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

Sebastian Krantz
Kiel Institute for the World Economy

Abstract

collapse is a large C/C++-based infrastructure package facilitating complex statistical computing, data transformation, and exploration tasks in R - at outstanding levels of performance and programming efficiency. It also implements a class-agnostic approach to R programming, supporting vector, matrix and data frame-like objects and their popular variants (e.g., '`factor`', '`ts`', '`xts`', '`tibble`', '`data.table`', '`sf`'), enabling its seamless integration with large parts of the R ecosystem. This article introduces the package's key components and design principles in a structured way, supported by a rich set of examples. A small benchmark demonstrates its computational performance.

Keywords: statistical computing, vectorization, data manipulation and transformation, summary statistics, class-agnostic programming, R.

1. Introduction

collapse[1] is a large C/C++-based R package that provides an integrated suite of statistical and data manipulation functions. Most of these statistical functions are vectorized along multiple dimensions (notably along groups and columns) and perform high-cardinality operations[2] very efficiently. It also offers vectorizations for advanced operations such as weighted statistics (including mode and quantiles), functions and classes for fully indexed (time-aware) computations on time series and panel data, recursive (list-processing) tools to deal with nested data and advanced descriptive statistical tools. This functionality is supported by efficient algorithms for intensive operations like grouping, unique values, matching, ordering, etc., tailored to R's data structures, and powerful data manipulation functions. The package also supplies many features for memory efficient R programming, such as data transformation and math by reference, and aversion of logical vectors. **collapse** is class-agnostic, i.e., it provides most statistical operations for atomic vectors, matrices, and data frames/lists, and seamlessly supports key variants of these objects used in the R ecosystem (e.g., '`tibble`', '`data.table`', '`sf`', '`xts`', '`pdata.frame`'). It is globally and interactively configurable, which includes setting different defaults for key function arguments (such as `na.rm` arguments to statistical functions, default `TRUE`), and modifying the package namespace itself.[3]

[1] Website: https://sebkrantz.github.io/collapse/

[2] With many columns and/or groups relative to data size.

[3] **collapse**'s namespace is fully compatible with base R and the **tidyverse**, but can be interactively modified to overwrite key functions like `unique`, `match`, `%in%`, `table`, `subset`, `mutate`, `summarise` etc. with much faster **collapse** equivalents. See Section 8.

What is the purpose of combining all of this in a package? The short answer is to make computations in R as flexible and powerful as possible. The more elaborate answer is to (1) facilitate complex data transformation, exploration, and computing tasks in R; (2) increase performance and parsimony by avoiding R-level repetition;[4] (3) increase the memory efficiency and flexibility of R programs;[5] and (4) to create a new foundation package for statistics and data manipulation in R that implements successful ideas developed in the R ecosystem and other programming environments such as Python or STATA (StataCorp LLC. 2023), including some new ideas, in a stable, high performance, and broadly compatible manor.[6]

R already has a large and tested data manipulation and statistical computing ecosystem. Notably, the **tidyverse** (Wickham *et al.* 2019) provides a consistent toolkit for data manipulation in R, centered around the '`tibble`' (Müller and Wickham 2023) object and tidy data principles (Wickham 2014). **data.table** (Dowle and Srinivasan 2023) provides an enhanced high-performance data frame with parsimonious data manipulation syntax. **sf** (Pebesma 2018) provides a data frame for spatial data and supporting functionality. **tsibble** (Wang *et al.* 2020) and **xts** (Ryan and Ulrich 2023) provide classes and operations for time series data, the former via an enhanced '`tibble`', the latter through an efficient matrix-based class. Econometric packages like **plm** (Croissant and Millo 2008) and **fixest** (Bergé 2018) also provide solutions to deal with panel data and irregularity in the time dimension. Packages like **matrixStats** (Bengtsson 2023) and **Rfast** (Papadakis *et al.* 2023) offer fast statistical calculations along the rows and columns of matrices and faster basic statistical procedures. **DescTools** (Signorell 2023) provides a wide variety of descriptive statistics, including weighted versions. **survey** (Lumley 2004) allows statistical computations on complex survey data. **labelled** (Larmarange 2023) provides tools to deal with labelled data. Packages like **tidyr** (Wickham *et al.* 2023b), **purrr** (Wickham and Henry 2023) and **rrapply** (Chau 2022) provide some functions to deal with nested data and messy structures.

collapse relates to and integrates key elements from these projects. It offers **tidyverse**-like data manipulation at the speed and stability of **data.table** for any data frame-like object. It can turn any vector/matrix/data frame into a time-aware indexed series or frame and perform operations such as lagging, differencing, scaling or centering, encompassing and enhancing core manipulation functionality of **plm**, **fixest**, and **xts**. It also performs fast (grouped, weighted) statistical computations along the columns of matrix-like objects, complementing and enhancing **matrixStats** and **Rfast**. Its low-level vectorizations and workhorse algorithms are accessible at the R and C-levels, unlike **data.table**, where most vectorizations and algorithms are internal. It also supports variable labels and intelligently preserves attributes of all objects, complementing **labelled**. It provides general (recursive) tools to deal with nested data, enhancing **tidyr**, **purrr**, and **rrapply**. Finally, it provides a small but consistent and powerful

[4]Such as applying R functions across columns or split-apply-combine computing to apply functions across groups or other divisions of data.

[5]E.g., by avoiding object conversions and the need for certain classes to do certain things, such as converting to data frame or '`data.table`' to do something "by groups" and then convert back to matrix to continue with linear algebra, and in general to reduce the need for metaprogramming.

[6]Examples of such ideas are **tidyverse** syntax, vectorized aggregations (**data.table**), data transformation by reference (Python, **pandas**), vectorized and verbose joins (**polars**, STATA), indexed time series and panel data (**xts**, **plm**), summary statistics for panel data (STATA), reshaping labelled data (myself) etc...

set of descriptive statistical tools, yielding sufficient detail for most data exploration purposes, requiring users to invoke packages like **DescTools** or **survey** only for specific statistics. In summary, **collapse** is a foundation package for statistical computing and data manipulation in R that enhances and integrates seamlessly with the R ecosystem while being outstandingly computationally efficient. A significant benefit is that, rather than piecing together a fragmented ecosystem oriented at different classes and tasks, many core computational tasks can be done with **collapse**, and easily extended by more specialized packages. This tends to result in R scripts that are shorter, more efficient, and more lightweight in dependencies.

Other programming environments such as Python and Julia now also offer computationally very powerful libraries for tabular data such as **DataFrames.jl** (Bouchet-Valat and Kamiński 2023), **Polars** (Vink *et al.* 2023), and **Pandas** (Wes McKinney 2010; **pandas** Development Team 2023), and supporting numerical libraries such as **Numpy** (Harris *et al.* 2020), or **StatsBase.jl** (JuliaStats 2023). In comparison with these, **collapse** offers a class-agnostic approach bridging the divide between data frames and atomic structures, has more advanced statistical capabilities,[7] supports recursive operations, variable labels, verbosity for critical operations such as joins, and is extensively globally configurable. In short, it is very utile for complex statistical workflows, rich datasets (e.g., surveys), and for integrating with different parts of the R ecosystem. On the other hand, **collapse**, for the most part, does not offer a sub-column-level parallel architecture and is thus not highly competitive with top frameworks, including **data.table**, on aggregating billion-row datasets with few columns.[8] Its vectorization capabilities are also limited to the statistical functions it provides and not, like **DataFrames.jl**, to any Julia function. However, as demonstrated in Section 3.1, vectorized statistical functions can be combined to calculate more complex statistics in a vectorized way.

The package has a built-in structured documentation facilitating its use. This documentation includes a central overview page linking to all other documentation pages and supplementary topic pages which briefly describe related functionality. The names of these extra pages are collected in a global macro `.COLLAPSE_TOPICS` and can be called directly with `help()`:

```
R> .COLLAPSE_TOPICS
```

```
 [1] "collapse-documentation"      "fast-statistical-functions"
 [3] "fast-grouping-ordering"      "fast-data-manipulation"
 [5] "quick-conversion"            "advanced-aggregation"
 [7] "data-transformations"        "time-series-panel-series"
 [9] "list-processing"             "summary-statistics"
[11] "recode-replace"              "efficient-programming"
[13] "small-helpers"               "collapse-options"
```

[7] Such as weighted statistics, including various quantile and mode estimators, support for fully time-aware computations on irregular series/panels, higher order centering, advanced (grouped, weighted, panel-decomposed) descriptive statistics etc., all supporting missing values.

[8] As can be seen in the DuckDB Benchmarks: **collapse** is highly competitive on the 10-100 million observations datasets, but deteriorates in performance at larger data sizes (except for joins where it remains competitive). There may be performance improvements for "long data" in the future, but, at present, the treatment of columns as fundamental units of computation is a tradeoff for the highly flexible class-agnostic architecture.

```
R> help("collapse-documentation")
```

collapse is too large and complex to fully present it in a single article, or even to present selected topics in depth. The following sections therefore briefly introduce its key components: (2) the *Fast Statistical Functions* and their (3) integration with data manipulation functions; (4) architecture for time series and panel data; (5) joins and reshaping; (6) list processing functions; (7) descriptive tools; and (8) global options. Section 9 provides a small benchmark, Section 10 concludes. For deeper engagement with **collapse**, a short vignette summarizing available documentation and resources is an excellent starting point.

