[radixorder(mtcars$mpg), ])
radixorder(mtcars$cyl, mtcars$vs)

o <- radixorder(mtcars$cyl, mtcars$vs, starts = TRUE)
st <- attr(o, "starts")
head(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, ..., classes = "data.frame")
```

Arguments

l	a list.
FUN	a function that can be applied to all elements in l.
...	additional elements passed to FUN.
classes	character. These are classes of list-based objects inside l which FUN should be applied to. Note that FUN is also applied to all non-list elements in l. It is thus quite different from the classes argument to rapply.

Value

A list of the same structure as l, where FUN was applied to all final (atomic) elements and list-based objects of a class included in classes.

See Also

rsplit, unlist2d, List Processing, Collapse Overview

Examples

```
l <- list(mtcars, list(mtcars, as.matrix(mtcars)))
rapply2d(l, fmean)
unlist2d(rapply2d(l, fmean))
```

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 (default is `0L`). data can be multi-typed, in which case appropriate columns can be selected through the `cols` argument. For numeric data a more versatile alternative is provided by `data.table::nafill` and `data.table::setnafill`.
- `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 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, set = FALSE)

recode_char(X, ..., default = NULL, missing = NULL, regex = FALSE,
            ignore.case = FALSE, fixed = FALSE, set = FALSE)

replace_NA(X, value = 0L, cols = NULL, set = FALSE)

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!

`set` logical. `TRUE` does (some) replacements by reference (i.e. in-place modification of the data). For `replace_NA` this feature is mature, and the result will be returned invisibly. For `recode_num` and `recode_char`, replacement by reference is still partial, so you need to assign the result to an object to materialize all changes.

`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.

`cols` select columns to replace missing values in using a function, column names, indices or logical vector.

`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 on standardized data. 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).

`ignore.case, fixed`
logical. Passed to `grepl` and only applicable if `regex = TRUE`.

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.

Simple replacing tasks on a vector can also effectively be handled by, `setv` / `copyv`. Fast vectorized switches are offered by package *kit* (functions `iif`, `nif`, `vswitch`, `nswitch`) as well as `data.table::fcase` and `data.table::fifelse`.

See Also

pad, Efficient Programming, 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
```

---

roworder *Fast Reordering of Data Frame Rows*

Description

A fast substitute for `dplyr::arrange`. It returns a sorted copy of the data frame, unless the data is already sorted in which case no copy is made. In addition, rows can be manually re-ordered. Use `data.table::setorder` to sort a data frame without creating a copy.

Usage

```
roworder(X, ..., na.last = TRUE)

roworderv(X, cols = NULL, neworder = NULL, decreasing = FALSE,
          na.last = TRUE, pos = "front")
```

Arguments

`X`	a data frame or list of equal-length columns.
`...`	comma-separated columns of `X` to sort by e.g. `var1,var2`. Negatives i.e. `-var1,var2` can be used to sort in decreasing order of `var1`.
`cols`	select columns to sort by using a function, column names, indices or a logical vector. The default `NULL` sorts by all columns in order of occurrence (from left to right).
`na.last`	logical. If `TRUE`, missing values in the sorting columns are placed last; if `FALSE`, they are placed first; if `NA` they are removed (argument passed to `radixorderv`).

decreasing	logical. Should the sort order be increasing or decreasing? Can also be a vector of length equal to the number of arguments in cols (argument passed to radixorderv).
neworder	an ordering vector, can be < nrow(X). if pos = "front" or pos = "end", a logical vector can also be supplied. This argument overwrites cols.
pos	integer or character. Different arrangement options if !is.null(neworder) && length(neworder) < nrow(X).

Int.	*String*	*Description*
1	"front"	move rows in neworder to the front (top) of X (the default).
2	"end"	move rows in neworder to the end (bottom) of X.
3	"exchange"	just exchange the order of rows in neworder, other rows remain in the same position.
4	"after"	place all further selected rows behind the first selected row.

Value

A copy of X with rows reordered. If X is already sorted, X is simply returned.

Note

If you don't require a copy of the data, use data.table::setorder (you can also use it in a piped call as it invisibly returns the data).

See Also

colorder, fsubset, Fast Grouping and Ordering, Collapse Overview.

