# **collapse**: Advanced and Fast Statistical Computing and Data Transformation in R

Sebastian Krantz

Junior Researcher,

Kiel Institute for the World Economy

22nd June 2022

# Table of Contents

# What is collapse? In One Sentence

A C/C++ based R package that facilitates statistical computations of high complexity, at outstanding levels of performance and programming efficiency, in a way that integrates seamlessly with popular classes and data manipulation frameworks, while fully supporting the efficient infrastructure available in base R.

# What is collapse?

**A C/C++ based infrastructure package for R that provides:**

- A large set of statistical functions and operations that are fully vectorized along both columns and groups, including weighted statistics and statistics for categorical data.
- Enhanced time series and panel data support (indexing, irregularity, advanced transformations, exploration)
- (Recursive) operations on lists of data objects
- Advanced descriptive statistics tools
- Data manipulation, programming and utility functions, including fast routines to group and order data, and to determine unique values.
- Data transformation by reference and OpenMP Multithreading (in progress)

- Class Agnostic Programming
  - S3 generic statistical functions + smart internal attribute handling.
  - Supports base R: vectors, factors, matrices, data frames, lists.
  - Supports 'ts', 'xts/zoo', 'data.table', 'tibble', 'sf', 'pseries/pdata.frame' and preserves many others such as 'tsibble'.
- Extreme Speed and Efficiency
  - C/C++ powered computations that scale near-linearly in data size, regardless of the number of columns or groups (dimensionality).
  - Highly optimized R code: primitive/internal base functions, checks at C/C++ level, no conversions $\rightarrow$ all functions execute in $< 50\mu s$.
  - Internal optimizations e.g. for strings, factors, integer & logical vectors, no NAs, singleton groups, pre-sorted and unique data.
  - Users can access many C/C++ level algorithms, helper functions, and core S3 methods directly in R $\rightarrow$ excellent for programming.
- Stable and Flexible API [SE & NSE, flexible inputs, shorthands]
- Flexible Namespace [global option to mask base R/dplyr functions]
- Designed for Socioeconomic Data [Efficient na.rm = TRUE default + computations on NAs = NA, and support for variable labels]

# Fast Statistical Functions

collapse provides a set of enhanced, S3 generic, statistical functions that offer much greater performance and flexibility than base R.

```
library(collapse)
.FAST_STAT_FUN # Global macro containing function names
##  [1] "fmean"      "fmedian"    "fmode"      "fsum"       "fprod"
##  [6] "fsd"        "fvar"       "fmin"       "fmax"       "fnth"
## [11] "ffirst"     "flast"      "fnobs"      "fndistinct"
#
methods(fmean) # Methods available for all .FAST_STAT_FUN
## [1] fmean.data.frame  fmean.default     fmean.grouped_df* fmean.list*
## [5] fmean.matrix
## see '?methods' for accessing help and source code
#
# Basic usage:
fmean(AirPassengers)  # Vector
## [1] 280.2986
fmean(EuStockMarkets) # Matrix
##      DAX      SMI      CAC     FTSE
## 2530.657 3376.224 2227.828 3565.643
fmean(airquality)     # Data Frame
##      Ozone    Solar.R       Wind       Temp      Month        Day
##  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

# Fast Statistical Functions

collapse provides a set of enhanced, S3 generic, statistical functions that offer much greater performance and flexibility than base R.

```
library(microbenchmark)
x <- rnorm(1e7)
microbenchmark(mean(x), fmean(x), fmean(x, nthreads = 4))
## Unit: milliseconds
##                    expr       min        lq      mean    median        uq      max neval
##                 mean(x) 19.483487 19.963105 20.174734 20.103899 20.300740 21.57252   100
##                fmean(x)  9.645988  9.950188 10.197586 10.073515 10.213592 16.55400   100
##  fmean(x, nthreads = 4)  2.657497  3.438363  4.127265  3.683522  4.301618 16.84149   100
#
microbenchmark(colMeans(EuStockMarkets), fmean(EuStockMarkets))
## Unit: microseconds
##                      expr   min    lq     mean median     uq      max neval
##  colMeans(EuStockMarkets) 9.676 9.881 10.80227 10.004 10.4960   61.746   100
##     fmean(EuStockMarkets) 8.569 8.651 28.48311  8.774  9.3685 1948.894   100
#
microbenchmark(sapply(mtcars, mean), fmean(mtcars))
## Unit: microseconds
##                  expr    min     lq     mean  median     uq     max neval
##  sapply(mtcars, mean) 19.926 20.336 24.19164 20.6845 22.468 288.148   100
##         fmean(mtcars)  1.599  1.722  3.72772  1.8450  1.968 180.687   100
#
# Slightly larger data (5000 rows, 11 columns, with ~10% missing values)
microbenchmark(base = sapply(GGDC10S[6:16], mean, na.rm = TRUE), fmean(GGDC10S[6:16]))
## Unit: microseconds
##                  expr     min       lq      mean   median       uq      max neval
##                  base 271.953 329.9885 469.31511 347.3315 371.3370 5695.105   100
##  fmean(GGDC10S[6:16])  54.366  54.8375  58.80097  55.3910  57.9125  133.373   100
```

# Fast Statistical Functions

**Syntax:**

```
FUN(x, g = NULL, [w = NULL,] TRA = NULL, [na.rm = TRUE,]
   use.g.names = TRUE, [drop = TRUE,] [nthreads = 1,] ...)
```

| *Argument* | *Description* |
| --- | --- |
| `g` | grouping vectors / lists of vectors or 'GRP' object |
| `w` | a vector of (frequency) weights |
| `TRA` | a quoted operation to transform `x` using the statistics |
| `na.rm` | efficiently skips missing values in `x` |
| `use.g.names` | generate names / row-names from `g` |
| `drop` | drop dimensions if `g = TRA = NULL` |
| `nthreads` | number of threads for OpenMP multithreading |
| `...` | optional argument `set = TRUE`, which toggles data transformation by reference if `!is.null(TRA)`, or `keep.group_vars` and `keep.w` for the grouped_df method |

```r
# Weighted Mean
w <- abs(rnorm(nrow(iris)))
all.equal(fmean(num_vars(iris), w = w), sapply(num_vars(iris), weighted.mean, w = w))
## [1] TRUE
# Missing weights are treated like 0-weights: observation omitted
wNA <- na_insert(w, prop = 0.05)
fmean(num_vars(iris), w = wNA) # weighted.mean() does not support missing weights
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.731918     3.058668     3.575994     1.101117
# Grouped Mean
fmean(iris$Sepal.Length, iris$Species)
##     setosa versicolor  virginica
##      5.006      5.936      6.588
microbenchmark(fmean = fmean(iris$Sepal.Length, iris$Species),
               tapply = tapply(iris$Sepal.Length, iris$Species, mean))
## Unit: microseconds
##    expr    min      lq     mean  median     uq     max neval
##   fmean  3.157  3.4440  3.70025  3.6080  3.731  14.104   100
##  tapply 17.917 18.4705 20.07196 18.7165 19.065 123.615   100
fmean(num_vars(iris), iris$Species)   # by default added as rownames (also for matrices)
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa            5.006       3.428        1.462       0.246
## versicolor        5.936       2.770        4.260       1.326
## virginica         6.588       2.974        5.552       2.026
iris |> fgroup_by(Species) |> fmean() # Using grouped_df method
##      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa        5.006       3.428        1.462       0.246
## 2 versicolor        5.936       2.770        4.260       1.326
## 3  virginica        6.588       2.974        5.552       2.026
iris |> dplyr::group_by(Species) |> fmean() # Also works, internally converts grouping object
## # A tibble: 3 x 5
##   Species    Sepal.Length Sepal.Width Petal.Length Petal.Width
##   <fct>             <dbl>       <dbl>        <dbl>       <dbl>
## 1 setosa             5.01        3.43         1.46       0.246
## 2 versicolor         5.94        2.77         4.26       1.33
```