2. Fast statistical functions

The *Fast Statistical Functions*, comprising `fsum()`, `fprod()`, `fmean()`, `fmedian()`, `fmode()`, `fvar()`, `fsd()`, `fmin()`, `fmax()`, `fnth()`, `ffirst()`, `flast()`, `fnobs()` and `fndistinct()`, are a consistent set of S3-generic statistical functions providing fully vectorized statistical operations in R. Specifically, operations such as calculating the mean via the S3 generic `fmean()` function are vectorized across columns and groups. They may also involve weights or transformations of the original data. The basic syntax of these functions is

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

with arguments `x` - data (vector, matrix or data frame-like), `g` - groups (atomic vector, list of vectors, or 'GRP' object), `w` - weights, `TRA` - transformation, `na.rm` - missing values, `use.g.names` - attach group names upon aggregation (if `g` is used), `drop` - drop dimensions (i.e., simplify to atomic vector if `is.null(g)` and `x` is matrix or data frame-like), `nthreads` - multithreading.[9] The following examples, taken from the **collapse** for **tidyverse** Users vignette demonstrate their basic usage to calculate (column-wise, grouped, weighted) statistics on different objects. As laid out in the vignette on object handling, statistical functions have basis S3 methods for vectors ('`default`'), '`matrix`', and '`data.frame`', which call corresponding C implementations that intelligently preserve object attributes. Thus, the functions can be applied to a broad set of '`matrix`' or '`data.frame`'-based objects without the need to define explicit methods. Users can also directly call the basis methods in case S3 dispatch does not yield the intended outcome. For example, `fmean.default(EuStockMarkets)` computes the mean of the entire matrix.

```
R> fmean(mtcars$mpg)

[1] 20.09

R> fmean(EuStockMarkets)

 DAX  SMI  CAC FTSE
2531 3376 2228 3566
```

[9] Not all functions are multithreaded, and parallelism is implemented differently for different functions (detailed in the documentation). The use of single instruction multiple data (SIMD) parallelism in single-threaded mode also implies limited gains from multithreading for simple operations such as `fsum()`.

```
R> fmean(mtcars[5:10])

   drat      wt    qsec      vs      am    gear
 3.5966  3.2173 17.8488  0.4375  0.4062  3.6875

R> fmean(mtcars$mpg, w = mtcars$wt)

[1] 18.55

R> fmean(mtcars$mpg, g = mtcars$cyl)

    4     6     8
26.66 19.74 15.10

R> fmean(mtcars$mpg, g = mtcars$cyl, w = mtcars$wt)

    4     6     8
25.94 19.65 14.81

R> fmean(mtcars[5:10], g = mtcars$cyl, w = mtcars$wt)

   drat    wt  qsec     vs     am  gear
4 4.031 2.415 19.38 0.9149 0.6498 4.047
6 3.569 3.152 18.12 0.6212 0.3788 3.821
8 3.206 4.133 16.89 0.0000 0.1204 3.241

R> fmean(mtcars$mpg, g = mtcars$cyl, TRA = "fill") |> head(20)

 [1] 19.74 19.74 26.66 19.74 15.10 19.74 15.10 26.66 26.66 19.74 19.74 15.10
[13] 15.10 15.10 15.10 15.10 15.10 26.66 26.66 26.66
```

2.1. Transformations

The final example invoking `TRA` expands the mean vector to full length, like `stats::ave(mtcars$mpg, mtcars$cyl)`, but much faster. The `TRA` argument invokes the `TRA()` function for column-wise (grouped) replacing and sweeping operations (by reference). Its syntax is

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

where `STATS` is a vector/matrix/data.frame of statistics used to transform `x`. Table 1 lists the 11 possible `FUN` operations, toggled using either an integer or string. `TRA()` is called internally in the *Fast Statistical Functions*, the `TRA` argument is passed to `FUN`. Thus `fmean(x, g, w, TRA = "-")` is equivalent to `TRA(x, fmean(x, g, w), "-", g)`. The `set` argument can also be passed to *Fast Statistical Functions* to toggle transformation by reference. The following examples demonstrate how this design allows flexible ad-hoc transformations using R's built-in `airquality` dataset with daily measurements in New York from May to September 1973.

Int	*String*	*Description*
0	`"replace_na"/"na"`	replace missing values in `x`
1	`"replace_fill"/"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`)

Table 1: Available `FUN` choices in `TRA()`.

```
R> fnobs(airquality)

  Ozone Solar.R    Wind    Temp   Month     Day
    116     146     153     153     153     153
```

This imputes columns `Ozone` and `Solar.R` by reference using the month median.

```
R> fmedian(airquality[1:2], airquality$Month, TRA = "replace_na", set = TRUE)
```

This performs different grouped and/or weighted transformations at once.

```
R> airquality |> fmutate(ozone_deg = Ozone / Temp,
+    rad_day = fsum(as.double(Solar.R), Day, TRA = "/"),
+    ozone_amed = Ozone > fmedian(Ozone, Month, TRA = "fill"),
+    ozone_resid = fmean(Ozone, list(Month, ozone_amed), ozone_deg, "-")
+    ) |> head(3)

  Ozone Solar.R Wind Temp Month Day ozone_deg rad_day ozone_amed ozone_resid
1    41     190  7.4   67     5   1    0.6119   0.191       TRUE     -10.279
2    36     118  8.0   72     5   2    0.5000   0.135       TRUE     -15.279
3    12     149 12.6   74     5   3    0.1622   0.168      FALSE      -3.035
```

2.2. Grouping objects and optimization

Whereas the `g` argument supports ad-hoc grouping with vectors and lists/data frames, the cost of grouping can be optimized by using factors or, even better, '`GRP`' objects, which readily contain all information **collapse**'s vectorized statistical functions might require to operate across groups. These objects can be created with `GRP()`. Its syntax is

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

The example below creates and displays a 'GRP' object from 3 columns in mtcars. The by argument also supports column names or indices, and X could also be an atomic vector.

```
R> str(g <- GRP(mtcars, ~ cyl + vs + am))

Class 'GRP'  hidden list of 9
 $ N.groups    : int 7
 $ group.id    : int [1:32] 4 4 3 5 6 5 6 2 2 5 ...
 $ group.sizes : int [1:7] 1 3 7 3 4 12 2
 $ groups      :'data.frame':        7 obs. of  3 variables:
  ..$ cyl: num [1:7] 4 4 4 6 6 8 8
  ..$ vs : num [1:7] 0 1 1 0 1 0 0
  ..$ am : num [1:7] 1 0 1 1 0 0 1
 $ group.vars  : chr [1:3] "cyl" "vs" "am"
 $ ordered     : Named logi [1:2] TRUE FALSE
  ..- attr(*, "names")= chr [1:2] "ordered" "sorted"
 $ order       : int [1:32] 27 8 9 21 3 18 19 20 26 28 ...
  ..- attr(*, "starts")= int [1:7] 1 2 5 12 15 19 31
  ..- attr(*, "maxgrpn")= int 12
  ..- attr(*, "sorted")= logi FALSE
 $ group.starts: int [1:7] 27 8 3 1 4 5 29
 $ call        : language GRP.default(X = mtcars, by = ~cyl + vs + am)
```

'GRP' objects make grouped statistical computations in **collapse** fully programmable. Below, the object is used with the *Fast Statistical Functions* and some utility functions to efficiently aggregate data (with optional frequency weights).

```
R> dat <- get_vars(mtcars, c("mpg", "disp")); w <- mtcars$wt;
R> add_vars(g$groups,
+    fmean(dat, g, w, use.g.names = FALSE) |> add_stub("w_mean_"),
+    fsd(dat, g, w, use.g.names = FALSE) |> add_stub("w_sd_")) |> head(2)

  cyl vs am w_mean_mpg w_mean_disp w_sd_mpg w_sd_disp
1   4  0  1      26.00       120.3    0.000       0.0
2   4  1  0      23.02       137.1    1.236      11.6
```

Similarly, data can be transformed, here using the S3 generic fscale() function.

```
R> mtcars |> add_vars(fmean(dat, g, w, "-") |> add_stub("w_demean_"),
+    fscale(dat, g, w) |> add_stub("w_scale_")) |> head(2)

              mpg cyl disp  hp drat    wt  qsec vs am gear carb w_demean_mpg
Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4       0.4357
Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4       0.4357
              w_demean_disp w_scale_mpg w_scale_disp
Mazda RX4             5.027      0.6657       0.6657
Mazda RX4 Wag         5.027      0.6657       0.6657
```

This programming access can become very useful. For example, the useR 2022 presentation Slides 18-19 aggregates the EORA Global Supply Chain Database from the country to the world region level. After defining a single grouping object, a list of value-added shares matrices (VB) and outputs (O) for years 1990-2021, is aggregated with no grouping cost using a single line of code like `lapply(VB_list, function(x) x$VB |> fsum(g) |> t() |> fmean(g, x$O) |> t())`. On an M1 Mac using 4 threads, this computation, involving 44.7 million summations and 2.6 million weighted means, takes only 0.33 seconds.[10]

3. Integration with data manipulation functions

collapse also provides a broad set of fast data manipulation functions familiar to R and **tidyverse** users, including `fselect()`, `fsubset()`, `fgroup_by()`, `fsummarise()`, `ftransform()`, `fmutate()`, `across()`, `frename()`, `fcount()`, etc. These are integrated with the *Fast Statistical Functions* to enable vectorized statistical operations in a familiar data frame oriented and **tidyverse**-like workflow. For example, the following code calculates the mean of columns

```
R> mtcars |> fsubset(mpg > 11) |> fgroup_by(cyl, vs, am) |>
+    fsummarise(across(c(mpg, carb, hp), fmean),
+               qsec_w_med = fmean(qsec, wt)) |> head(2)

  cyl vs am  mpg  carb    hp qsec_w_med
1   4  0  1 26.0 2.000 91.00      16.70
2   4  1  0 22.9 1.667 84.67      21.04
```

`mpg`, `carb` and `hp`, and the weighted mean of `qsec`, after subsetting and grouping the data. This code is very fast (especially with many groups) because data does not need to be split by groups at all. There is also no need to call `lapply()` inside the `across()` statement: `fmean.data.frame()` is applied to a subset of the data containing the three columns.[11] The *Fast Statistical Functions* also have a method for grouped data, so `fsummarise` is not always needed. The following example calculates weighted group means. By default (`keep.w = TRUE`) `fmean.grouped_df` also sums the weights in each group.[12]

```
R> mtcars |> fsubset(mpg > 11, cyl, vs, am, mpg, carb, hp, wt) |>
+    fgroup_by(cyl, vs, am) |> fmean(wt) |> head(2)

  cyl vs am sum.wt   mpg carb   hp
1   4  0  1  2.140 26.00 2.00 91.0
2   4  1  0  8.805 23.02 1.72 83.6
```

[10] Another recent example involved numerically optimizing a parameter a in an equation of the form $y_j = \sum_i x_{ij}^a \ \forall j \in J$ where there are J groups (1 million in my case), and the optimal value of a is determined by the proximity of the aggregated vector $\mathbf{y}$ to another vector $\mathbf{z}$. Thus each iteration of the numerical routine raises the vector $\mathbf{x}$ to a different power (a), sums it in 1 million groups (j) to generate $\mathbf{y}$, and computes the Euclidean distance to $\mathbf{z}$ (using `collapse::fdist`). Without grouping objects and vectorization, this would have been difficult to handle within reasonable computing times (of a few seconds on the M1).