Examples

```
head(roworder(airquality, Month, -Ozone))
head(roworder(airquality, Month, -Ozone, na.last = NA)) # Removes the missing values in Ozone

## Same in standard evaluation
head(roworderv(airquality, c("Month", "Ozone"), decreasing = c(FALSE, TRUE)))
head(roworderv(airquality, c("Month", "Ozone"), decreasing = c(FALSE, TRUE), na.last = NA))

## Custom reordering
head(roworderv(mtcars, neworder = 3:4))               # Bring rows 3 and 4 to the front
head(roworderv(mtcars, neworder = 3:4, pos = "end"))  # Bring them to the end
head(roworderv(mtcars, neworder = mtcars$vs == 1))    # Bring rows with vs == 1 to the top
```

---

rsplit *Recursive Splitting*

Description

rsplit recursively splits a vector or data frame into subsets according to combinations of (multiple) vectors / factors - by default returning a (nested) list. If flatten = TRUE, the list is flattened yielding the same result as split. rsplit is implemented as a wrapper around gsplit, and faster than split.

Usage

```
rsplit(x, ...)

## Default S3 method:
rsplit(x, fl, drop = TRUE, flatten = FALSE, use.names = TRUE, ...)

## S3 method for class 'data.frame'
rsplit(x, by, drop = TRUE, flatten = FALSE, cols = NULL,
       keep.by = FALSE, simplify = TRUE, use.names = TRUE, ...)
```

Arguments

x	a vector, data.frame or list.
fl	a vector / factor, GRP object, or list of vectors / factors used to split x.
by	*data.frame method*: Same as fl, but also allows one- or two-sided formulas i.e. ~ group1 or var1 + var2 ~ group1 + group2. See Examples.
drop	logical. TRUE removes unused levels or combinations of levels from factors before splitting; FALSE retains those combinations yielding empty list elements in the output.
flatten	logical. If fl is a list of vectors / factors, TRUE calls GRP on the list, creating a single grouping used for splitting; FALSE yields recursive splitting.
use.names	logical. TRUE returns a named list (like split); FALSE returns a plain list.
cols	*data.frame method*: Select columns to split using a function, column names, indices or a logical vector. *Note*: cols is ignored if a two-sided formula is passed to by.
keep.by	logical. If a formula is passed to by, then TRUE preserves the splitting (right-hand-side) variables in the data frame.
simplify	*data.frame method*: Logical. TRUE calls rsplit.default if a single column is split e.g. rsplit(data,col1 ~ group1) becomes the same as rsplit(data$col1,data$group1).
...	further arguments passed to GRP. Sensible choices would be sort = FALSE, decreasing = TRUE or na.last = FALSE. Note that these options only apply if fl is not already a factor.

Value

a (nested) list containing the subsets of x.

See Also

gsplit, rapply2d, unlist2d, List Processing, Collapse Overview

Examples

```
rsplit(mtcars$mpg, mtcars$cyl)
rsplit(mtcars, mtcars$cyl)

rsplit(mtcars, mtcars[.c(cyl, vs, am)])
rsplit(mtcars, ~ cyl + vs + am, keep.by = TRUE)  # Same thing
rsplit(mtcars, ~ cyl + vs + am)

rsplit(mtcars, ~ cyl + vs + am, flatten = TRUE)

rsplit(mtcars, mpg ~ cyl)
rsplit(mtcars, mpg ~ cyl, simplify = FALSE)
rsplit(mtcars, mpg + hp ~ cyl + vs + am)
rsplit(mtcars, mpg + hp ~ cyl + vs + am, keep.by = TRUE)
```

---

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 downwards.
`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` lets `seqid` track 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 as a workaround to deal with problems of computing lagged values, differences and growth rates on irregularly spaced time series and panels before *collapse* version 1.5.0 (#26). Now flag, fdiff and fgrowth natively support irregular data so this workaround is superfluous, except for iterated differencing which is not yet supported with irregular data.

The theory of the workaround was to express an irregular time series or panel series as a regular panel series with a group-id created such that the time-periods within each group are consecutive. seqid makes this very easy: For an irregular panel with some 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.

In general, for any regularly spaced panel the identity given by identical(groupid(id,order(id,time)),seqid(time,or should hold.