```
# Weighted Group Mean
fmean(num_vars(iris), iris$Species, w)
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa         4.932130    3.378306     1.479969   0.2246907
## versicolor     5.893926    2.764235     4.213325   1.2809271
## virginica      6.648945    3.009451     5.647726   2.0461023
iris |> add_vars(w) |> fgroup_by(Species) |> fmean(w) # use keep.w = FALSE to omit sum.w
##      Species    sum.w Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa 46.07939     4.932130    3.378306     1.479969   0.2246907
## 2 versicolor 37.53640     5.893926    2.764235     4.213325   1.2809271
## 3  virginica 37.88109     6.648945    3.009451     5.647726   2.0461023
# Grouping and/or weighting has little overhead
microbenchmark(base = lapply(num_vars(iris), weighted.mean, w),
               clp_w = fmean(num_vars(iris), w = w),
               clp_g = fmean(num_vars(iris), iris$Species),
               clp_g_w = fmean(num_vars(iris), iris$Species, w))
## Unit: microseconds
##     expr    min      lq     mean median      uq    max neval
##     base 13.366 13.9605 16.59434 14.965 15.6825 94.259   100
##    clp_w  2.911  3.1160  3.30132  3.280  3.4030  4.879   100
##    clp_g  6.232  6.4370  6.88226  6.601  6.8470 26.240   100
##  clp_g_w  6.273  6.4780  6.79411  6.642  6.8470 12.423   100
#
# Here a benchmark with 10M obs. averaged over 1M groups
library(data.table) # Serial CRAN Binary for M1 MAC
g <- sample.int(1e6, 1e7, replace = TRUE); w <- abs(rnorm(1e7))
dt <- setkey(data.table(x, g), g); g <- GRP(g)
microbenchmark(clp = fmean(x, na.rm = FALSE), clp_g = fmean(x, g, use.g.names = FALSE, na.rm = FALSE),
         clp_g_w = fmean(x, g, w, use.g.names = FALSE, na.rm = FALSE), dt = dt[, mean(x), keyby = g])
## Unit: milliseconds
##     expr        min         lq       mean     median         uq        max
##      clp   9.316061   9.375306   9.450482   9.403309   9.502652   9.763125
##    clp_g  12.668918  13.442506  14.826425  13.882784  16.171425  47.235854
##  clp_g_w  46.979932  53.915656  57.739099  57.505493  59.045514  96.965000
##       dt 134.898528 139.757561 145.082348 141.236000 142.593265 176.611887
```

# Considerations with Missing Data

```
collap(wlddev, GINI ~ country, list(mean, median, min, max, sum, prod),
       na.rm = TRUE, give.names = FALSE) |> head()
##          country     mean median  min  max   sum         prod
## 1    Afghanistan      NaN     NA  Inf -Inf   0.0 1.000000e+00
## 2        Albania 31.41111   31.7 27.0 34.6 282.7 2.902042e+13
## 3        Algeria 34.36667   35.3 27.6 40.2 103.1 3.916606e+04
## 4 American Samoa      NaN     NA  Inf -Inf   0.0 1.000000e+00
## 5        Andorra      NaN     NA  Inf -Inf   0.0 1.000000e+00
## 6         Angola 48.66667   51.3 42.7 52.0 146.0 1.139065e+05
# collapse is very consistent here: computations on NA yield NA
collap(wlddev, GINI ~ country, list(fmean, fmedian, fmin, fmax, fsum, fprod),
       give.names = FALSE) |> head()
##          country    fmean fmedian fmin fmax  fsum        fprod
## 1    Afghanistan       NA      NA   NA   NA    NA           NA
## 2        Albania 31.41111    31.7 27.0 34.6 282.7 2.902042e+13
## 3        Algeria 34.36667    35.3 27.6 40.2 103.1 3.916606e+04
## 4 American Samoa       NA      NA   NA   NA    NA           NA
## 5        Andorra       NA      NA   NA   NA    NA           NA
## 6         Angola 48.66667    51.3 42.7 52.0 146.0 1.139065e+05
```

## Inspired by commercial software like STATA

```
. collapse (mean) mean=GINI (median) median=GINI (min) min=GINI (max) max=GINI (sum) sum=GINI, by(country)
. list in 1/6, separator(10)
```

| | country | mean | median | min | max | sum |
|---|---|---|---|---|---|---|
| 1. | Afghanistan | . | . | . | . | 0 |
| 2. | Albania | 31.411111 | 31.7 | 27 | 34.6 | 282.7 |
| 3. | Algeria | 34.366667 | 35.3 | 27.6 | 40.2 | 103.1 |
| 4. | American Samoa | . | . | . | . | 0 |
| 5. | Andorra | . | . | . | . | 0 |
| 6. | Angola | 48.666667 | 51.3 | 42.7 | 52 | 146 |

# Considerations with Labelled Data

## Base R/dplyr often drops attributes such as labels

```
wlddev |>
  ftransform(pop_units = units::as_units(POP / 1000, "kg")) |>
  fgroup_by(country) |>
  fselect(PCGDP, pop_units) |>
  fmean() |> str()
## 'data.frame': 216 obs. of  3 variables:
##  $ country  : chr  "Afghanistan" "Albania" "Algeria" "American Samoa" ...
##   ..- attr(*, "label")= chr "Country Name"
##  $ PCGDP    : num  484 2819 3532 10071 40083 ...
##   ..- attr(*, "label")= chr "GDP per capita (constant 2010 US$)"
##  $ pop_units: Units: [kg] num  18362.3 2708.3 25305.3 43.1 51.5 ...
##   ..- attr(*, "label")= chr "Population, total"
#
library(dplyr)
wlddev |>
  mutate(pop_units = units::as_units(POP / 1000, "kg")) |>
  group_by(country) |>
  summarise(across(c(PCGDP, pop_units), mean, na.rm = TRUE)) |> str()
## tibble [216 x 3] (S3: tbl_df/tbl/data.frame)
##  $ country  : chr [1:216] "Afghanistan" "Albania" "Algeria" "American Samoa" ...
##   ..- attr(*, "label")= chr "Country Name"
##  $ PCGDP    : num [1:216] 484 2819 3532 10071 40083 ...
##  $ pop_units: Units: [kg] num [1:216] 18362.3 2708.3 25305.3 43.1 51.5 ...
```

## collapse Statistical Functions generally keep attributes[1]

[1] Exceptions are if results don't have the same data type, and aggregations on matrices, and 'ts' objects. Attributes "names", "dim", "dimnames", "row-names" etc. are always adjusted as necessary. When using base R functions in data aggregation commands, collapse also applies these conditions to apply attributes ex-post.

# The TRA() Function

For (grouped) replacing and sweeping out statistics (by reference)

**Syntax:**

```
   TRA(x, STATS, FUN = "-", g = NULL, set = FALSE, ...)
setTRA(x, STATS, FUN = "-", g = NULL, ...)
```

where `STATS` is a vector/matrix/list of statistics to transform `x`.

## Supported Operations (`FUN`)

| *Integer* | *String* | *Description* |
|---|---|---|
| 0 | `"replace_NA"` | replace missing values in `x` |
| 1 | `"replace_fill"` | replace data and missing values in `x` |
| 2 | `"replace"` | replace data but preserve missing values in `x` |
| 3 | `"-"` | subtract (i.e. center) |
| 4 | `"-+"` | center on overall average statistic |
| 5 | `"/"` | divide (i.e. scale) |
| 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. make data divisible by `STATS`) |

# The TRA() Function

TRA() is called internally in the *Fast Statistical Functions*, the `TRA` argument is passed to `FUN`. Thus the following two are equivalent:

```
fmean(x, g, w, "-")  # TRA = "-"
TRA(x, fmean(x, g, w), "-", g)
```

where `x` can be any data object and `g`/`w` can be `NULL`/omitted. Similarly the following two transform data by reference.

```
fmean(x, g, w, "-", set = TRUE)
setTRA(x, fmean(x, g, w), "-", g)
```

```
attach(iris)
all_obj_equal(Sepal.Length - ave(Sepal.Length, Species),
              fmean(Sepal.Length, Species, TRA = "-"),
              TRA(Sepal.Length, fmean(Sepal.Length, Species), "-", Species))
## [1] TRUE
microbenchmark(base = Sepal.Length - ave(Sepal.Length, Species),
               fmean = fmean(Sepal.Length, Species, TRA = "-"),
               fmean_TRA = TRA(Sepal.Length, fmean(Sepal.Length, Species), "-", Species))
## Unit: microseconds
##       expr    min     lq     mean  median      uq    max neval
##       base 23.575 24.518 26.28346 25.1330 26.0555 73.472   100
##      fmean  4.305  4.715  5.08810  4.9200  5.2480  9.430   100
##  fmean_TRA  5.330  5.658  6.05037  5.8835  6.1090 21.443   100
detach(iris)
```