[11] Internally, the `g` argument of the statistical functions is set as a keyword argument by `fsummarise/across` and the function is evaluated on a suitable subset of columns. Thus `w` becomes the second positional argument...

[12] '`grouped_df`' methods in **collapse** support grouped data created with either `fgroup_by()` or `dplyr::group_by()`. The latter requires an additional `C` routine to convert the **dplyr** grouping object to a '`GRP`' object, and is thus less efficient.

3.1. Vectorizations for advanced tasks

`fsummarise()` and `fmutate()` can also evaluate arbitrary statistical functions in the classical way (split-apply-combine) and handle more complex expressions involving multiple columns and/or functions. However, using any *Fast Statistical Function* causes the whole expression to be vectorized, i.e., evaluated only once and not for every group. This eager vectorization approach enables efficient grouped calculation of more complex statistics. The example below calculates grouped (`vs`) bivariate regression slopes (`mpg ~ carb`) in a vectorized way.

```
R> mtcars |> fgroup_by(vs) |>
+   fmutate(dm_carb = fmean(carb, TRA = "-")) |>
+   fsummarise(slope = fsum(mpg, dm_carb) %/=% fsum(dm_carb^2))

  vs   slope
1  0 -0.5557
2  1 -2.0706
```

Apart from vectorization, this code avoids 3 intermediate copies: (1) `fmean(carb, TRA = "-")` avoids an expanded vector of group means, (2) `fsum(mpg, dm_carb)` uses the weights (`w`) argument to `fsum()` to avoid materializing a multiplication (as in `fsum(mpg * dm_carb)`), and (3) division by reference (`%/=%`) avoids allocating an additional vector for the final result. Under the hood, the expression boils down to an (expensive) grouping step, 5 allocations (of which 2 full length), and 6 loops in C to calculate the result. Any modern laptop can calculate 1 million regression slopes in less than 1 second like this. Another very neat example, shared by Andrew Ghazi in a recent blog post,[13] vectorizes an expression to compute the p value across 300k groups for a simulation study, yielding a 70x performance increase over **dplyr**.

collapse also vectorizes advanced statistics, such as weighted medians and modes. The following example calculates a weighted set of summary statistics by groups, with weighted quantiles type 8 following Hyndman and Fan (1996).[14] and a weighted maximum mode.[15]

```
R> mtcars |> fgroup_by(cyl, vs, am) |>
+    fmutate(o = radixorder(GRPid(), mpg)) |>
+    fsummarise(mpg_min = fmin(mpg),
+               mpg_Q1 = fnth(mpg, 0.25, wt, o = o, ties = "q8"),
+               mpg_mean = fmean(mpg, wt),
+               mpg_median = fmedian(mpg, wt, o = o, ties = "q8"),
+               mpg_mode = fmode(mpg, wt, ties = "max"),
+               mpg_Q3 = fnth(mpg, 0.75, wt, o = o, ties = "q8"),
+               mpg_max = fmax(mpg)) |> head(3)

  cyl vs am mpg_min mpg_Q1 mpg_mean mpg_median mpg_mode mpg_Q3 mpg_max
1   4  0  1    26.0  26.00    26.00      26.00     26.0  26.00    26.0
2   4  1  0    21.5  21.90    23.02      23.16     24.4  24.38    24.4
3   4  1  1    21.4  22.37    27.74      28.28     30.4  31.51    33.9
```

[13] https://andrewghazi.github.io/posts/collapse_is_sick/sick.html

[14] **collapse** calculates weighted quantiles by replacing the sample size with the sum of weights and 1 with the minimum non-zero weight in the respective quantile definition. See fquantile for more details.

[15] The weighted maximum mode is the largest element with the maximum sum of weights.

Both weighted mode and quantiles have a sub-column parallel implementation,[16] and, as shown above, can also harness an (optional) optimization by computing an overall ordering vector and passing it to each quantile function to avoid repeated partial sorting (using quickselect) of the same elements within groups. For advanced data aggregation, **collapse** also provides a convenience function, `collap()`, which (by default) uses `fmean` for numeric and `fmode` for non-numeric columns. Below, it aggregates GDP per capita, life expectancy, and country name by World Bank income group, with population weights.[17] This yields population-weighted statistics, the largest country, and each income group's total population (sum of weights) for each year, preserving (default `keep.col.order = TRUE`) the order of columns.

```
R> collap(wlddev, country + PCGDP + LIFEEX ~ year + income, w = ~ POP) |>
+    head(4)

        country year              income   PCGDP LIFEEX       POP
1 United States 1960         High income 12768.7  68.59 7.495e+08
2      Ethiopia 1960          Low income   658.5  38.33 1.474e+08
3         India 1960 Lower middle income   500.8  45.27 9.280e+08
4         China 1960 Upper middle income  1166.1  49.86 1.184e+09
```

4. Time series and panel data

collapse also provides a flexible high-performance architecture to perform (time aware) computations on time series and panel series. In particular, the user enjoys great flexibility in deciding the desired degree of indexation and mode of computation. It is possible to apply time series and panel data transformations without any indexation by passing individual and/or time identifiers to the respective functions in an ad-hoc fashion, or by using '`indexed_frame`' and '`indexes_series`' classes, which implement full and deep indexation. Table 2 summarizes **collapse**'s time series and panel data architecture.

4.1. Ad-hoc computations

Time series functions such as `fgrowth()` (to compute growth rates) are S3 generic and can be applied to most time series classes. In addition to a `g` argument for grouped computation, these functions also have a `t` argument for indexation. If `t` is a plain numeric vector or a factor, it is coerced to integer and interpreted as time steps.[18] If `t` is a numeric time object (e.g., '`Date`', '`POSIXct`', etc.), then it is internally passed through `timeid()` which computes the greatest common divisor (GCD) and generates an integer time-id. For the GCD approach to work, `t` must have an appropriate class, e.g., for monthly/quarterly data, `zoo::yearmon()/zoo::yearqtr()` should be used instead of '`Date`' or '`POSIXct`'.

```
R> fgrowth(airmiles) |> round(2)
```

[16] Use `set_collapse(nthreads = #)` or the `nthreads` arguments to `fnth/fmedian/fmode` (default 1).

[17] `wlddev` is a dataset supplied by **collapse**, extracted from the World Bank World Development Indicators.

[18] This is premised on the observation that the most common form of temporal identifier is a numeric variable denoting calendar years. Users need to manually call `timeid()` on plain numeric vectors with decimals to yield an appropriate integer representation.

Classes, constructors and utilities
`findex_by()`, `findex()`, `unindex()`, `reindex()`, `timeid()`, `is_irregular()`, `to_plm()` + S3 methods for '`indexed_frame`', '`indexed_series`' and '`index_df`'
Core time-based functions
`flag()`, `fdiff()`, `fgrowth()`, `fcumsum()`, `psmat()` `psacf()`, `pspacf()`, `psccf()`
Data transformation functions with supporting methods
`fscale()`, `f[hd]between()`, `f[hd]within()`
Data manipulation functions with supporting methods
`fsubset()`, `funique()`, `roworder[v]()` (internal), `na_omit()` (internal)
Summary functions with supporting methods
`varying()`, `qsu()`

Table 2: Time series and panel data architecture.

```
Time Series:
Start = 1937
End = 1960
Frequency = 1
 [1]    NA 16.50 42.29 54.03 31.65  2.38 15.23 33.29 54.36 76.92  2.71 -2.10
[13] 12.91 18.51 32.03 18.57 17.82 13.61 18.19 12.83 13.32  0.01 15.49  4.25
```

The following code creates an irregular series by removing the 3rd and 15th observation and shows how indexation with the `t` argument accounts for this.

```
R> am_ir <- airmiles[-c(3, 15)]
R> t <- time(airmiles)[-c(3, 15)]
R> fgrowth(am_ir, t = t) |> round(2)

 [1]    NA 16.50    NA 31.65  2.38 15.23 33.29 54.36 76.92  2.71 -2.10 12.91
[13] 18.51    NA 17.82 13.61 18.19 12.83 13.32  0.01 15.49  4.25

R> fgrowth(am_ir, -1:3, t = t) |> head(4)

         FG1   --    G1  L2G1  L3G1
[1,] -14.167  412    NA    NA    NA
[2,]      NA  480 16.50    NA    NA
[3,] -24.043 1052    NA 119.2 155.3
[4,]  -2.327 1385 31.65    NA 188.5
```

For these functions, there also exists shorthands in the form of statistical operators, e.g., `L()/D()/G()` are shorthands for `flag()/fdiff()/fgrowth()`, which facilitate their use inside

formulas and also provide enhanced data frame interfaces for convenient ad-hoc computations. With panel data, `t` can be omitted, but this requires sorted data with consecutive groups.[19]

```
R> G(wlddev, c(1, 10), by = POP + LIFEEX ~ iso3c, t = ~ year) |> head(3)

  iso3c year G1.POP L10G1.POP G1.LIFEEX L10G1.LIFEEX
1   AFG 1960     NA        NA        NA           NA
2   AFG 1961  1.917        NA     1.590           NA
3   AFG 1962  1.985        NA     1.544           NA

R> settransform(wlddev, POP_growth = G(POP, g = iso3c, t = year))
```