Regularly spaced panels with gaps in time (such as a panel-survey with measurements every 2 years) 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 Grouping and Ordering, 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

## This gave a gaps in time error previous to collapse 1.5.0
L(data, 1, value ~ id, ~time)

## 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 this way
L(data, 1, value ~ newid, ~time)

## A different possibility: Creating a consecutive time variable
settransform(data, newtime = data.table::rowid(id))
data
L(data, 1, value ~ id, ~newtime)

## With sorted data, the time variable can also just be omitted..
L(data, 1, value ~ id)
```

---

small-helpers *Small (Helper) Functions*

Description

Convenience functions in the *collapse* package that help to deal with object attributes such as variable names and labels, matching and object checking, and that improve the workflow.

Usage

```
.c(...)                 # Non-standard concatenation i.e. .c(a, b) == c("a", "b")
nam %=% values                # Multiple-assignment e.g. .c(x, y) %=% c(1, 2),
massign(nam, values,          # can also assign to different environment.
        envir = parent.frame())
vlabels(X, attrn = "label",  # Get labels of variables in X, in attr(X[[i]], attrn)
        use.names = TRUE)
vlabels(X, attrn = "label") <- value   # Set labels of variables in X (by reference)
setLabels(X, value, attrn = "label",   # Set labels of variables in X (by reference)
          cols = NULL)                    # and return X
vclasses(X, use.names = TRUE) # Get classes of variables in X
namlab(X, class = FALSE,      # Return data frame of names and labels,
  attrn = "label", N = FALSE, # and (optionally) classes, number of observations
  Ndistinct = FALSE)          # and number of non-missing distinct values
add_stub(X, stub, pre = TRUE, # Add a stub (i.e. prefix or postfix) to column names
         cols = NULL)
rm_stub(X, stub, pre = TRUE,  # Remove stub from column names, also supports general
        regex = FALSE,        # regex matching and removing of characters
        cols = NULL, ...)
x %!in% table                 # The opposite of %in%
ckmatch(x, table,         # Check-match: throws an informative error if non-matched
     e = "Unknown columns:")
all_identical(...)        # Check exact equality of multiple objects or list-elements
all_obj_equal(...)        # Check near equality of multiple objects or list-elements
```

```
setRownames(object,
 nm = if(is.atomic(object))   # Set rownames of object and return object
 seq_row(object) else NULL)
setColnames(object, nm)       # Set colnames of object and return object
setDimnames(object, dn,
            which = NULL)     # Set dimension names of object and return object
unattrib(object)              # Remove all attributes from object
setAttrib(object, a)        # Replace all attributes with list of attributes 'a'
copyAttrib(to, from)       # Copy all attributes from object 'from' to object 'to'
copyMostAttrib(to, from)    # Copy most attributes from object 'from' to object 'to'
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).
x, table	a (atomic) vector.
object, to, from	a suitable R object.
a	a suitable list of attributes.
attrn	character. Name of attribute to store labels or retrieve labels from.
N, Ndistinct	logical. Options to display the number of observations or number of distinct non-missing values.
value	for whichv and alloc: a single value of any vector type. For vlabels<- and setLabels: a matching character vector or list of variable labels.
use.names	logical. Preserve names if X is a list.
cols	integer. (optional) indices of columns to apply the operation to. Note that for these small functions this needs to be integer, whereas for other functions in the package this argument is more flexible.
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.
regex	logical. Match pattern anywhere in names using a regular expression and remove it with gsub.
nm	a suitable vector of row- or column-names.
dn	a suitable vector or list of names for dimension(s).
which	integer. If NULL, dn has to be a list fully specifying the dimension names of the object. Alternatively, a vector or list of names for dimensions which can be supplied. See Examples.
e	the error message thrown by ckmatch for non-matched elements. The message is followed by the comma-separated non-matched elements.

`nam`	character. A vector of object names.
`values`	a matching atomic vector or list of objects.
`envir`	the environment to assign into.
`...`	for `.c`: Comma-separated expressions. 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 / numeric equality. For `rm_stub`: further arguments passed to `gsub`.