```r
library(magrittr)
num_vars(iris) %<>% na_insert(prop = 0.05)
#
# Missing value imputation by reference using the Species-median
num_vars(iris) |> fmedian(iris$Species, TRA = "replace_NA", set = TRUE)
#
# Different grouped and/or weighted transformations at once
mtcars |> ftransform(A = fsum(mpg, TRA = "%"),
                     B = mpg > fmedian(mpg, cyl, TRA = "replace_fill"),
                     C = fmedian(mpg, list(vs, am), wt, "-"),
                     D = fmean(mpg, vs,, 1L) > fmean(mpg, am,, 1L)) |> head(3)
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb        A     B
## Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4 3.266449  TRUE
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4 3.266449  TRUE
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1 3.546430 FALSE
##                  C     D
## Mazda RX4      1.3 FALSE
## Mazda RX4 Wag  1.3 FALSE
## Datsun 710    -7.6  TRUE
#
# Row and columns-wise Arithmetic operators based on TRA()
mtcars %c/% mtcars |> head(2)
##               mpg cyl disp hp drat wt qsec  vs am gear carb
## Mazda RX4       1   1    1  1    1  1    1 NaN  1    1    1
## Mazda RX4 Wag   1   1    1  1    1  1    1 NaN  1    1    1
#
mtcars %r-% fmedian(mtcars) |> head(2)
##               mpg cyl  disp  hp  drat     wt  qsec vs am gear carb
## Mazda RX4     1.8   0 -36.3 -13 0.205 -0.705 -1.25  0  1    0    2
## Mazda RX4 Wag 1.8   0 -36.3 -13 0.205 -0.450 -0.69  0  1    0    2
```

# Efficient Inputs: Grouping Objects

The g argument can process any vectors / list of vectors defining a grouping. Grouping is relatively expensive, thus collapse exports its grouping algorithms enabling optimization of repeated grouping. The main function to group data, GRP(), returns a list-like object of class 'GRP', which can be passed to g/by arguments.

**Syntax:**

```
GRP(X, by = NULL, sort = TRUE, decreasing = FALSE,
    na.last = TRUE, return.groups = TRUE,
    return.order = sort, method = "auto", ...)
```

```
g <- GRP(iris, by = ~ Species)
print(g)
## collapse grouping object of length 150 with 3 ordered groups
##
## Call: GRP.default(X = iris, by = ~Species), X is sorted
##
## Distribution of group sizes:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      50      50      50      50      50      50
##
## Groups with sizes:
##     setosa versicolor  virginica
##         50         50         50
```

# Efficient Inputs: Grouping Objects

```
# Contains useful information for many kinds of grouped operations
str(g)
## Class 'GRP'  hidden list of 9
##  $ N.groups    : int 3
##  $ group.id    : int [1:150] 1 1 1 1 1 1 1 1 1 1 ...
##  $ group.sizes : int [1:3] 50 50 50
##  $ groups      :'data.frame': 3 obs. of  1 variable:
##   ..$ Species: Factor w/ 3 levels "setosa","versicolor",..: 1 2 3
##  $ group.vars  : chr "Species"
##  $ ordered     : Named logi [1:2] TRUE TRUE
##   ..- attr(*, "names")= chr [1:2] "ordered" "sorted"
##  $ order       : int [1:150] 1 2 3 4 5 6 7 8 9 10 ...
##   ..- attr(*, "starts")= int [1:3] 1 51 101
##   ..- attr(*, "maxgrpn")= int 50
##   ..- attr(*, "sorted")= logi TRUE
##  $ group.starts: int [1:3] 1 51 101
##  $ call        : language GRP.default(X = iris, by = ~Species)
#
# 0-cost input for grouped computations
fmean(num_vars(iris), g)
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa            5.002       3.422        1.464       0.248
## versicolor        5.902       2.768        4.256       1.322
## virginica         6.596       2.974        5.548       2.032
#
# This performs a subset using the group.starts element (extremely fast)
ffirst(num_vars(iris), g, na.rm = FALSE)
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa              5.1         3.5          1.4         0.2
## versicolor          7.0         3.2          4.7         1.4
## virginica           6.3         3.3          6.0         2.5
```

# Example Application: Global Value Chain Analysis

```r
# EORA 2021 Global Supply Chain Database, decomposed using the 'decompr' package.
str(VB$`1990`[c("VB", "O")])  # O is Output, VB is the Value-Added Shares Matrix ~200MB RAM
## List of 2
##  $ VB: num [1:4888, 1:4888] 0.894008 0.000698 0.000618 0.00722 0.00046 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:4888] "AFG.AGR" "AFG.FIS" "AFG.MIN" "AFG.FBE" ...
##   .. ..$ : chr [1:4888] "AFG.AGR" "AFG.FIS" "AFG.MIN" "AFG.FBE" ...
##   ..- attr(*, "long")= logi FALSE
##   ..- attr(*, "k")= chr [1:188] "AFG" "ALB" "DZA" "AND" ...
##   ..- attr(*, "i")= chr [1:26] "AGR" "FIS" "MIN" "FBE" ...
##   ..- attr(*, "decomposition")= chr "leontief"
##   ..- attr(*, "post")= chr "none"
##  $ O : Named num [1:4888] 209890 11125 44005 234718 49507 ...
##   ..- attr(*, "names")= chr [1:4888] "AFG.AGR" "AFG.FIS" "AFG.MIN" "AFG.FBE" ...
# Check Value-Added Shares Matrix
sapply(VB, function(x) fmean(fsum(x$VB, na.rm = FALSE, nthreads = 4))) |> round(3)
## 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005
##    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1
## 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021
##    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1
#
# Grouping defining aggregation to 11 world regions
(rsg <- GRP(paste(rcodes_long, scodes_long, sep = "."), sort = FALSE))
## collapse grouping object of length 4888 with 286 groups
##
## Call: GRP.default(X = paste(rcodes_long, scodes_long, sep = "."), sort = FALSE)
##
## Distribution of group sizes:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    3.00    6.00   10.00   17.09   28.00   50.00
##
## Groups with sizes:
## SAS.AGR SAS.FIS SAS.MIN SAS.FBE SAS.TEX SAS.WAP
##       8       8       8       8       8       8
```

# Example Application: Global Value Chain Analysis

```r
# This aggregates the 32 matrices, 5.7GB RAM.
microbenchmark(call = VB_R <- lapply(VB, function(x) x$VB |>
  fsum(rsg, nthreads = 4, na.rm = FALSE) |> t() |>      # We can sum the rows
  fmean(rsg, x$O, nthreads = 4, na.rm = FALSE) |> t())) # Averaging columns with output weights
## Unit: milliseconds
##  expr      min       lq    mean   median       uq      max neval
##  call 308.8761 313.1945 414.728 321.1538 361.7048 2869.827   100
# -> 44.7M sums, 2.6M weighted means, 64 transpositions and 64 parallel setups in ~0.33s!
# Check
sapply(VB_R, function(x) fmean(fsum(x, na.rm = FALSE, nthreads = 4))) |> round(3)
## 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005
##    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1
## 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021
##    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1
#
# Generating a long-form representation
unlist2d(VB_R, idcols = "Year", row.names = "Source_Region_Sector",
         id.factor = TRUE, DT = TRUE) |>
  data.table::melt(1:2, variable.name = "Using_Region_Sector", value.name = "VA_Share")
##          Year Source_Region_Sector Using_Region_Sector      VA_Share
##       1: 1990              SAS.AGR             SAS.AGR  8.762486e-01
##       2: 1990              SAS.FIS             SAS.AGR  1.966394e-04
##       3: 1990              SAS.MIN             SAS.AGR  7.640502e-03
##       4: 1990              SAS.FBE             SAS.AGR  1.530230e-03
##       5: 1990              SAS.TEX             SAS.AGR  1.162649e-04
##      ---
## 2617468: 2021              ROA.PAD             ROA.REI  1.037213e-01
## 2617469: 2021              ROA.EHO             ROA.REI  2.518079e-01
## 2617470: 2021              ROA.PHH             ROA.REI  5.772590e-02
## 2617471: 2021              ROA.OTH             ROA.REI  1.083316e-01
## 2617472: 2021              ROA.REI             ROA.REI -1.024538e+02
```

# Efficient Inputs: Factors

Apart from 'GRP' objects, *collapse* can also directly use factors in many operations. The `qF()` function efficiently creates factors:

```
x <- na_insert(rnorm(1e7), prop = 0.01) # 10 million obs, 1% missing
g <- sample.int(1e6, 1e7, TRUE)         # 1 million random groups
system.time(gg <- GRP(g))
##    user  system elapsed
##   0.153   0.012   0.165
# Factor generation: same as factor(g, exclude = NULL)
system.time(f <- qF(g, na.exclude = FALSE))
##    user  system elapsed
##   0.113   0.012   0.124
# The "na.included" class signifies that the factor contains no integer missing values
class(f)
## [1] "factor"      "na.included"
# Internal check for factors before C/C++, if fails, a level is added for missing values
collapse:::is.nmfactor
## function (x)
## inherits(x, "factor") && (inherits(x, "na.included") || !anyNA(unclass(x)))
## <bytecode: 0x109b6d8b0>
## <environment: namespace:collapse>
# Testing, f2 does not have the additional class
f2 <- `class<-`(f, "factor")
microbenchmark(fmean(x, g), fmean(x, gg), fmean(x, gg, na.rm = FALSE), fmean(x, f), fmean(x, f2),
               ffirst(x, gg, na.rm = FALSE))
## Unit: milliseconds
##                          expr        min         lq      mean     median
##                   fmean(x, g) 146.060983 150.493309 155.02585 152.197822
##                  fmean(x, gg)  25.354564  27.709625  29.48497  29.022157
##   fmean(x, gg, na.rm = FALSE)  13.184534  13.783585  15.61769  14.128067
##                   fmean(x, f)  24.847271  27.503661  29.47271  29.248580
##                  fmean(x, f2)  28.450433  31.447553  33.41934  32.811829
##  ffirst(x, gg, na.rm = FALSE)   3.150153   3.588792   3.79700   3.676347
```

# Efficient Inputs: Quick-Group Objects

Factor creation can be expensive when levels need to be coerced to character. Thus *collapse* introduces a factor-light class 'qG':

```
system.time(qg <- qG(g, na.exclude = FALSE)) # No big gain as integer -> character is efficient
##    user  system elapsed
##   0.126   0.009   0.136
str(qg) # qG() behaves just like qF(), just doesn't create levels
##  'qG' int [1:10000000] 454560 840953 816728 39316 524168 6380 219287 837126 107389 900329 ...
##  - attr(*, "N.groups")= int 999953
attr(g, "N.groups") <- fndistinct(g) # Can also simply turn the vector g into one
class(g) <- c("qG", "na.included")
microbenchmark(fmean(x, qg), fmean(x, g)) # Sorting does not really matter here
## Unit: milliseconds
##          expr      min       lq     mean   median       uq      max neval
##  fmean(x, qg) 27.42310 29.75690 33.39924 31.80352 34.69736 97.40280   100
##   fmean(x, g) 27.27349 30.18986 33.25877 32.10368 34.70744 50.71868   100
```

`qG()` also sorts. Setting `sort = FALSE` in `qF()/qG()/GRP()` calls `group()`, to group (multivariate) data in first-appearance-order:

```
attributes(g) <- NULL       # Notice the performance gain of not sorting.
system.time(qg <- group(g)) # on unsorted data, setting sort = FALSE can give a large speedup
##    user  system elapsed
##   0.085   0.010   0.095
str(qg) # also 'qG', so we can use it for 0-cost grouping as well
##  'qG' int [1:10000000] 1 2 3 4 5 6 7 8 9 10 ...
##  - attr(*, "N.groups")= int 999953
microbenchmark(fmean(x, g), fmean(x, g, TRA = "replace_fill")) # A paradox?
## Unit: milliseconds
##                                expr      min       lq     mean   median       uq
##                         fmean(x, g) 146.6723 150.7583 155.5740 153.8852 159.0192
##  fmean(x, g, TRA = "replace_fill") 114.1253 117.8344 122.0243 120.1109 125.1906
```

# Fast Grouping and Ordering: Summary

*User Facing: General Grouping*

| | |
|---|---|
| `GRP()` | Fast sorted or unsorted grouping of multivariate data, returns detailed object of class 'GRP' |
| `qF()/qG()` | Fast generation of factors and quick-group ('qG') objects from atomic vectors |
| `finteraction()` | Fast interactions: returns factor or 'qG' objects |
| `fdroplevels()` | Efficiently remove unused factor levels |

*Mostly Programmers: Workhorse Functions*

| | |
|---|---|
| `radixorder()` | Fast ordering and ordered grouping |
| `group()` | Fast first-appearance-order grouping: returns 'qG' object |
| `gsplit()` | Split vector based on 'GRP' object |
| `greorder()` | Reorder the results |

*Mostly Programmers: Specialized Functions that also return 'qG' objects*

| | |
|---|---|
| `groupid()` | Generalized run-length-type grouping |
| `seqid()` | Grouping of integer sequences |
| `timeid()` | Grouping of time sequences (based on GCD) |

# Further Data Transformation Functions

*Split-Apply-Combine Computing with Arbitrary R functions*

| | |
|---|---|
| `dapply()` | Apply a function to rows or columns of data.frame or matrix based objects. |
| `BY()` | Apply a function to vectors or matrix/data frame columns by groups. |

*Specialized Data Transformation Functions*

| | |
|---|---|
| `fbetween()`, `fwithin()` | Fast averaging and (quasi-)centering. |
| `fhdbetween()`, `fhdwithin()` | Higher-Dimensional averaging/centering and linear prediction/partialling out (powered by *fixest*'s algorithm for multiple factors). |
| `fscale()` | (advanced) scaling and centering. |

*Time / Panel Series Functions*

| | |
|---|---|
| `fcumsum()` | Cumulative sums |
| `flag()` | Lags and leads |
| `fdiff()` | (Quasi-, Log-, Iterated-) differences |
| `fgrowth()` | (Compounded-) growth rates |

# Data Manipulation Functions

Mostly performance improved versions of base R and dplyr functions, but with some minor differences / improvements.

`fselect()`, `fsubset()`, `[f/set]transform[v]()`, `fgroup_by()`, `fmutate()`, `fsummarise()`, `across()` (internal), `roworder[v]()`, `colorder[v]()`, `[f/set]rename()`, `[set]relabel()`

`set*` functions modify data by reference. `fgroup_by()` creates a class-agnostic grouped data frame (i.e. all other methods apply).

```
# Grouping data.table and scaling within groups
qDT(mtcars) |> fgroup_by(cyl, vs, am) |> fselect(-(hp:qsec)) |> fscale()
##     cyl vs am        mpg       disp       gear       carb
##  1:   6  0  1  0.5773503  0.5773503 -0.5773503 -0.5773503
##  2:   6  0  1  0.5773503  0.5773503 -0.5773503 -0.5773503
## ---
## 31:   8  0  1 -0.7071068 -0.7071068        NaN  0.7071068
## 32:   4  1  1 -1.4652937  1.6593867 -0.3779645  1.0690450
##
## Grouped by:  cyl, vs, am  [7 | 5 (3.8) 1-12]
```

Stats: `[N.groups | Mean (SD) Min-Max]` - latter 4 on group sizes.

# collapse is efficient: in syntax evaluation, grouping and computing

```
fdim(wlddev)
## [1] 13176    13
microbenchmark( # A typical short pipeline
  collapse_base = qTBL(wlddev) |>
    fsubset(year >= 1990) |>
    fmutate(ODA_POP = ODA / POP) |>
    fgroup_by(region, income, OECD) |>
    fsummarise(across(PCGDP:POP, sum, na.rm = TRUE)) |>
    roworder(income, -PCGDP),

  collapse_optimized = qTBL(wlddev) |>
    fsubset(year >= 1990, region, income, OECD, PCGDP:POP) |>
    fmutate(ODA_POP = ODA / POP) |>
    fgroup_by(1:3, sort = FALSE) |> fsum() |>
    roworder(income, -PCGDP),

  dplyr = qTBL(wlddev) |>
    filter(year >= 1990) |>
    mutate(ODA_POP = ODA / POP) |>
    group_by(region, income, OECD) |>
    summarise(across(PCGDP:POP, sum, na.rm = TRUE), .groups = "drop") |>
    arrange(income, desc(PCGDP)),

  data.table = qDT(wlddev)[, ODA_POP := ODA / POP][
   year >= 1990, lapply(.SD, sum, na.rm = TRUE),
   by = .(region, income, OECD), .SDcols = PCGDP:ODA_POP][
   order(income, -PCGDP)])
## Unit: microseconds
##                expr       min         lq       mean     median         uq       max neval
##       collapse_base   364.039   436.7115   519.6012   471.1925   494.6445  4407.254   100
##  collapse_optimized   258.054   289.5010   327.5986   306.1880   328.4305  1546.397   100
##               dplyr 16756.454 17556.7125 19484.8138 19399.4780 20603.2995 37717.950   100
##          data.table  1051.445  1144.8635  1395.1398  1269.0115  1310.8725  3789.876   100
```

## With *Fast Statistical Functions*, `fsummarise` and `fmutate` essentially translate syntax.