These functions and operators are also integrated with `fsummarise()` and `fmutate()` for vectorized grouped computations.

```
R> wlddev |> fgroup_by(iso3c) |> fselect(iso3c, year, POP, LIFEEX) |>
+    fmutate(across(c(POP, LIFEEX), G, t = year)) |> head(2)

  iso3c year     POP LIFEEX G1.POP G1.LIFEEX
1   AFG 1960 8996973  32.45     NA        NA
2   AFG 1961 9169410  32.96  1.917      1.59
```

Similarly, functions to scale, center, and average data have groups (`g`) and also weights (`w`) arguments, and corresponding operators `STD(),[HD]W(),[HD]B()` to facilitate ad-hoc transformations. Below, two ways to perform grouped scaling are demonstrated. The operator version is slightly faster and renames the transformed columns by default (`stub = TRUE`).

```
R> iris |> fgroup_by(Species) |> fscale() |> head(2)

  Species Sepal.Length Sepal.Width Petal.Length Petal.Width
1  setosa       0.2667      0.1899       -0.357     -0.4365
2  setosa      -0.3007     -1.1291       -0.357     -0.4365

R> STD(iris, ~ Species) |> head(2)

  Species STD.Sepal.Length STD.Sepal.Width STD.Petal.Length STD.Petal.Width
1  setosa           0.2667          0.1899           -0.357         -0.4365
2  setosa          -0.3007         -1.1291           -0.357         -0.4365
```

The following example demonstrates a fixed-effects regression à la Mundlak (1978).

```
R> lm(mpg ~ carb + B(carb, cyl), data = mtcars) |> coef()

 (Intercept)         carb B(carb, cyl)
     34.8297      -0.4655      -4.7750
```

[19] This is because a group-lag is computed in a single pass, requiring all group elements to be consecutive.

collapse also offers higher-dimensional between and within transformations, powered by C++ code conditionally imported (and accessed directly) from **fixest**. The following detrends GDP per Capita and Life Expectancy at Birth using country-specific cubic polynomials.

```
R> HDW(wlddev, PCGDP + LIFEEX ~ iso3c * poly(year, 3), stub = F) |> head(2)

   PCGDP   LIFEEX
1  8.885 0.023614
2 13.685 0.006724
```

4.2. Indexed series and frames

For more complex use cases, indexation is very convenient. **collapse** supports **plm**'s 'pseries' and 'pdata.frame' classes through dedicated methods. Flexibility and performance considerations lead to the creation of new classes 'indexes_series' and 'indexed_frame' which inherit from the former. Any data frame-like object can be an 'indexed_frame' with any number of individual and/or time identifiers (e.g., an indexed 'data.table' is fully functional for other operations). The technical implementation of these classes is described in the vignette on object handling and, in more detail, in the documentation. The basic syntax is:

```
data_ix <- findex_by(data, id1, ..., time)
data_ix$indexed_series; with(data, indexed_series)
index_df <- findex(data_ix)
```

Data can be indexed using one or more indexing variables. Unlike 'pdata.frame', an 'indexed_frame' is a deeply indexed structure, i.e., every series inside the frame is already an 'indexes_series' and contains, in its 'index_df' attribute, an external pointer to the 'index_df' attribute of the frame (to avoid duplication in memory). A comprehensive set of methods for subsetting and manipulation, and applicable 'pseries' and 'pdata.frame' methods for time series and transformation functions like flag()/L(), ensure that these objects behave in a time-/panel-aware manor in any caller environment (created by with(), lm() etc.). Indexation can be undone using unindex() and redone with reindex() and a suitable 'index_df'. 'indexes_series' can be atomic vectors or matrices (including objects such as 'ts' or 'xts') and can also be created directly using reindex().

```
data <- unindex(data_ix)
data_ix <- reindex(data, index = index_df)
indexed_series <- reindex(vec/mat, index = vec/index_df)
```

An example using the wlddev data follows:

```
R> wldi <- wlddev |> findex_by(iso3c, year)
R> wldi |> fsubset(-3, iso3c, year, PCGDP:POP) |> G() |> head(4)

  iso3c year G1.PCGDP G1.LIFEEX G1.GINI G1.ODA G1.POP
1   AFG 1960       NA        NA      NA     NA     NA
2   AFG 1961       NA     1.590      NA  98.75  1.917
```

```
3   AFG 1963       NA         NA      NA     NA     NA
4   AFG 1964       NA      1.448      NA  24.48  2.112

Indexed by:  iso3c [1] | year [4 (61)]
```

The index statistics are: `[N. ids] | [N. periods (total periods: (max-min)/GCD)]`. This creates an '`indexes_series`' of life expectancy and demonstrates its properties:

```
R> LIFEEXi <- wldi$LIFEEX; str(LIFEEXi, width = 70, strict = "cut")

 'indexed_series' num [1:13176] 32.4 33 33.5 34 34.5 ...
 - attr(*, "label")= chr "Life expectancy at birth, total (years)"
 - attr(*, "index_df")=Classes 'index_df', 'pindex' and 'data.frame'..
  ..$ iso3c: Factor w/ 216 levels "ABW","AFG","AGO",..: 2 2 2 2 2 2 ..
  .. ..- attr(*, "label")= chr "Country Code"
  ..$ year : Ord.factor w/ 61 levels "1960"<"1961"<..: 1 2 3 4 5 6 7..
  .. ..- attr(*, "label")= chr "Year"

R> c(is_irregular(LIFEEXi), is_irregular(LIFEEXi[-5]))

[1] FALSE  TRUE

R> G(LIFEEXi[c(1:5, 7:10)])

[1]    NA 1.590 1.544 1.494 1.448    NA 1.366 1.362 1.365

Indexed by:  iso3c [1] | year [9 (61)]
```

The transformation and estimation below demonstrate the deep indexation of '`indexed_frame`"s, allowing correct computations in arbitrary data masking environments.

```
R> settransform(wldi, PCGDP_ld = Dlog(PCGDP))
R> lm(D(LIFEEX) ~ L(PCGDP_ld, 0:5) + B(PCGDP_ld), wldi) |>
+    summary() |> coef() |> round(3)

                   Estimate Std. Error t value Pr(>|t|)
(Intercept)           0.299      0.007  44.412    0.000
L(PCGDP_ld, 0:5)--    0.300      0.080   3.735    0.000
L(PCGDP_ld, 0:5)L1    0.269      0.081   3.332    0.001
L(PCGDP_ld, 0:5)L2    0.227      0.079   2.854    0.004
L(PCGDP_ld, 0:5)L3    0.200      0.078   2.563    0.010
L(PCGDP_ld, 0:5)L4    0.143      0.076   1.871    0.061
L(PCGDP_ld, 0:5)L5    0.095      0.073   1.301    0.193
B(PCGDP_ld)          -1.021      0.316  -3.234    0.001
```

The above example could also have been executed in one line as `lm(D(LIFEEX) ~ L(Dlog(PCGDP), 0:5) + B(Dlog(PCGDP)), wldi)`, log-differencing `PCGDP` twice.

In comparison with existing solutions, the flexibility of this architecture is new to the R ecosystem: A '`pdata.frame`' or '`fixest_panel`' only works inside **plm**/**fixest** estimation functions.[20] Time series classes like '`xts`' and '`tsibble`' also do not provide deeply indexed structures for time series operations or native handling of irregularity in basic operations. '`indexed_series`' and '`indexed_frame`', on the other hand, work anywhere and can be superimposed on any suitable object (such as '`sf`' to create a spatiotemporal panel), as long as **collapse**'s functions (`flag()`/`L()` etc.) are used to perform the time-based computations. The '`index_df`' attached to these objects can be used with other general tools such as `collapse::BY()` to perform grouped computations using 3rd-party functions. An example of calculating a 5-year rolling average is given below. Last but not least, the performance of these classes is second to none, as demonstrated in the useR 2022 presentation on slide 40.

```
R> BY(LIFEEXi, findex(LIFEEXi)$iso3c, data.table::frollmean, 5) |> head(10)
```

```
 [1]    NA    NA    NA    NA 33.46 33.96 34.46 34.95 35.43 35.92

Indexed by:  iso3c [1] | year [10 (61)]
```

5. Table joins and pivots

While **collapse** has a broad set of data manipulation functions, its implementations of table joins and pivots is particularly noteworthy since they offer several new features, including rich verbosity for table joins, pivots supporting variable labels, and 'recast' pivots. Both implementations provide outstanding computational performance and memory efficiency.

5.1. Joins

Compared to commercial environments such as STATA, the implementation of joins in most open-source software, including R, is non-verbose, i.e., provides no information on how many and which records were joined from both tables. This is somewhat unsatisfying and often provokes manual efforts to validate the join operation. `collapse::join` provides a rich set of options to make table join operations intelligible. Its syntax is:

```
join(x, y, on = NULL, how = "left", suffix = NULL, validate = "m:m",
  multiple = FALSE, sort = FALSE, keep.col.order = TRUE,
  drop.dup.cols = FALSE, verbose = 1, column = NULL, attr = NULL, ...)
```

By default (`verbose = 1`), it prints information about the join operation and number of records joined. Users can request the generation of a `.join` column (`column = "name"/TRUE`), akin to STATA's `_merge` column, indicating the origin of records in the joined table.