See Also

Efficient Programming, Collapse Overview

Examples

```
## Non-standard concatenation
.c(a, b, "c d", e == f)

## Multiple assignment
.c(a, b) %=% list(1, 2)
.c(T, N) %=% dim(EuStockMarkets)
names(iris) %=% iris
list2env(iris)          # Same thing
rm(list = c("a", "b", "T", "N", names(iris)))

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

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

## Setting dimension names of an object
head(setRownames(mtcars))
ar <- array(1:9, c(3,3,3))
setRownames(ar)
setColnames(ar, c("a","b","c"))
setDimnames(ar, c("a","b","c"), which = 3)
setDimnames(ar, list(c("d","e","f"), c("a","b","c")), which = 2:3)
setDimnames(ar, list(c("g","h","i"), c("d","e","f"), c("a","b","c")))

## 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)
```

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 (weighted) pairwise correlations, covariances and observation counts on matrices and data frames. 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 frames or lists of vectors	Detailed statistical
pwcor	No methods, for matrices or data frames	Pairwise correlatio
pwcov	No methods, for matrices or data frames	Pairwise covarianc
pwnobs	No methods, for matrices or data frames	Pairwise observati
varying	default, matrix, data.frame, pseries, pdata.frame, grouped_df	Fast variation chec

See Also

Collapse Overview, Fast Statistical Functions

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 regular / balanced or irregular / unbalanced time series and panel data.
- Similarly, `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**. `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.
- `fcumsum` is an S3 generic to efficiently compute cumulative sums on time series and panel data. In contrast to `cumsum`, it can handle missing values and supports both grouped and ordered computations.
- `psmat` is an S3 generic to efficiently convert panel-vectors or `plm::pseries` and data frames 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 frames 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
`fcumsum`	`default, matrix, data.frame, pseries, pdata.frame, grouped_df`	Compute cumulative sums
`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

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 data frame (class 'grouped_df').
STATS	a matching set of summary statistics. See Details and Examples.
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 in x
2	"replace"	replace but preserve missing values in x
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. For mean-preserving scaling see also fscale)
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 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 Details.
keep.group_vars	*grouped_df method:* Logical. FALSE removes grouping variables after computation. 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 sweep, albeit 4-times more efficient on matrices and many times more efficient on data frames. In this case all methods support an atomic vector of statistics of length NCOL(x) passed to STATS. The matrix and

data frame methods also support a 1-row matrix or 1-row data frame / list, respectively. TRA always preserves all attributes of x.

With groups passed 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) equals the number of groups (i.e. the number of levels if g is a factor)]. If this condition is satisfied, TRA will assume that the first row of STATS is the set of statistics computed on the first group/level of g, the second row on the second group/level 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,STATS,"-",g) = c(-3.75,1.05,-1.05,2.90,3.75,
[same as fmean(x,g,TRA = "-")] does the equivalent of the following for-loop: for(i in 1:6) out[i] = x[i] -STATS[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 factors (see qF) and lists will be converted to 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 functions from other packages internally group atomic vectors or lists in a non-sorted way. [*Note*: as.factor conversions are ok as this also involves sorting.]

If x is a grouped data frame ('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 return 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 fungroup() or dplyr::ungroup()).

Value

x with columns replaced or swept out using STATS, (optionally) grouped by g.

Note

In most cases there is no need to call the TRA() function, because of the TRA-argument to all Fast Statistical Functions (ensuring that the exact same grouping vector is used for computing statistics and subsequent transformation). In addition the functions fbetween/B and fwithin/W and fscale/STD provide optimized solutions 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)

iris %>%  fgroup_by(Species) %>% TRA(fmean(.))  # Faster collapse grouping...
```

---

t_list *Efficient List Transpose*

Description

`t_list` turns a list of lists inside-out. The performance is quite efficient regardless of the size of the list.

Usage

```
t_list(l)
```

Arguments

`l`	a list of lists. Elements inside the sublists can be heterogeneous, including further lists.

Value

`l` transposed such that the second layer of the list becomes the top layer and the top layer the second layer. See Examples.

See Also

rsplit, List Processing, Collapse Overview

Examples

```
# Homogenous list of lists
l <- list(a = list(c = 1, d = 2), b = list(c = 3, d = 4))
str(l)
str(t_list(l))