```r
library(magrittr)
# Ad-hoc grouping, often fastest
mtcars |> ftransform(mpg_sum = fsum(mpg, cyl, TRA = "replace_fill"))
#
# ftransform() natively ignores grouped data, these are also equivalent
mtcars %>% fgroup_by(cyl) %>% ftransform(mpg_sum = fsum(mpg, GRP(.), TRA = "replace_fill"))
mtcars |> fgroup_by(cyl) |> fmutate(mpg_sum = fsum(mpg))
#
# The syntax translation allows for cutomizations e.g. specifying a different TRA argument
mtcars |> fgroup_by(cyl) |> fmutate(mpg_prop = mpg / fsum(mpg))      # Slower
mtcars |> fgroup_by(cyl) |> fmutate(mpg_prop = fsum(mpg, TRA = "/")) # Faster
#
# Whenever a fast function is found in a call, the whole expression is vectorized
mtcars |> fgroup_by(cyl) |> fmutate(mpg_prop2 = fsum(mpg) / sum(mpg)) # This does not give ones
#
# ftransform is useful also because it allows nested pipelines
mtcars %>% fgroup_by(cyl) %>% ftransform(fselect(., hp:qsec) %>% fbetween() %>%
                                           fungroup() %>% fsum(TRA = "/"))
# These two are thus also equivalent
mtcars %>% fgroup_by(cyl) %>% ftransform(fselect(., hp:qsec) %>% fsum(TRA = "/"))
mtcars %>% fgroup_by(cyl) %>% fmutate(across(hp:qsec, fsum, TRA = "/"))
# Finally, with fast functions we can always add set = TRUE, to transform by reference
mtcars %>% fgroup_by(cyl) %>% fmutate(across(hp:qsec, fsum, TRA = "/", set = TRUE)) |> invisible()
# -> Note that the data is not grouped by reference, only transformed, so mtcars is not grouped
# An added feature of across() is that we can apply functions to data subsets with .apply = FALSE
mtcars %>% fgroup_by(cyl) %>% fsummarise(across(hp:qsec, \(x) qDF(pwcor(x), "var"), .apply = FALSE))
```

Apart from `fsummarise` and `fmutate`, **all** other manipulation functions *deliberately ignore* grouped data!

# Further Functions: Select, Replace and Add Variables

```r
## Select and replace columns by names, indices, logical vectors,
## regular expressions or using functions to identify columns
get_vars(x, vars, return = "data", regex = FALSE, ...)
get_vars(x, vars, regex = FALSE, ...) <- value

## Add columns at any position within a data.frame
add_vars(x, ..., pos = "end")
add_vars(x, pos = "end") <- value
```

Can also select and replace by data type, or return names or indices of variables of a certain type: fully in C!

```r
## Select and replace columns by data type
num_vars(x, return = "data")
num_vars(x) <- value
cat_vars(x, return = "data")  # !is.numeric
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") # 'Date', 'POSIXct' or 'POSIXlt'
date_vars(x) <- value
```

# Other Programming Functions

*Quick Conversions:* matrices ⇔ data[frames/tables/tibbles] ⇔ lists

`qDF(), qDT(), qTBL(), qM(), mrtl(), mctl()`

*Selected Utilities*

`massign(), %=%, .c(), vlabels[<-](), namlab(), copyAttrib(), unattrib(), %!in%, f[n]unique()`

*(Memory) Efficient Programming*

```
[any/all]v(x, value)          # Faster than any/all(x == value)
allNA(x)                      # Faster than all(is.na(x))
whichv(x, value, invert = F)# Faster than which(x (!/=)= value)
whichNA(x, invert = FALSE)   # Faster than which((!)is.na(x))
x %(!/=)=% value              # Infix for whichv(v, value, TRUE/FALSE)
setv(X, v, R, ...)            # x[x(!/=)=v]<-r / x[v]<-r[v] (by reference)
setop(X, op, V, rowwise = F)# Faster than X <- X +/-/*// V (by reference)
X %(+/-/*//)=% V              # Infix for setop(X, "+/-/*//", V)
na_rm(x)                      # Fast: if(anyNA(x)) x[!is.na(x)] else x,
na_omit(X, cols = NULL, ...)# Faster na.omit for matrices and data frames
vlengths(X, use.names=TRUE) # Faster version of lengths()
frange(x, na.rm = TRUE)      # Much faster base::range
fdim(X)                       # Faster dim for data frames
```

# Case Study: Open Street Map

This is work in progress to evaluate the effects of road quality improvement on local economic activity in Africa at a continent scale using Open Street Map. The approach followed here is to study the growth of the map in the vicinity of upgraded roads, assigning weights to different types of features (shops, restaurants, etc.) based on how they relate to aggregate spatial activity.

```r
library(qs)
library(collapse)
library(data.table)
library(fastmap)

features_long <- qread(paste0(path, "/africa_osm_history_2010_2022_features_long.qs"))
nearest_features <- qread(paste0(path, "/roads_nearest_features.qs"))
qread(paste0(path, "/feature_weights.qs")) |> list2env(envir = environment())
## <environment: R_GlobalEnv>
#
# Setting monthly frequency and getting rid of sub-monthly changes.
features_long <- features_long |>
  ftransform(validFromMonth = zoo::as.yearmon(validFrom),
             validToMonth = zoo::as.yearmon(validTo)) |>
  funique(cols = .c(osmId, validFromMonth, validToMonth)) |>
  fsubset(validFromMonth < validToMonth) |>
  fgroup_by(osmId) |>
  fsummarise(category = fmode(category),
             validFromMonth = fmin(validFromMonth),
             validToMonth = fmax(validToMonth)) |> qDT()
```

```
# OSM Africa, changes in 26 feature categories since 2010.
head(features_long, 3)
##              osmId      category validFromMonth validToMonth
## 1: node/1000010065          shop       Nov 2010     Aug 2015
## 2: node/1000064915        school       Nov 2010     Jan 2022
## 3: node/1000064925 amenity_other       Nov 2010     Jan 2022
fndistinct(features_long)
##          osmId       category validFromMonth   validToMonth
##        2756810             26            144            144
# Jan 2010 to Jan 2022 to be precise
features_long |> fselect(validFromMonth, validToMonth) |> dapply(frange)
##    validFromMonth validToMonth
## 1:       Jan 2010     Feb 2010
## 2:       Dec 2021     Jan 2022
#
# OSM roads with all features <= 25km from the road (+distance in m)
str(nearest_features[1:3])
## List of 3
##  $ way/1000335115: Named num [1:51] 24118 19306 7640 18133 17331 ...
##   ..- attr(*, "names")= chr [1:51] "node/7963593285" "node/8007133985" "way/1000335133" "way/615633421'
##  $ way/1000335119: Named num [1:38] 12138 220 21243 23954 22987 ...
##   ..- attr(*, "names")= chr [1:38] "node/8007133985" "way/1000335133" "way/509844677" "way/615633421" 
##  $ way/1000356515: Named num [1:98] 24940 24787 24481 24211 24008 ...
##   ..- attr(*, "names")= chr [1:98] "node/2290522162" "node/2290522164" "node/2290522166" "node/2290522
qsu(vlengths(nearest_features))
##        N       Mean         SD  Min    Max
##    69961  1073.0581  3577.5943    0  88054
#
# Finally, a set of weights reflecting features contribution to economic activity
qsu(parametric$weights) # nonparametric$weights: alternative weights estimate
##    N    Mean      SD      Min      Max
##   26  4.3706  4.5139  -0.0814  18.6303
sort(parametric$weights, decreasing = TRUE) |> head()
##   marketplace          line          fuel amenity_other    university       edu_alt
##     18.630274     12.593628      9.816919      9.341456      8.687138      6.973102
```

```r
# Creating monthly timeline.
timeline <- zoo::as.yearmon(seq(2010, 2022, by = 1/12))
(ng <- length(timeline))
## [1] 145
#
# Here creating uniform 'qG' columns representing time, and weight columns
settransform(features_long,
  VFMg = timeid(validFromMonth) |> setattr("N.groups", ng),
  VTMg = timeid(validToMonth) %+=% 1L |> setattr("N.groups", ng),
  w_unit = 1,                                   # Baseline: no feature specific weights
  w_pm = unname(parametric$weights[category]),    # Parametric (main) estimate
  w_npm = unname(nonparametric$weights[category]) # Nonparametric estimate
)
head(features_long, 3)
##               osmId      category validFromMonth validToMonth VFMg VTMg w_unit       w_pm     w_npm
## 1: node/1000010065          shop       Nov 2010     Aug 2015   11   68      1 0.2127659 4.000000
## 2: node/1000064915        school       Nov 2010     Jan 2022   11  145      1 1.7990378 1.975000
## 3: node/1000064925 amenity_other       Nov 2010     Jan 2022   11  145      1 9.3414556 5.833333
#
# Creating a plain list of weights and time variables
fll <- .subset(features_long, .c(VFMg, VTMg, w_unit, w_pm, w_npm))
#
# Using fastmap to do a large lookup of the nearest feature indices
m <- fastmap()
m$mset(.list = features_long |> with(setNames(as.list(seq_along(osmId)), osmId)))
system.time(indlist <- lapply(nearest_features, function(x) m$mget(names(x))))
##    user  system elapsed
##  69.106   0.317  69.483
m$reset(); rm(m); gc()
##              used   (Mb) gc trigger   (Mb) limit (Mb)  max used   (Mb)
## Ncells   6882241  367.6   18918195 1010.4         NA  23647743 1263.0
## Vcells 261136306 1992.4  403405934 3077.8      16384 797096613 6081.4
# Just a vector of weight variable names for efficient retrieval
w <- .c(w_unit, w_pm, w_npm)
```