[20] And, in the case of **fixest**, inside **data.table** due to dedicated methods.

```
R> df1 <- data.frame(id1 = c(1, 1, 2, 3), id2 = c("a", "b", "b", "c"),
+    name = c("John", "Jane", "Bob", "Carl"), age = c(35, 28, 42, 50))
R> df2 <- data.frame(id1 = c(1, 2, 3, 3), id2 = c("a", "b", "c", "e"),
+    salary = c(60000, 55000, 70000, 80000),
+    dept = c("IT", "Marketing", "Sales", "IT"))
R> join(df1, df2, on = c("id1", "id2"), how = "full", column = TRUE)

full join: df1[id1, id2] 3/4 (75%) <m:m> df2[id1, id2] 3/4 (75%)
  id1 id2 name age salary      dept   .join
1   1   a John  35  60000        IT matched
2   1   b Jane  28     NA      <NA>     df1
3   2   b  Bob  42  55000 Marketing matched
4   3   c Carl  50  70000     Sales matched
5   3   e <NA>  NA  80000        IT     df2
```

An alternative to the join column is to request an attribute (`attr = "name"/TRUE`) that also summarizes the join operation, including the output of `fmatch()` (the workhorse of `join()` if `sort = FALSE`). Users can also invoke the `validate` argument to check the uniqueness of the join keys in either table: passing a '1' for a non-unique key produces an error.

```
R> join(df1, df2, on = c("id1", "id2"), validate = "1:1", attr = "join") |>
+    attr("join") |> str(width = 70, strict = "cut")

left join: df1[id1, id2] 3/4 (75%) <1:1> df2[id1, id2] 3/4 (75%)
List of 3
 $ call   : language join(x = df1, y = df2, on = c("id1", "id2"), v"..
 $ on.cols:List of 2
  ..$ x: chr [1:2] "id1" "id2"
  ..$ y: chr [1:2] "id1" "id2"
 $ match  : 'qG' int [1:4] 1 NA 2 3
  ..- attr(*, "N.nomatch")= int 1
  ..- attr(*, "N.groups")= int 4
  ..- attr(*, "N.distinct")= int 3
```

A few further particularities are worth highlighting. First, `collapse::join` is also class-agnostic and preserves the attributes of `x`. It supports 6 different join operations (`"left"`, `"right"`, `"inner"`, `"full"`, `"semi"`, or `"anti"`) and defaults to `"left"`, so the default behavior simply adds columns to `x`. By default (`sort = FALSE`), the order of rows in `x` is also preserved. Setting `sort = TRUE` sorts all records in the joined table by the keys.[21] Additionally, by default (`multiple = FALSE`), only the first matches from `y` are joined to avoid silent cartesian duplication of records. In multi-match settings, this will be reflected by few records from `y` being used. `fmatch()` also has a built-in overidentification check, which issues a warning if more key columns than necessary to identify the records are provided:

[21] This is done using a separate sort-merge-join algorithm, so it is faster than performing a hash join (using `fmatch()`) followed by sorting, particularly if the data is already sorted on the keys.

```
R> df2$name = df1$name
R> join(df1, df2) |> capture.output(type = "m") |>
+    strwrap(77) |> cat(sep = "\n")

left join: df1[id1, id2, name] 1/4 (25%) <m:m> df2[id1, id2, name] 1/4 (25%)
  id1 id2 name age salary dept
1   1   a John  35  60000   IT
2   1   b Jane  28     NA <NA>
3   2   b  Bob  42     NA <NA>
4   3   c Carl  50     NA <NA>
Warning in fmatch(x[ixon], y[iyon], nomatch = NA_integer_, count = count, :
Overidentified match/join: the first 2 of 3 columns uniquely match the
records. With overid > 0, fmatch() continues to match columns. Consider
removing columns or setting overid = 0 to terminate the algorithm after 2
columns (the results may differ, see ?fmatch). Alternatively set overid = 2
to silence this warning.
```

A final noteworthy feature is the handling of duplicate non-id columns in both tables:

```
R> join(df1, df2, on = c("id1", "id2"))

left join: df1[id1, id2] 3/4 (75%) <m:m> df2[id1, id2] 3/4 (75%)
duplicate columns: name => renamed using suffix '_df2' for y
  id1 id2 name age salary      dept name_df2
1   1   a John  35  60000        IT     John
2   1   b Jane  28     NA      <NA>     <NA>
3   2   b  Bob  42  55000 Marketing     Jane
4   3   c Carl  50  70000     Sales      Bob
```

By default (`suffix = NULL`), `join()` extracts the name of the `y` table and appends `y`-columns with it. `x`-columns are not renamed. This is congruent to the principle of adding columns to `x` and altering this table as little as possible. Another option, `drop.dup.cols = "x"/"y"`, can be used to simply drop duplicate columns from `x` or `y` before the join operation.

5.2. Pivots

The reshaping/pivoting functionality of both commercial and open source software is also unsatisfying for complex datasets such as surveys or disaggregated production, trade, or financial sector data, where variable names resemble codes and variable labels are essential to making sense of the data. Such datasets can presently only be reshaped by losing these labels or additional manual efforts to retain them. Modern R packages also offer different functions for different reshaping operations, such as `data.table::melt/tidyr::pivot_longer` to combine columns and `data.table::dcast/tidyr::pivot_wider` to expand them, requiring users to learn both. Since the depreciation of **reshape(2)** (Wickham 2007), there is also no modern replacement for `reshape2::recast()`, requiring R users to consecutively call two reshaping functions, incurring a high cost in terms of syntax and memory.

`collapse::pivot` provides a modern class-agnostic implementation of reshaping for R that addresses these shortcomings: it has a single intuitive syntax to perform 'longer', 'wider', and 'recast' pivots, and supports complex labelled data without loss of information. Its syntax is:

```
pivot(data, ids = NULL, values = NULL, names = NULL, labels = NULL,
  how = "longer", na.rm = FALSE, factor = c("names", "labels"),
  check.dups = FALSE, nthreads = 1, fill = NULL, drop = TRUE,
  sort = FALSE, transpose = FALSE)
```

The demonstration below employs a generated dataset about fruits. We could equivalently think about a survey with households and individuals, or sectors and firms. Variable labels are stored in `attr(column, "label")`. The documentation provides more elaborate examples.

```
R> data <- data.frame(type = rep(c("A", "B"), each = 2),
+    type_name = rep(c("Apples", "Bananas"), each = 2),
+    id = rep(1:2, 2), r = abs(rnorm(4)), h = abs(rnorm(4)*2))
R> setrelabel(data, id = "Fruit Id", r = "Fruit Radius", h = "Fruit Height")
R> print(data)

  type type_name id      r     h
1    A    Apples  1 0.3139 2.983
2    A    Apples  2 0.4095 1.572
3    B   Bananas  1 0.4302 2.220
4    B   Bananas  2 0.1851 2.380

R> vlabels(data)

          type      type_name             id              r              h
            NA             NA     "Fruit Id" "Fruit Radius" "Fruit Height"
```

To reshape this dataset into a longer format, it suffices to call `pivot(data, ids = c(...))`. If `labels = "lab_name"` is specified, variable labels are saved to an additional column named `lab_name`. In addition, `names = list(variable = "var_name", value = "val_name")` can be passed to assign alternative names to the `variable` and `value` columns, respectively.

```
R> (dl <- pivot(data, ids = c("type", "type_name", "id"), labels = "label"))

  type type_name id variable        label  value
1    A    Apples  1        r Fruit Radius 0.3139
2    A    Apples  2        r Fruit Radius 0.4095
3    B   Bananas  1        r Fruit Radius 0.4302
4    B   Bananas  2        r Fruit Radius 0.1851
5    A    Apples  1        h Fruit Height 2.9829
6    A    Apples  2        h Fruit Height 1.5722
7    B   Bananas  1        h Fruit Height 2.2203
8    B   Bananas  2        h Fruit Height 2.3797
```

```
R> vlabels(dl)

      type  type_name         id   variable      label      value
        NA         NA "Fruit Id"         NA         NA         NA
```

`pivot()` only requires essential information and intelligently guesses the rest. For example, the same result could have been obtained by `pivot(data, values = c("r", "h"), labels = "label")`. An exact reverse operation can also be performed by specifying as little as `pivot(dl, labels = "label", how = "w")`.

The second option is a wider pivot with `how = "wider"`. Here, `names` and `labels` can be used to select columns containing the names of new columns and their labels.[22] Note how the labels are combined with existing labels such that also this operation is without loss of information. It is, however, a destructive operation, i.e., with 2 or more columns selected through `values`, `pivot()` is not able to reverse it. Further arguments like `na.rm`, `fill`, `sort`, and `transpose` can be used to control the casting process.

```
R> (dw <- pivot(data, "id", names = "type", labels = "type_name", how = "w"))

  id    r_A    r_B   h_A  h_B
1  1 0.3139 0.4302 2.983 2.22
2  2 0.4095 0.1851 1.572 2.38

R> namlab(dw)

  Variable                  Label
1       id               Fruit Id
2      r_A  Fruit Radius - Apples
3      r_B Fruit Radius - Bananas
4      h_A  Fruit Height - Apples
5      h_B Fruit Height - Bananas
```

For the recast pivot (`how = "recast"`), unless a column named `variable` exists in the data, the source and (optionally) destination of variable names need to be specified using a list passed to `names`, and similarly for `labels`. Again, taking along labels is optional, and omitting either the list's `from` or `to` elements will omit the respective operations.

```
R> (dr <- pivot(data, ids = "id", names = list(from = "type"),
+               labels = list(from = "type_name", to = "label"), how = "r"))

  id variable        label      A      B
1  1        r Fruit Radius 0.3139 0.4302
2  2        r Fruit Radius 0.4095 0.1851
3  1        h Fruit Height 2.9829 2.2203
4  2        h Fruit Height 1.5722 2.3797
```

[22] multiple columns with names and labels could be selected, which would be combined using `"_"` for names and `" - "` for labels.

```
R> vlabels(dr)
```

```
        id   variable      label          A          B
"Fruit Id"         NA         NA   "Apples"  "Bananas"
```

As with the other pivots, this operation does not incur any loss of information. A suitable reverse operation also exists: `pivot(dr, "id", names = list(to = "type"), labels = list(from = "label", to = "type_name"), how = "r")`. More features of `pivot()` are demonstrated in the documentation examples. Notably, it is also possible to perform longer and recast pivots without id variables. The recast pivot without ids resembles a generalization of `data.table::transpose()`, albeit slightly less efficient.

6. List processing

Often in programming, nested structures are needed. A typical use case involves running statistical procedures for multiple configurations of variables and parameters and saving multiple objects (such as a model object, performance statistics, and predictions) in a list. Nested data is also often the result of web scraping or web APIs. A typical use case in development involves serving different data according to user choices, e.g., in response to nested user inputs in shiny apps. Except for certain recursive functions found in packages such as **purr**, **tidyr**, or **rrapply**, R lacks a general recursive toolkit to create, query, and tidy nested data. **collapse**'s list processing functions attempt to provide a basic toolkit.