# Heterogenous case
l2 <- list(a = list(c = 1, d = letters), b = list(c = 3:10, d = list(4, e = 5)))
attr(l2, "bla") <- "abc"  # Attributes other than names are preserved
str(l2)
str(t_list(l2))

rm(l, l2)
```

---

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 through recursive flattening and intelligent row-binding operations. It is a full 2-dimensional generalization of unlist, but best understood as a recursive generalization of do.call(rbind,...). This function is a powerful tool to create a tidy data frame representation from (nested) lists of vectors, data frames, matrices, arrays or heterogeneous objects.

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. For plain matrices in l, integer row names are generated.

recursive	logical. if FALSE, only process the lowest (deepest) level of l.
id.factor	if TRUE and idcols != FALSE, create id columns as factors instead of character or integer vectors. Alternatively it is possible to specify id.factor = "ordered" to generate ordered factor id's. This is useful if id's are used for further analysis e.g. as inputs to ggplot2.
DT	logical. TRUE returns 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.
- Identify 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 frames 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 id-columns on the left, filled with the object names or indices (if the (sub-)list is unnamed). If row.names != FALSE, 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 != FALSE), 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 efficiently binds into 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

rsplit, rapply2d, List Processing, Collapse Overview

Examples

```
## Basic Examples:
l <- list(mtcars, list(mtcars, mtcars))
tail(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)))
tail(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, give.attr = FALSE)                 # 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 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"), 2)
```

---

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 (e.g. 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 grouped data frame / compatibility with dplyr:

## S3 method for class 'grouped_df'
varying(x, any_group = TRUE, use.g.names = FALSE, drop = TRUE,
        keep.group_vars = TRUE, ...)

# Methods for grouped data frame / compatibility with sf:

## S3 method for class 'sf'
varying(x, by = NULL, cols = NULL, any_group = TRUE, use.g.names = TRUE, drop = TRUE, ...)
```

Arguments

x	a vector, matrix, data.frame, panel series (plm::pseries), panel data frame (plm::pdata.frame) or grouped data frame (class 'grouped_df'). Data must not be numeric.
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 (e.g. is.numeric). Two-sided formulas passed to by overwrite cols.
use.g.names	logical. Make group-names and add to the result as names (default method) or row-names (matrix and data frame methods). No row-names are generated for *data.table*'s.
drop	*matrix and data.frame methods:* Logical. TRUE drops dimensions and returns an atomic vector if the result is 1-dimensional.
effect	*plm* methods: Select the panel identifier by which variation in the data should be examined. 1L takes the first variable in the plm::index, 2L the second etc.. Index variables can also be called by name. More than one index variable can be supplied, which will be interacted.
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.

The *sf* method simply ignores the geometry column.

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

Summary Statistics, 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, ~ country)

## Same as above but done for each country individually, countries without data are coded NA
head(varying(wlddev, ~ country, any_group = FALSE))
```

`wlddev` *World Development Dataset*

Description

This dataset contains 5 indicators from the World Bank's World Development Indicators (WDI) database: (1) GDP per capita, (2) Life expectancy at birth, (3) GINI index, (4) Net ODA and official aid received and (5) Population. The panel data is balanced and covers 216 present and historic countries from 1960-2020 (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 13176 observations on the following 13 variables. All variables are labeled e.g. have a 'label' attribute.

`country` *chr* Country Name

`iso3c` *fct* Country Code

`date` *date* Date Recorded (Fictitious)

`year` *int* Year

`decade` *int* 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 official development assistance and official aid received (constant 2018 US$)

`POP` *num* Population, total

Source

https://data.worldbank.org/, accessed via the WDI package. The codes for the series are c("NY.GDP.PCAP.KD","SP.DYN.LE00.IN","SI.POV.GINI","DT.ODA.ALLD.KD","SP.POP.TOTL").

See Also

GGDC10S, Collapse Overview

Examples

```
data(wlddev)

# Panel-summarizing the 5 series
qsu(wlddev, pid = ~iso3c, cols = 9:13, vlabels = TRUE)

# By Region
qsu(wlddev, by = ~region, cols = 9:13, vlabels = TRUE)

# Panel-summary by region
qsu(wlddev, by = ~region, pid = ~iso3c, cols = 9:13, vlabels = TRUE)

# Pairwise correlations: Ovarall
print(pwcor(get_vars(wlddev, 9:13), N = TRUE, P = TRUE), show = "lower.tri")

# Pairwise correlations: Between Countries
print(pwcor(fmean(get_vars(wlddev, 9:13), wlddev$iso3c), N = TRUE, P = TRUE), show = "lower.tri")

# Pairwise correlations: Within Countries
print(pwcor(fwithin(get_vars(wlddev, 9:13), wlddev$iso3c), N = TRUE, P = TRUE), show = "lower.tri")
```

Index