```r
# Aggregate nearest features for each road using activity and inverse distance weights.
do_agg <- function(x, ind) {      # For each nearest-feature distance vector to a road x
  if(length(x) < 1L) return(NULL)
  names(x) <- NULL
  x <- x[vlengths(ind, use.names = FALSE) > 0L]
  ind <- unlist(ind, use.names = FALSE)
  if(is.null(ind)) return(NULL)
  x[x < 100] <- 100 # If within 100m of the road, the inverse distance weight is one
  x <- 100 / x      # This gets inverse distance weights
  fllind <- ss(fll, ind) # Efficient subsetting: get nearest features
  # Inverse-distance-weighted aggregation by month of initialization
  res <- fsum(fllind[w], fllind$VFMg, x, use.g.names = FALSE, na.rm = FALSE)
  # If some features are phased out before Jan 2022, need to subtract again
  if(!allv(fllind$VTMg, ng)) {
    res2 <- fsum(fllind[w], fllind$VTMg, x, use.g.names = FALSE, na.rm = FALSE)
    res %-=% flag(res2, fill = 0, stubs = FALSE) # Phasing them out in the next period
  }
  c(list(YearMonth = timeline), fcumsum(res, na.rm = FALSE)) # Cumulatively summing everything
}
#
# Computing: approx. 80 microseconds per road!!
system.time(roads_activity <- Map(do_agg, nearest_features, indlist))
##    user  system elapsed
##   5.863   0.381   6.250
#
# Creating final frame: balanced 145 months panel data
(roads_activity <- rbindlist(roads_activity, idcol = "osmId") |> setorder(osmId, YearMonth))
##                    osmId YearMonth   w_unit     w_pm   w_npm
##       1: way/1000335115  Jan 2010  0.00000  0.00000  0.0000
##       2: way/1000335115  Feb 2010  0.00000  0.00000  0.0000
##      ---
## 9609004:   way/99961478  Dec 2021 47.10345 70.63715 21.0553
## 9609005:   way/99961478  Jan 2022 47.10345 70.63715 21.0553
```

# Time Series and Panel Series

*collapse* provides a flexible and high-performance architecture to perform *time aware* computations on time series and panel data.

```
fgrowth(airmiles) # growth rate
## Time Series:
## Start = 1937
## End = 1960
## Frequency = 1
##  [1]          NA 16.50485437 42.29166667 54.02635432 31.65399240  2.38267148
##  [7] 15.23272214 33.29253366 54.36179982 76.91850089  2.70679220 -2.09526927
## [13] 12.90754055 18.51029172 32.02549044 18.56899489 17.81609195 13.61111111
## [19] 18.18832369 12.83112165 13.31723459  0.01183899 15.49145721  4.25364720
#
# Creating irregular series by removing 5 random obs.
rmind <- -sample.int(length(airmiles), 5)
am_ir <- airmiles[rmind] # subsetting removes the class
t <- time(airmiles)[rmind] |> timeid()
#
# Computations with collapse are fully time aware
fgrowth(am_ir, t = t)
##  [1]        NA        NA 31.653992  2.382671 15.232722 33.292534        NA
##  [8]  2.706792 -2.095269 12.907541        NA 18.568995 17.816092 13.611111
## [15] 18.188324 12.831122        NA 15.491457  4.253647
#
# And very general (here compounding to quarterly growth rates)
fgrowth(am_ir, -1:3, power = 1/4, t = t) |> head()
##             FG1   --        G1      L2G1     L3G1
## [1,]         NA  480        NA        NA       NA
## [2,] -6.6441599 1052        NA 21.672835       NA
## [3,] -0.5869529 1385 7.1170265        NA 30.33232
## [4,] -3.4825044 1418 0.5904184  7.749465       NA
## [5,] -6.9323937 1634 3.6081586  4.219880 11.63724
## [6,]         NA 2178 7.4487718 11.325694 11.98298
```

```
# Sequence of lagged / leaded and iterated matrix differences
fdiff(EuStockMarkets[, c("DAX", "SMI")], n = -1:1, diff = 1:2)
## Time Series:
## Start = c(1991, 130)
## End = c(1998, 169)
## Frequency = 260
##          FD1.DAX FD2.DAX     DAX  D1.DAX  D2.DAX FD1.SMI        FD2.SMI    SMI D1.SMI         D2.SMI
## 1991.496   15.12    8.00 1628.75      NA      NA   -10.4 -2.030000e+01 1678.1     NA             NA
## 1991.500    7.12   21.65 1613.63  -15.12      NA     9.9  1.540000e+01 1688.5   10.4             NA
## 1991.504  -14.53  -17.41 1606.51   -7.12    8.00    -5.5 -3.000000e+00 1678.6   -9.9 -2.030000e+01
## 1991.508    2.88   -4.67 1621.04   14.53   21.65    -2.5 -1.750000e+01 1684.1    5.5  1.540000e+01
## 1991.512    7.55   27.69 1618.16   -2.88  -17.41    15.0  2.630000e+01 1686.6    2.5 -3.000000e+00
##  [ reached getOption("max.print") -- omitted 1855 rows ]
#
# Here adding growth rates to panel data (apply = FALSE ensures we use fgrowth.data.frame)
ftransformv(wlddev, c(PCGDP, LIFEEX, POP), fgrowth, c(1, 10), g = iso3c, t = year,
            apply = FALSE, stubs = TRUE)
##       country iso3c       date year decade     region     income  OECD PCGDP LIFEEX GINI       ODA
## 1 Afghanistan   AFG 1961-01-01 1960   1960 South Asia Low income FALSE    NA 32.446   NA 116769997
## 2 Afghanistan   AFG 1962-01-01 1961   1960 South Asia Low income FALSE    NA 32.962   NA 232080002
##       POP G1.PCGDP L10G1.PCGDP G1.LIFEEX L10G1.LIFEEX   G1.POP L10G1.POP
## 1 8996973       NA          NA        NA           NA       NA        NA
## 2 9169410       NA          NA  1.590335           NA 1.916611        NA
##  [ reached 'max' / getOption("max.print") -- omitted 13174 rows ]
#
# Another panel, with 2 id variables, computing decadal compound growth rates
ftransformv(GGDC10S, c(AGR:MAN, SUM), fgrowth, 10, g = list(Variable, Country), t = Year,
            power = 1/10, apply = FALSE) |> na_omit()
##   Country Regioncode             Region Variable Year      AGR      MIN      MAN        PU      CON
## 1     BWA        SSA Sub-saharan Africa       VA 1974 12.87397 18.60544 35.68236  3.016397 23.18768
## 2     BWA        SSA Sub-saharan Africa       VA 1975 14.57275 24.11740 36.65907  6.307011 23.18768
## 3     BWA        SSA Sub-saharan Africa       VA 1976 14.03928 36.61835 44.29743 10.146062 30.34010
##        WRT      TRA      FIRE      GOV       OTH      SUM
## 1 21.64702 8.977434  7.762307 19.85785  8.977562 17.18065
## 2 26.72287 8.977434 11.325868 27.02873 10.148548 18.77042
```

*collapse* supported *plm*'s 'pseries' and 'pdata.frame' classes from the very beginning. Flexibility and performance considerations lead to the creation of new classes 'indexes_series' and 'indexed_frame' to bring high-performance time-aware computations to **all of R**!