To create nested data, `rsplit()` generalizes `split()` and (recursively) splits up data frame-like objects into a (nested) list.

```
R> (dl <- mtcars |> rsplit(mpg + hp + carb ~ vs + am)) |> str(max.level = 2)
```

```
List of 2
 $ 0:List of 2
  ..$ 0:'data.frame':        12 obs. of  3 variables:
  ..$ 1:'data.frame':        6 obs. of  3 variables:
 $ 1:List of 2
  ..$ 0:'data.frame':        7 obs. of  3 variables:
  ..$ 1:'data.frame':        7 obs. of  3 variables:
```

If a nested structure is not wanted, argument `flatten = TRUE` lets `rsplit()` operate like a faster version of `split()`. With a single column on the LHS of the formula, the default (`simplify = TRUE`) returns a nested list of atomic vectors. Having created a nested list, `rapply2d()` is used to fit a linear model on each frame,[23] followed by `get_elem()` to obtain the coefficient matrices. `get_elem()` offers several options for filtering lists but, by default, simplifies the list tree as much as possible while maintaining existing hierarchies. In this case, it returns the same nested list with coefficient matrices in all final nodes.

[23] `rapply2d()` is just a recursive wrapper around `lapply()`, with different defaults than `rapply()`. Notably, by default, it excludes data frames from being considered as sub-lists and does not simplify the result.

```
R> nest_lm_coef <- dl |> rapply2d(lm, formula = mpg ~ .) |>
+    rapply2d(summary, classes = "lm") |> get_elem("coefficients")
R> nest_lm_coef |> str(give.attr = FALSE, strict = "cut")

List of 2
 $ 0:List of 2
  ..$ 0: num [1:3, 1:4] 15.8791 0.0683 -4.5715 3.655 0.0345 ...
  ..$ 1: num [1:3, 1:4] 26.9556 -0.0319 -0.308 2.293 0.0149 ...
 $ 1:List of 2
  ..$ 0: num [1:3, 1:4] 30.896903 -0.099403 -0.000332 3.346033 0.03587 ...
  ..$ 1: num [1:3, 1:4] 37.0012 -0.1155 0.4762 7.3316 0.0894 ...
```

At last, `unlist2d()` is applied to unlist the nested list to a data frame. This function can create a data frame (or '`data.table`') representation of any nested list containing data using recursive row-binding and coercion operations while generating (optional) id variables representing the list tree and (optionally) saving row names of matrices or data frames.

```
R> nest_lm_coef |> unlist2d(c("vs", "am"), row.names = "variable") |> head(2)

  vs am    variable Estimate Std. Error t value Pr(>|t|)
1  0  0 (Intercept) 15.87915    3.65495   4.345 0.001865
2  0  0          hp  0.06832    0.03449   1.981 0.078938
```

This example does not represent an optimal workflow for this specific task[24] but exemplifies the power of these tools to create, query, and combine nested data in very general ways. **collapse**'s list processing toolkit provides further useful functions such as `t_list()` to turn lists of lists inside out, `has_elem()` to check for the existence of elements, `ldepth()` to return the maximum level of recursion, and `is_unlistable()` to check whether a list has atomic elements in all final nodes. A non-recursive and class-agnostic `rowbind()` function also exists to efficiently bind lists of data frame-like objects (like `data.table::rbindlist()`).

7. Summary statistics

collapse's summary statistics functions offer a parsimonious and powerful toolset to examine complex datasets. A particular focus has been on providing tools for examining longitudinal (panel) data. Recall the indexed world development panel (`wldi`) from Section 4. The function `varying()` indicates which of these variables are time-varying:

```
R> varying(wldi)

country    date    year  decade  region  income    OECD   PCGDP  LIFEEX
  FALSE    TRUE    TRUE    TRUE   FALSE   FALSE   FALSE    TRUE    TRUE
   GINI     ODA     POP
   TRUE    TRUE    TRUE
```

[24] A better way of achieving the same result would be `mtcars |> fgroup_by(vs, am) |> fsummarise(qDF(lmtest::coeftest(lm(mpg  hp + carb)), "variable"))`.

```
R> varying(wldi, any_group = FALSE) |> head(3)
```

```
    country date year decade region income  OECD PCGDP LIFEEX GINI  ODA  POP
ABW   FALSE TRUE TRUE   TRUE  FALSE  FALSE FALSE  TRUE   TRUE   NA TRUE TRUE
AFG   FALSE TRUE TRUE   TRUE  FALSE  FALSE FALSE  TRUE   TRUE   NA TRUE TRUE
AGO   FALSE TRUE TRUE   TRUE  FALSE  FALSE FALSE  TRUE   TRUE TRUE TRUE TRUE
```

Country-variance can be examined using `varying(wldi, effect = "year")`. For non-indexed data, `varying()` also has a `g` argument. A related exercise is to decompose the variance of a panel series into a component due to variation between countries and one capturing variance within countries over time. Using the `W()/B()` operators and the `LIFEEXi` '`indexed_series`' from Section 4, this is easily demonstrated:

```
R> all.equal(fvar(W(LIFEEXi)) + fvar(B(LIFEEXi)), fvar(LIFEEXi))
```

```
[1] TRUE
```

The function `qsu()` (quick-summary) provides an efficient method to approximately compute this decomposition, considering the group-means instead of the between transformation[25] and adding the mean back to the within transformation to preserve the scale of the data.

```
R> qsu(LIFEEXi)
```

```
             N/T     Mean       SD      Min      Max
Overall    11670  64.2963  11.4764   18.907  85.4171
Between      207  64.9537   9.8936  40.9663  85.4171
Within   56.3768  64.2963   6.0842  32.9068  84.4198
```

This decomposition shows more variation in life expectancy between countries than within countries over time. It can also be computed for different subgroups, such as OECD members and non-members, and with sampling weights, such as population. `qsu()` can also return Pearson's measures of higher-order statistics.

```
R> qsu(LIFEEXi, g = wlddev$OECD, w = wlddev$POP, higher = TRUE) |> aperm()
```

```
, , FALSE

             N/T     Mean      SD      Min      Max     Skew    Kurt
Overall     9503  63.5476  9.2368   18.907  85.4171  -0.7394  2.7961
Between      171  63.5476  6.0788  43.0905  85.4171  -0.8041   3.082
Within   55.5731  65.8807  6.9545  30.3388  82.8832  -1.0323  4.0998

, , TRUE

             N/T     Mean      SD      Min      Max     Skew    Kurt
Overall     2156  74.9749  5.3627   45.369  84.3563  -1.2966  6.5505
Between       36  74.9749  2.9256  66.2983  78.6733  -1.3534  4.5999
Within   59.8889  65.8807  4.4944  44.9513  77.2733   -0.627  3.9839
```

[25] This is more efficient and equal to using the between transformation if the panel is balanced.

The output shows that the variation in life expectancy is significantly larger for non-OECD countries and that for these countries, the between- and within-country variation is approximately equal in magnitude.[26] For more detailed (grouped, weighted) statistics, `descr()` provides a rich statistical description of variables in a dataset.

```
R> descr(wlddev, LIFEEX ~ OECD, w = ~ replace_na(POP))

Dataset: wlddev, 1 Variables, N = 13176, WeightSum = 313233706778
Grouped by: OECD [2]
           N   Perc       WeightSum  Perc
FALSE  10980  83.33  2.49344474e+11  79.6
TRUE    2196  16.67  6.38892329e+10  20.4
-------------------------------------------------------------------------------
LIFEEX (numeric): Life expectancy at birth, total (years)
Statistics (N = 11659, 11.51% NAs)
          N   Perc  Ndist   Mean    SD    Min    Max   Skew  Kurt
FALSE  9503  81.51   8665  63.55  9.24  18.91  85.42  -0.74   2.8
TRUE   2156  18.49   2016  74.97  5.36  45.37  84.36   -1.3  6.55

Quantiles
          1%     5%    10%    25%    50%    75%    90%    95%    99%
FALSE  41.39  45.78  49.08  57.51  65.98  70.14  74.12  75.63  76.91
TRUE   56.65  65.98   69.7  71.85  75.38  78.64  81.26  82.43   83.6
-------------------------------------------------------------------------------
```

While `descr()` does not support panel-variance decompositions like `qsu()`, it also computes detailed (grouped, weighted) frequency tables for categorical data and is thus very utile with complex surveys. A `stepwise` argument toggles describing one variable at a time, allowing users to naturally 'click-through' a large dataset rather than printing a massive output to the console. More details and examples are in the documentation. Both `qsu()` and `descr()` provide an `as.data.frame()` method for efficient tidying and further analysis.

A final noteworthy function from **collapse**'s descriptive statistics toolkit is `qtab()`, an enhanced drop-in replacement for `base::table`. It is enhanced both in a statistical and computational sense, providing a remarkable performance boost, an option (`sort = FALSE`) to preserve the first-appearance-order of vectors being cross-tabulated, support for frequency weights (`w`), and the ability to compute different statistics representing table entries using these weights - vectorized when using *Fast Statistical Functions*, as demonstrated below.

```
R> library("magrittr")
R> wlda15 <- wlddev |> fsubset(year >= 2015) |> fgroup_by(iso3c) |> flast()
R> wlda15 %$% qtab(OECD, income)

       income
OECD    High income Low income Lower middle income Upper middle income
```

[26] `qsu()` also has a convenient formula interface to perform these transformations in an ad-hoc fashion, e.g., the above can be obtained using `qsu(wlddev, LIFEEX  OECD,  iso3c,  POP, higher = TRUE)`, without prior indexation.

```
  FALSE          45          30                  47                  58
  TRUE           34           0                   0                   2
```

This shows the total population (latest post-2015 estimates) in millions.

```
R> wlda15 %$% qtab(OECD, income, w = POP) %>% divide_by(1e6)

       income
OECD    High income Low income Lower middle income Upper middle income
  FALSE       93.01     694.89             3063.54             2459.71
  TRUE      1098.75       0.00                0.00              211.01
```

This shows the average life expectancy in years. The use of `fmean()` toggles an efficient vectorized computation of the table entries (i.e., `fmean()` is only called once).

```
R> wlda15 %$% qtab(OECD, income, w = LIFEEX, wFUN = fmean) %>% replace_na(0)

       income
OECD    High income Low income Lower middle income Upper middle income
  FALSE       78.75      62.81               68.30               73.81
  TRUE        81.09       0.00                0.00               76.37
```

Finally, this calculates a population-weighted average of life expectancy in each group.

```
R> wlda15 %$% qtab(OECD, income, w = LIFEEX, wFUN = fmean,
+                  wFUN.args = list(w = POP)) %>% replace_na(0)

       income
OECD    High income Low income Lower middle income Upper middle income
  FALSE       77.91      63.81               68.76               75.93
  TRUE        81.13       0.00                0.00               76.10
```

'`qtab`' objects inherit the '`table`' class, thus all '`table`' methods apply. Apart from the above functions, **collapse** also provides functions **pwcor**, **pwcov**, **pwnobs** for convenient (pairwise, weighted) correlations, covariances, and observations counts, and also functions **psacf**, **pspacf** and **psccf** for auto- and cross-covariance and correlation function estimation on panel series.

8. Global options

collapse is globally configurable to an extent few packages are: the default value of key function arguments governing the behavior of its algorithms, and the exported namespace, can be adjusted interactively through the `set_collapse()` function. These options are saved in an internal environment called `.op` (for safety and performance reasons) visible in the documentation of some functions. Its contents can be accessed using `get_collapse()`.

The current set of options comprises the default behavior for missing values (`na.rm` arguments in all statistical functions and algorithms), sorted grouping (`sort`), multithreading and algorithmic optimizations (`nthreads`, `stable.algo`), presentational settings (`stub`, `digits`, `verbose`), and, surpassing all else, the package namespace itself (`mask`, `remove`).

As evident from previous sections, **collapse** provides performance-improved or otherwise enhanced versions of functionality already present in base R (like the *Fast Statistical Functions*, `funique()`, `fmatch()`, `fsubset()`, `ftransform()`, etc.) and other packages (esp. **dplyr** (Wickham *et al.* 2023a): `fselect()`, `fsummarise()`, `fmutate()`, `frename()`, etc.). The objective of being namespace compatible warrants such a naming convention, but this has a syntactical cost, particularly when **collapse** is the primary data manipulation package.

To reduce this cost, **collapse**'s `mask` option allows masking existing R functions with the faster **collapse** versions by creating additional functions in the namespace and instantly exporting them. All **collapse** functions starting with 'f' can be passed to the option (with or without the 'f'), e.g., `set_collapse(mask = c("subset", "transform"))` creates `subset <- fsubset` and `transform <- ftransform` and exports them. Special functions are `"n"`, `"table"/"qtab"`, and `"%in%"`, which create `n <- GRPN` (for use in `(f)summarise`/`(f)mutate`), `table <- qtab`, and replace `%in%` with a fast version using `fmatch`, respectively. There are also several convenience keywords to mask related groups of functions. The most powerful of these is `"all"`, which masks all f-functions + specials, as shown below.

```
set_collapse(mask = "all", na.rm = FALSE, sort = FALSE, nthreads = 4)
wlddev |> subset(year >= 1990 & is.finite(GINI)) |>
  group_by(year) |>
  summarise(n = n(), across(PCGDP:GINI, mean, w = POP))
with(mtcars, table(cyl, vs, am))
sum(mtcars)
diff(EuStockMarkets)
mean(num_vars(iris), g = iris$Species)
unique(wlddev, cols = c("iso3c", "year"))
range(wlddev$date)
wlddev |> index_by(iso3c, year) |>
  mutate(PCGDP_lag = lag(PCGDP),
         PCGDP_diff = PCGDP - PCGDP_lag,
         PCGDP_growth = growth(PCGDP)) |> unindex()
```

The above is now 100% **collapse** code. Similarly, using this option, all code in this article could have been written without f-prefixes. Thus, **collapse**, together with namespace masking, is able to provide a fast and syntactically clean experience of R - without the need to even restart the session. Masking is completely interactive and reversible within the active session: calling `set_collapse(mask = NULL)` instantly removes the additional functions. Option `remove` can further be used to remove any **collapse** function from the list of exported functions, allowing manual conflict management. Function `fastverse::fastverse_conflicts()` from the related **fastverse** project[27] can be used to display namespace conflicts with **collapse**. Invoking

[27] Website: https://fastverse.github.io/fastverse/

either `mask` or `remove` detaches **collapse** and reattaches it at the top of the search path, letting its namespace to take precedence over other packages.

9. Benchmark

This section offers a small benchmark to demonstrate that **collapse** provides best-in-R performance for many basic statistical and data manipulation tasks. They are executed on an Apple M1 MacBook Pro with 16 GB unified memory. The DuckDB Benchmarks compare more software packages on larger datasets, using a large server with many (slow) cores.[28]

```
R> setDTthreads(4)
R> set_collapse(na.rm = FALSE, sort = FALSE, nthreads = 4)
R> set.seed(101)
R> m <- matrix(rnorm(1e7), ncol = 1000)
R> data <- qDT(replicate(100, rnorm(1e5), simplify = FALSE))
R> g <- sample.int(1e4, 1e5, TRUE)
R> microbenchmark(R = colMeans(m),
+                 Rfast = Rfast::colmeans(m, parallel = TRUE, cores = 4),
+                 collapse = fmean(m))
```

```
Unit: milliseconds
     expr   min    lq   mean median    uq    max neval
        R 9.827 9.858 10.061  9.885 9.970 16.701   100
    Rfast 1.349 1.819  2.135  1.877 2.009 11.236   100
 collapse 1.321 1.475  1.728  1.521 1.701  5.502   100
```

[28] A lot may be said about benchmarking **collapse**, which would be beyond the scope of this article. Users should note, however, that its defaults (`na.rm = TRUE, sort = TRUE, stable.algo = TRUE, nthreads = 1`) cater to convenience rather than maximum performance. For maximum performance, set these 3 settings to `FALSE` and increase the number of threads. To also provide a minimalistic guide for R users seeking to understand the relative performance of **collapse** and **data.table**, reflecting current (spring 2024) developments: **collapse** has highly efficient algorithms for grouping and computing statistics, but presently does not provide sub-column level parallel grouping architecture. Simple statistics like `fmean()` are parallelized across columns and perform grouped computations in a single pass. More complex ones `fmedian(), fmode()` have group-level parallelism. **data.table**, on the other hand, has sub-column parallel grouping and also group-level parallel implementations for simple statistics such as `mean()`, but no parallelism for complex statistics such as `median()`. **data.table**'s GForce optimization also only applies to simple statistics, not complex expressions or weighted statistics - as can be vectorized using *Fast Statistical Functions* in **collapse**. Thus, if your data is moderately sized ($\leq$100mio. obs.), you have more than 1 column to compute on, you want to do complex statistical things, or if your processor is very fast (high single core speed), **collapse** is a great choice. On the other hand, if your data is really long (>100mio. obs.), you have only a few columns to compute on, you are computing simple statistics that **data.table** optimizes, and you have massive parallel compute, then **data.table** is a great choice. My recommendation: use both, just need to call `library(fastverse)`. Finally, let me note that **polars** uses optimized memory buffers based on Apache Arrow, multithreaded hash-based grouping, SIMD instructions and multithreading at the group-level, and a query optimizer - all implemented in Rust, a thread-safe programming language. While some of these parallel algorithms could be ported to **collapse**, this is more challenging since C, and particularly R's C API, is not thread safe - and it would still be lacking the benefits of Arrow memory buffers. At core, R is a 30-year old statistical language and not intended to work like an optimized database. **collapse** seamlessly integrates with R's data structures; **polars**, at present, has nothing to do with them (and is therefore also not part of this benchmark).

```
R> microbenchmark(R = rowsum(data, g, reorder = FALSE),
+                 data.table = data[, lapply(.SD, sum), by = g],
+                 collapse = fsum(data, g))

Unit: milliseconds
       expr    min     lq   mean median     uq    max neval
          R 11.002 11.331 12.255 11.516 11.946 24.269   100
 data.table 13.918 19.036 21.956 19.746 21.676 64.273   100
   collapse  1.911  2.743  3.408  3.136  3.702  6.514   100

R> add_vars(data) <- g
R> microbenchmark(data.table = data[, lapply(.SD, median), by = g],
+                 collapse = data |> fgroup_by(g) |> fmedian())

Unit: milliseconds
       expr    min     lq   mean median     uq   max neval
 data.table 136.94 137.72 141.04 138.48 140.01 265.1   100
   collapse  70.48  76.56  81.03  79.72  82.17 191.9   100

R> d <- data.table(g = unique(g), x = 1, y = 2, z = 3)
R> microbenchmark(data.table = d[data, on = "g"],
+                 collapse = join(data, d, on = "g", verbose = 0))

Unit: milliseconds
       expr   min     lq   mean median     uq    max neval
 data.table 9.153 12.140 24.382 14.118 20.849 66.701   100
   collapse 1.220  1.391  1.472  1.424  1.466  2.948   100

R> microbenchmark(data.table = melt(data, "g"),
+                 collapse = pivot(data, "g"))

Unit: milliseconds
       expr   min    lq mean median    uq   max neval
 data.table 11.96 15.34 22.6  16.63 18.41 75.20   100
   collapse 11.74 15.01 22.0  16.32 18.28 63.01   100

R> settransform(data, id = rowid(g))
R> cols = grep("^V", names(data), value = TRUE)
R> microbenchmark(data.table = dcast(data, g ~ id, value.var = cols),
+            collapse = pivot(data, ids = "g", names = "id", how = "w"))

Unit: milliseconds
       expr   min     lq   mean median     uq   max neval
 data.table 72.48 116.74 119.65  119.7 123.68 142.7   100
   collapse 55.83  87.97  92.83   90.6  93.19 200.7   100
```