```
# GDP per Capita, Life Expectancy, and Population for Germany since 1970, with artifical gaps at beginning
GER <- fsubset(wlddev, country == "Germany" & year >= 1970, year, PCGDP, LIFEEX, POP) |> ss(-c(2, 5))
# Indexing this data
GERI <- GER |> findex_by(year)
GERI # Tells us already that there are 2 missing periods
##   year    PCGDP   LIFEEX      POP
## 1 1970 19681.32 70.63978 78169289
## 2 1972 21031.08 70.86700 78688452
## 3 1973 21966.55 71.01668 78936666
##  [ reached 'max' / getOption("max.print") -- omitted 46 rows ]
##
## Indexed by:  year [49 (51)]
# Computing growth rates (G = shortcut with some convenient defaults)
G(GERI)
##   year G1.PCGDP G1.LIFEEX    G1.POP
## 1 1970       NA        NA        NA
## 2 1972       NA        NA        NA
## 3 1973 4.448017 0.2112167 0.3154389
##  [ reached 'max' / getOption("max.print") -- omitted 46 rows ]
##
## Indexed by:  year [49 (51)]
# Manipulation
G(GERI[1:10,1:3])
##   year G1.PCGDP G1.LIFEEX
## 1 1970       NA        NA
## 2 1972       NA        NA
## 3 1973 4.448017 0.2112167
## 4 1975       NA        NA
## 5 1976 5.400212 0.3255028
```

```
GERI |> fsubset(year < 2000, PCGDP, LIFEEX) |> fgrowth()
##      PCGDP    LIFEEX
## 1       NA        NA
## 2       NA        NA
## 3 4.448017 0.2112167
## 4       NA        NA
## 5 5.400212 0.3255028
## 6 3.581437 0.3483487
## 7 3.098182 0.3611526
##  [ reached 'max' / getOption("max.print") -- omitted 21 rows ]
##
## Indexed by:  year [28 (51)]
fgrowth(GERI$PCGDP) # creating indexed series: transfers index to PCGDP vector
##  [1]        NA        NA  4.4480172        NA  5.4002121  3.5814371  3.0981816  4.1043313
##  [9]  1.1986939  0.3762425 -0.3000578  1.8390342  3.1789872  2.5568836  2.2405350
##  [ reached getOption("max.print") -- omitted 34 entries ]
## attr(,"label")
## [1] "GDP per capita (constant 2010 US$)"
##
## Indexed by:  year [49 (51)]
flag(fgrowth(GERI$PCGDP, c(1, 10)), c(1, 2)) # Supports nested computations
##          L1.G1     L2.G1  L1.L10G1  L2.L10G1
##  [1,]       NA        NA        NA        NA
##  [2,]       NA        NA        NA        NA
##  [3,]       NA        NA        NA        NA
##  [ reached getOption("max.print") -- omitted 46 rows ]
## attr(,"label")
## [1] "GDP per capita (constant 2010 US$)"
## attr(,"class")
## [1] "numeric" "matrix"
##
## Indexed by:  year [49 (51)]
```

```
# The amazing thing is that these also work: without any special methods being created
with(GERI, fgrowth(PCGDP))
##  [1]        NA        NA  4.4480172        NA  5.4002121  3.5814371  3.0981816  4.1043313
##  [9]  1.1986939  0.3762425 -0.3000578  1.8390342  3.1789872  2.5568836  2.2405350
##  [ reached getOption("max.print") -- omitted 34 entries ]
## attr(,"label")
## [1] "GDP per capita (constant 2010 US$)"
##
## Indexed by:  year [49 (51)]
ftransform(GERI, PCGDP_growth = fgrowth(PCGDP))
##   year    PCGDP   LIFEEX      POP PCGDP_growth
## 1 1970 19681.32 70.63978 78169289           NA
## 2 1972 21031.08 70.86700 78688452           NA
## 3 1973 21966.55 71.01668 78936666     4.448017
##  [ reached 'max' / getOption("max.print") -- omitted 46 rows ]
##
## Indexed by:  year [49 (51)]
# This works not only with lm(), but ANY statistical model with standard formula interpretation...
coef(lm(G(PCGDP) ~ G(LIFEEX), GERI))
## (Intercept)   G(LIFEEX)
##    1.097664    2.603599
coef(lm(G(PCGDP) ~ G(LIFEEX), unindex(GERI))) # Removing index: wrong result
## (Intercept)   G(LIFEEX)
##    1.205007    2.436394
# including packages like 'fixest', 'lfe' and 'plm' (see also ?to_plm)
library(fixest)
wlddev |>
  findex_by(iso3c, year) |>
  feols(G(PCGDP) ~ G(LIFEEX) | iso3c)
## OLS estimation, Dep. Var.: G(PCGDP)
## Observations: 8,806
## Fixed-effects: iso3c: 190
## Standard-errors: Clustered (iso3c)
##           Estimate Std. Error t value   Pr(>|t|)
## G(LIFEEX) 0.696164   0.164408 4.23438 3.5747e-05 ***
```

# All of this generalizes to complex panel data

```
GGDCI <- GGDC10S |> na_omit() |> ss(-3) |> findex_by(Variable, Country, Year)
G(GGDCI, cols = c("AGR", "MAN", "SUM"))
##   Country Variable Year    G1.AGR    G1.MAN     G1.SUM
## 1     BWA       VA 1964        NA        NA         NA
## 2     BWA       VA 1965 -3.524492  38.23529  4.9751645
## 3     BWA       VA 1967        NA        NA         NA
## 4     BWA       VA 1968 10.204082 -20.00000 -0.6102057
## 5     BWA       VA 1969  3.614458 185.18519 24.4977512
##  [ reached 'max' / getOption("max.print") -- omitted 3370 rows ]
##
## Indexed by:  Variable.Country [67] | Year [67]
```

# So how does this work?

```
# An 'index_df' = a data frame of factors identifying individual and/or time dimensions is attached
str(findex(GGDCI)) # the time factor has extra levels for missing periods
## Classes 'index_df', 'pindex' and 'data.frame': 3375 obs. of  2 variables:
##  $ Variable.Country: Factor w/ 67 levels "VA.BWA","EMP.BWA",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Year            : Ord.factor w/ 67 levels "1947"<"1948"<..: 18 19 21 22 23 24 25 26 27 28 ...
##  - attr(*, "nam")= chr [1:3] "Variable" "Country" "Year"
# Each series in the frame is an indexed series, with an external pointer to the index
str(with(GGDCI, AGR))
##  'indexed_series' num [1:3375] 16.3 15.7 19.1 21.1 21.9 ...
##  - attr(*, "label")= chr "Agriculture "
##  - attr(*, "format.stata")= chr "%10.0g"
##  - attr(*, "index_df")=<externalptr>
# The index can be fetched from that pointer inside ANY data masking environments
str(with(GGDCI, findex(AGR)))
## Classes 'index_df', 'pindex' and 'data.frame': 3375 obs. of  2 variables:
##  $ Variable.Country: Factor w/ 67 levels "VA.BWA","EMP.BWA",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Year            : Ord.factor w/ 67 levels "1947"<"1948"<..: 18 19 21 22 23 24 25 26 27 28 ...
##  - attr(*, "nam")= chr [1:3] "Variable" "Country" "Year"
# All of this is done in a way that does not interfere with the functionality of the object!
```

# Recall the OSM roads data generated

```r
# Let's add some fictitious spatial information
library(sf)
point <- st_sfc(st_point(c(0,0)), crs = 4326)
roads_activity_nzw_sf <- roads_activity |>
  fsubset(w_unit > 0) |> # Removing zero weights: crating an unbalanced panel..
  fmutate(geometry = copyAttrib(lapply(seq_along(osmId), \(i) point[[1]]), point)) |>
  st_as_sf(sf_column_name = "geometry")
fdim(roads_activity_nzw_sf)
## [1] 7387440       6
# Indexing
roads_activity_nzw_sf <- findex_by(roads_activity_nzw_sf, osmId, YearMonth)
# This is a fully identified spatiotemporal panel structure
st_coordinates(roads_activity_nzw_sf) |> head(2) # It supports spatial operations ...
##   X Y
## 1 0 0
## 2 0 0
# ... and time series / panel data operations, here computing centered growth rates
settransform(roads_activity_nzw_sf, w_pm_wgr = fwithin(fgrowth(w_pm))) |> system.time()
##    user  system elapsed
##   0.057   0.012   0.069
# Subsetting and other methods apply, note how the index was also subset
roads_activity_nzw_sf[1:3, ]
## Simple feature collection with 3 features and 6 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 0 ymin: 0 xmax: 0 ymax: 0
## Geodetic CRS:  WGS 84
##             osmId YearMonth    w_unit      w_pm     w_npm    geometry  w_pm_wgr
## 1 way/1000335115  Aug 2018 0.1145512 0.1302773 0.5057988 POINT (0 0)        NA
## 2 way/1000335115  Sep 2018 0.1145512 0.1302773 0.5057988 POINT (0 0) -17.25371
##  [ reached 'max' / getOption("max.print") -- omitted 1 rows ]
##
## Indexed by:  osmId [1] | YearMonth [3 (145)]
```

# How fast is this structure?