The benchmark below further shows that **collapse** provides faster algorithms for basic computationally intensive operations such as unique values and matching. These algorithms power much of its functionality, such as efficient factor generation with `qF()`, cross-tabulation with `qtab()`, `join()`'s, `pivot()`'s, etc.

```
R> set.seed(101)
R> g_int <- sample.int(1e3, 1e7, replace = TRUE)
R> char <- c(letters, LETTERS, month.abb, month.name)
R> char <- outer(char, char, paste0)
R> g_char <- sample(char, 1e7, replace = TRUE)
R> microbenchmark(base_int = unique(g_int), collapse_int = funique(g_int),
+              base_char = unique(g_char), collapse_char = funique(g_char))

Unit: milliseconds
          expr    min     lq  mean median     uq    max neval
      base_int 60.294 63.508 65.73 65.251  66.62 110.97   100
  collapse_int  8.677  9.347 10.49  9.595  10.91  15.83   100
     base_char 92.569 95.032 99.15 98.483 100.67 141.98   100
 collapse_char 21.756 23.128 24.91 23.691  24.46  88.37   100

R> microbenchmark(base_int = match(g_int, 1:1000),
+                 collapse_int = fmatch(g_int, 1:1000),
+                 base_char = match(g_char, char),
+                 data.table_char = chmatch(g_char, char),
+                 collapse_char = fmatch(g_char, char), times = 10)

Unit: milliseconds
            expr   min     lq   mean median      uq    max neval
        base_int 27.24 27.502 29.085 28.093  30.789  32.41    10
    collapse_int  8.78  8.867  9.429  8.891   9.185  13.29    10
       base_char 94.72 96.474 99.857 97.048 101.644 110.06    10
 data.table_char 42.21 42.412 44.063 42.758  43.287  51.68    10
   collapse_char 36.61 36.769 37.592 36.959  37.667  41.62    10
```

Apart from the raw algorithmic efficiency demonstrated here, **collapse** is often more efficient than other solutions by simply doing less. For example, if grouping columns are factor variables, **collapse**'s algorithms in `funique()`, `group()` or `fmatch()`, etc., use the values as hashes without checking for collisions. Similarly, if data is already sorted/unique, it is directly returned by functions like `roworder()`/`funique()`.

10. Conclusion

It is coming close to 4 years since the first CRAN release of **collapse** in March 2020, and since then, the package has grown and matured considerably. At the time of writing this article in early 2024, it has been downloaded >1.5 million times off CRAN. In this article, I have articulated key ideas and design principles and demonstrated some core features of the package. In summary, my work with R as an applied economist has led me to believe that there should be a new foundation package for statistical computing and data manipulation in R that is statistically advanced, class-agnostic, flexible, fast, lightweight, stable, and able to manipulate complex scientific data with ease. **collapse** is my attempt at providing such a package, and the feedback I have received over the years, particularly from users in academia, government, and international organizations, is a strong indication that I have responded to a need felt in larger parts of the R community. As mentioned, a single article cannot comprehensively introduce **collapse**, but there is a modern website with comprehensive documentation resources.

Computational details

The results in this paper were obtained using R (R Core Team 2023) 4.3.0 with **collapse** 2.0.10, **data.table** 1.15.0, **Rfast** 2.1.0, **fixest** 0.11.3, **magrittr** (Bache and Wickham 2022) 2.0.3 and **microbenchmark** (Mersmann 2023) 1.4.10. All packages used are available from the Comprehensive R Archive Network (CRAN) at https://CRAN.R-project.org/. The benchmark was run on an Apple M1 MacBook Pro (2020) with 16GB unified memory. Packages were compiled from source using Homebrew Clang version 16.0.4 with OpenMP enabled and the -O3 optimization flag.

Acknowledgments

The source code of **collapse** has been heavily inspired by (and partly copied from) **data.table** (Matt Dowle and Arun Srinivasan), R's source code (R Core Team and contributors worldwide), the **kit** package (Morgan Jacob), and **Rcpp** (Dirk Eddelbuettel). Packages **plm** (Yves Croissant, Giovanni Millo, and Kevin Tappe) and **fixest** (Laurent Berge) have also provided a lot of inspiration (and a port to its demeaning algorithm in the case of **fixest**). I also thank many people from diverse fields for helpful answers on Stackoverflow and many other people for encouragement, feature requests, and helpful issues and suggestions.

References

Bache SM, Wickham H (2022). ***magrittr**: A Forward-Pipe Operator for R*. R package version 2.0.3, URL https://CRAN.R-project.org/package=magrittr.

Bengtsson H (2023). ***matrixStats**: Functions that Apply to Rows and Columns of Matrices (and to Vectors)*. R package version 1.0.0, URL https://CRAN.R-project.org/package=matrixStats.

Bergé L (2018). "Efficient Estimation of Maximum Likelihood Models with Multiple Fixed-Effects: the R Package **FENmlm**." *CREA Discussion Papers*, (13).

Bouchet-Valat M, Kamiński B (2023). "**DataFrames.jl**: Flexible and Fast Tabular Data in Julia." *Journal of Statistical Software*, **107**(4), 1–32. doi:10.18637/jss.v107.i04. URL https://www.jstatsoft.org/index.php/jss/article/view/v107i04.

Chau J (2022). ***rrapply**: Revisiting Base Rapply*. R package version 1.2.6, URL https://CRAN.R-project.org/package=rrapply.

Croissant Y, Millo G (2008). "Panel Data Econometrics in R: The **plm** Package." *Journal of Statistical Software*, **27**(2), 1–43. doi:10.18637/jss.v027.i02.

Dowle M, Srinivasan A (2023). ***data.table**: Extension of 'data.frame'*. R package version 1.14.8, URL https://CRAN.R-project.org/package=data.table.

Harris CR, *et al.* (2020). "Array programming with **NumPy**." *Nature*, **585**, 357–362. doi:10.1038/s41586-020-2649-2.

Hyndman RJ, Fan Y (1996). "Sample Quantiles in Statistical Packages." *American Statistician*, pp. 361–365.

JuliaStats (2023). "**StatsBase.jl**: Julia Package for Basic Statistics." URL https://github.com/JuliaStats/StatsBase.jl.

Larmarange J (2023). ***labelled**: Manipulating Labelled Data*. R package version 2.12.0, URL https://CRAN.R-project.org/package=labelled.

Lumley T (2004). "Analysis of Complex Survey Samples." *Journal of Statistical Software*, **9**(1), 1–19. R package verson 2.2.

Mersmann O (2023). ***microbenchmark**: Accurate Timing Functions*. R package version 1.4.10, URL https://CRAN.R-project.org/package=microbenchmark.

Mundlak Y (1978). "On the Pooling of Time Series and Cross Section Data." *Econometrica: Journal of the Econometric Society*, pp. 69–85.

Müller K, Wickham H (2023). ***tibble**: Simple Data Frames*. R package version 3.2.1, URL https://CRAN.R-project.org/package=tibble.

pandas Development Team (2023). "pandas-dev/pandas: **pandas**." doi:10.5281/zenodo.10426137. URL https://doi.org/10.5281/zenodo.10426137.

Papadakis M, *et al.* (2023). ***Rfast**: A Collection of Efficient and Extremely Fast R Functions*. R package version 2.1.0, URL https://CRAN.R-project.org/package=Rfast.

Pebesma E (2018). "Simple Features for R: Standardized Support for Spatial Vector Data." *The R Journal*, **10**(1), 439–446. doi:10.32614/RJ-2018-009. URL https://doi.org/10.32614/RJ-2018-009.

R Core Team (2023). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Ryan JA, Ulrich JM (2023). ***xts**: eXtensible Time Series*. R package version 0.13.1, URL https://CRAN.R-project.org/package=xts.

Signorell A (2023). ***DescTools**: Tools for Descriptive Statistics*. R package version 0.99.52, URL https://CRAN.R-project.org/package=DescTools.

StataCorp LLC (2023). *STATA Statistical Software: Release 18*. College Station, TX. URL https://www.stata.com.

Vink R, *et al.* (2023). "pola-rs/polars: Python **polars** 0.20.2." doi:10.5281/zenodo.10413093. URL https://doi.org/10.5281/zenodo.10413093.

Wang E, *et al.* (2020). "A New Tidy Data Structure to Support Exploration and Modeling of Temporal Data." *Journal of Computational and Graphical Statistics*, **29**(3), 466–478. doi:10.1080/10618600.2019.1695624. URL https://doi.org/10.1080/10618600.2019.1695624.

Wes McKinney (2010). "Data Structures for Statistical Computing in Python." In Stéfan van der Walt, Jarrod Millman (eds.), *Proceedings of the 9th Python in Science Conference*, pp. 56 – 61. doi:10.25080/Majora-92bf1922-00a.

Wickham H (2007). "Reshaping Data with the **reshape** Package." *Journal of Statistical Software*, **21**(12), 1–20. URL http://www.jstatsoft.org/v21/i12/.

Wickham H (2014). "Tidy Data." *Journal of Statistical Software*, **59**(10), 1–23. doi:10.18637/jss.v059.i10. URL https://www.jstatsoft.org/index.php/jss/article/view/v059i10.

Wickham H, Henry L (2023). ***purrr**: Functional Programming Tools*. R package version 1.0.1, URL https://CRAN.R-project.org/package=purrr.

Wickham H, *et al.* (2019). "Welcome to the **tidyverse**." *Journal of Open Source Software*, **4**(43), 1686. doi:10.21105/joss.01686.

Wickham H, *et al.* (2023a). ***dplyr**: A Grammar of Data Manipulation*. R package version 1.1.2, URL https://CRAN.R-project.org/package=dplyr.

Wickham H, *et al.* (2023b). ***tidyr**: Tidy Messy Data*. R package version 1.3.0, URL https://CRAN.R-project.org/package=tidyr.

Affiliation:

Sebastian Krantz
Kiel Institute for the World Economy
Haus Welt-Club
Düsternbrooker Weg 148
24105 Kiel, Germany
E-mail: sebastian.krantz@ifw-kiel.de