```
roads_sf <- unindex(roads_activity_nzw_sf) # Removing index + shortening name
roads_dt <- qDT(roads_sf); system.time(setkey(roads_dt, osmId)) # data.table + keying
##    user  system elapsed
##   0.101   0.010   0.111
system.time(roads_fp <- panel(qDT(roads_dt), ~ osmId + YearMonth)) # fixest panel
##    user  system elapsed
##  33.941   0.574  34.540
microbenchmark(indexing = roads_sf <- findex_by(roads_sf, osmId, YearMonth), times = 10)
## Unit: milliseconds
##      expr      min       lq     mean   median       uq      max neval
##  indexing 55.03229 61.58909 67.20255 64.10924 65.54912 106.3928    10
microbenchmark(lag = fmutate(roads_sf, lag = flag(w_pm)),
  lag_dt = qDT(roads_sf)[, lag := flag(w_pm)], # This works, because of the extptr
  lag_g = fmutate(roads_sf, lag = flag(w_pm, shift = "row")), # Incorrect, what data.table does
  diff = fmutate(roads_sf, diff = fdiff(w_pm)),
  center = fmutate(roads_sf, center = fwithin(w_pm)),
  DT_lag = roads_dt[, lag := shift(w_pm), keyby = osmId],          # Incorrect
  DT_diff = roads_dt[, diff := w_pm - shift(w_pm), keyby = osmId], # Incorrect
  DT_center = roads_dt[, center := w_pm - mean(w_pm), keyby = osmId],
  fixest_lag = roads_fp[, lag := l(w_pm)],
  fixest_diff = roads_fp[, diff := d(w_pm)], times = 10)
## Unit: milliseconds
##         expr        min         lq      mean     median         uq        max neval
##          lag  28.730832  28.896267  29.99168  29.145485  31.121583   32.91582    10
##       lag_dt  32.987698  33.387735  41.30091  35.001843  35.882011   98.10410    10
##        lag_g   7.671838   7.834485  11.88754   8.488927   9.239555   42.79162    10
##         diff  30.576652  30.925357  58.64716  32.986325  35.295629  264.72097    10
##       center  32.739771  32.963303  34.61904  33.315964  35.967250   39.75462    10
##       DT_lag 325.861604 329.561813 431.76838 343.098783 388.580452 1155.66770    10
##      DT_diff 345.026275 346.151438 392.51333 373.496368 398.447307  586.45371    10
##    DT_center 141.834006 143.441985 157.96573 149.356522 163.362901  222.14292    10
##   fixest_lag  74.607126  75.698464  95.27107  79.822080 103.803185  172.92242    10
##  fixest_diff  74.187983  75.692355  94.17714  78.888715  97.625059  160.25231    10
```

# Time Series & Panel Series: Summary

*Classes, Constructors and Utilities*

`findex_by()`, `findex()`, `reindex()`, `unindex()`, `is_irregular()`, `to_plm()`, `timeid()`, and a rich set of S3 methods for 'indexed_frame', 'indexed_series' and 'index_df'.

*Core Time-Based Functions*

`flag()`, `fdiff()`, `fgrowth()`, `fcumsum()`, `psmat()` [panel data to array conversions] and `psacf()`, `pspacf()`, `psccf()` [autocorrelation functions for panel data]

*Data Transformation Functions with Supporting Methods*

`f[hd]between()`, `f[hd]within()`, `fscale()` [scaling and (higher-dimensional) centering]

*Data Manipulation Functions with Supporting Methods*

`fsubset()`, `funique()`, `roworder[v]()` [internal], and `na_omit()` [internal]

*Summary Functions with Supporting Methods*

`qsu()`, `varying()` [panel-variance decomposed statistics]

# APIs Extensions Ia: Statistical Operators

Extreme parsimony through function shorthands enables fast development and also facilitates ad-hoc use of functions, at the cost of readability. *collapse* implements this in two ways:

*(a) Statistical Operators for Data Transformation Functions*

```
.OPERATOR_FUN # Global macro with names of statistical operators
##  [1] "STD"  "B"    "W"    "HDB"  "HDW"  "L"    "F"    "D"    "Dlog" "G"
```

`fscale -> STD`

`f[hd]between -> [HD]B`

`f[hd]within -> [HD]W`

`flag -> L, F`

`fdiff -> D, Dlog`

`fgrowth -> G`

facilitate ad-hoc use e.g. `fsubset(mtcars, hp > B(hp, cyl))` or `lm(G(PCGDP) ~L(G(LIFEEX), 0:3), iby(wlddev, iso3c, year))` and have an enhanced data frame method e.g. `L(wlddev, 1:2, PCGDP ~iso3c, ~year)`.

# API Extensions Ib: Function Shorthands

*(b) Function Shorthands* obtained by compacting function names to the main consonants:

```
fselect -> slt
fsubset -> sbt
[f/set]transform[v] -> [set]tfm[v]
fmutate -> mtt
fsummarise -> smr
across -> acr
fgroup_by -> gby
findex_by -> iby
findex -> ix
frename -> rnm
get_vars -> gv
num_vars -> nv
add_vars -> av
```

These are simply shorthands that work just like their parents.

# API Extensions II: Namespace Masking

Many *collapse* functions begin with `f-` to signify performance improved versions of existing functions. Users can substitute these functions in-place by setting `options("collapse_mask")` before loading the package.

For example `options(collapse_mask = c("fselect", "fsubset"))` will export `select` and `subset` alsongside `fselect` and `fsubset` when loading *collapse*.

A few keywords exist to mask multiple functions: `"manip"`, `"helper"`, `"fast-fun"`, `"fast-stat-fun"`, `"fast-tfrm-fun"` and `"all"`. `"all"` masks all `f-` functions in the package.

The best way to set this option is inside an `.Rprofile` file placed in the user or project directory.

## This is now 100% *collapse* code running:

```
options(collapse_mask = "all")
library(collapse)

wlddev |>
  subset(year >= 1990) |>
  group_by(year) |>
  summarise(across(PCGDP:GINI, mean, w = POP),
            n = n())

sum(mtcars)
diff(EuStockMarkets)
droplevels(wlddev)
mean(nv(iris), g = iris$Species)
scale(nv(GGDC10S), g = GGDC10S$Variable)
unique(GGDC10S, cols = c("Variable", "Country"))
range(wlddev$date)

wlddev |>
  index_by(iso3c, year) |>
  mutate(PCGDP_lag = lag(PCGDP),
         PCGDP_diff = PCGDP - PCGDP_lag,
         PCGDP_growth = growth(PCGDP)) |>
  unindex()
```

# Documentation

To quickly learn more about *collapse*, use the documentation.

```
library(collapse)
help("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* | *Functions* |
|---|---|---|
| Fast Statistical Functions | Fast (grouped and weighted) statistical functions for vector, matrix, data frame and grouped data frames (class 'grouped_df', *dplyr* compatible). | `fsum`, `fprod`, `fmean`, `fmedian`, `fmode`, `fvar`, `fsd`, `fmin`, `fmax`, `fnth`, `ffirst`, `flast`, `fnobs`, `fndistinct` |
| Fast Grouping and Ordering | Fast (ordered) groupings from vectors, data frames, lists. 'GRP' objects are efficient inputs for programming with *collapse*'s fast functions. `fgroup_by` can attach them to a data frame, for fast dplyr-style grouped computations. Fast splitting of vectors based on 'GRP' objects. Fast radix-based ordering and hash-based grouping (the workhorses behind `GRP`). Fast unique values/rows, factor generation, vector grouping, interactions, dropping unused factor levels, generalized run-length type grouping and grouping of integer sequences and time vectors. | `GRP`, `as_factor_GRP`, `GRPN`, `GRPnames`, `is_GRP`, `fgroup_by`, `fgroup_vars`, `fungroup`, `gsplit`, `greorder`, `radixorder(v)`, `group`, `funique`, `fnunique`, `qF`, `qG`, `is_qG`, `finteraction`, `fdroplevels`, `groupid`, `seqid`, `timeid` |
| Fast Data Manipulation | Fast and flexible select, subset, summarise, mutate/transform, sort/reorder, rename and relabel data. Some functions modify by reference and/or allow assignment. In addition a set of (standard evaluation) functions for fast selecting, replacing or adding data frame columns, including shortcuts to select and replace variables by data type. | `fselect(<-)`, `fsubset/ss`, `fsummarise`, `fmutate`, `across`, `(f/set)transform(v)(<-)`, `fcompute(v)`, `roworder(v)`, `colorder(v)`, `(f/set)rename`, `(set)relabel`, `get_vars(<-)`, `add_vars(<-)`, `num_vars(<-)`, `cat_vars(<-)`, `char_vars(<-)`, |

# Conclusion

As an applied economist, *collapse* has pushed the boundaries of what I am capable of accomplishing with R, and made me more productive in my research. I hope you will find time to try it and have a similar experience. Its API will remain stable and the package will receive further development over the coming years. You are very welcome to contribute by any suitable means.

Thank you for your Attention!

**Links**

GitHub | Website | Twitter
(https://sebkrantz.github.io/collapse/)