JOURNAL OF REPRODUCIBLE STATISTICS
FAIR Press Journals · jrepstat.fairpressjournals.com · ISSN 3043-2790
Vol. 1 (2026), Art. 15
10.68365/jrepstat.2026.15

TEACHING CORNER · FOUNDING ARTICLE

Taking R to its limits: 90+ tips

Michail Tsagris[1*] and Manos Papadakis[2]

[1] *Department of Economics & Institute of Research in Education and Digital Social Sciences and Humanities, University of Crete, Gallos Campus, Rethimno, Greece*
[2] *Independent researcher, Heraklion, Greece*

[*]Corresponding author: mtsagris@uoc.gr

ARTICLE INFO

Keywords
R programming
efficient code
statistical computing
vectorization
memory optimization
numerical stability

History
Received 27 Sep 2026
Final version 18 Nov 2026
Accepted 14 Nov 2026
Version 1

Handling editor
Matthias Templ

ABSTRACT

R has many capabilities, most of which are unknown to many of its users and are waiting to be discovered. For this reason we provide a collection of more than 90 tips on how to write truly efficient code without having to program in C++, general programming advice, and tips for avoiding errors and numerical overflows.

✓ Reproduced verified 2026-11-13 REPRODUCIBILITY RECORD

Result	Reported results reproduced. The code runs end to end on Linux. Every printed comparison keeps its direction, most speed-up factors are smaller than on the authors' laptop.	✓ *measured*
Entry point	`cd code && Rscript R_code.R`	✓ *executed*
Environment	R 4.6.1, Rfast 2.1.5.2, data.table 1.18.6.1, microbenchmark 1.5.0, Matrix 1.7.5, float 0.3.4, matrixStats 1.5.0; rocker/r-ver:4.6.1 on colima, linux/aarch64	✓ *measured*
Determinism	Two runs byte-identical on Linux aarch64. Timings excluded, they measure the machine.	✓ *measured*
Archive	`taking-r-to-its-limits-90-tips-v1-repro.zip` · reviewed bundle SHA-256 `194e7784cbead53da72b320ea147e56e76be01ff11d70c61dc9e02398292f1f0`	✓ *computed*
Data	Open.	*declared, route checked*
AI use	Language editing, model not stated.	*declared*

1 Introduction

R is a scripting language and is not computationally efficient when compared to compiled languages, yet it has strong capabilities that are unknown to most of its users. Efficient code is very important in large–scale datasets, computations, or simulation studies. Efficient *R* programming has been treated before. (Wickham, 2015) devotes several chapters in his book to performance and memory, (Burns, 2011) catalogues common pitfalls in his book The R Inferno, and Gillespie & Lovelace (2016) gave a book–length treatment.

We have listed 91 tips that can help the user write more efficient code, use less memory, and become more familiar with *R* and its commands. The present list is oriented towards statistical computing, with an

CITE AS Tsagris, M., & Papadakis, M. (2026). Taking R to its limits: 90+ tips. Journal of Reproducible Statistics, 1, 15. https://doi.org/10.68365/jrepstat.2026.15

emphasis on matrix algebra, regression and simulation, includes many tips not documented elsewhere, and treats numerical stability alongside speed. The list is not exhaustive and we are certain there are tips we are not aware of. This paper is an updated extension of (Tsagris & Papadakis, 2018), where we removed some tips and added more than 20 tips. We ran some of the functions on a Dell laptop with Intel Core i7–1355U processor, 16GB RAM, 500 GB SSD and Windows OS, using *R* 4.6.1, and compared them to *R*'s built–in functions. We have not compared them to competing packages. In the uploaded *R* code we have added the session info of the *R* environment we used. The LAPACK version is 3.12.1, and BLAS version is 4.61.24651.0 (single threaded). We emphasize that the speed–up factors may vary across machines and or operating systems.

The speed–ups one obtains in small samples or low dimensions will be small, but in larger datasets the differences are observable. To put it simply, for someone who needs a simple car, a very expensive car or a jeep might not be of much use, especially if they do not go off–road. But for the user who needs a jeep, every bit of computational power, every second they can gain, matters. One might observe a time difference of only 5 seconds in the whole process. Differences from 40 seconds to 12 seconds, or to 22 seconds, are still worthwhile. Sometimes one tip saves 1 second, then another tip saves 1 more second, and so on, until 5 seconds are saved. For a researcher who has to run 1,000 simulations, this amounts to 5,000 seconds. Our motto is that every decrease in time or memory matters.

2 Measuring execution time

In order to measure the time a process, computation or simulation needs, we can do the following

```
runtime <- proc.time()
## put your function here
runtime <- proc.time() - runtime
## runtime gives you 3 numbers (all in seconds) like the ones below
user  system elapsed
0.18    0.07    3.35
```

The `elapsed` value is the desired metric. Alternatively, the package microbenchmark (Mersmann, 2015) allows for the comparison of two or more functions, measuring their execution time even at the nanosecond scale. The package bench (Hester & Vaughan, 2025) does the same and also reports the memory allocated.

3 General advice

We begin with a list of general programming advice.

1. When writing code, leave spaces between operations and parentheses, and indent the lines inside `for` loops. Put a comment at the end of every loop so as to know to which loop each closing bracket refers. This is really helpful if the code is long. Make the code readable and *clean*, not only for someone else, but mainly for ourselves. Clean code is easier to understand and edit. A good programmer makes mistakes, but detects them fast. If the code is long and has no spaces and no comments, spotting mistakes can become a difficult task.
2. Profile before optimizing. We are usually wrong about where our code spends its time. `Rprof` interrupts *R* every 20 milliseconds (by default) and records which function is running; `summaryRprof` then reports the time spent in each function. Only the functions at the top of the list are worth optimizing.

```
Rprof("prof.out")
## put your function here
Rprof(NULL)
summaryRprof("prof.out")$by.self
## profvis::profvis( your function ) gives the same
## information as an interactive graph
```

The function below is a worked example of profiling. It appends to a vector inside a loop, sorts it and returns the mean.

```
f <- function(n) {
  x <- NULL
  for ( i in 1:n )  x <- c(x, sqrt(i))
  y <- sort(x)
  sum(y) / length(y)
}
Rprof("prof.out", interval = 0.01)
a <- f(1e5)
Rprof(NULL)
summaryRprof("prof.out")$by.self
             self.time self.pct total.time total.pct
"c"               5.31    99.81       5.31     99.81
"Sys.getenv"      0.01     0.19       0.01      0.19
```

Almost all the time is spent in `c()`, which copies the whole vector at every iteration. The sorting, which one might have suspected, is negligible. Preallocating removes the bottleneck.

```
x <- NULL
for ( i in 1:n )  x <- c(x, sqrt(i))  ## not efficient
x <- numeric(n)
for ( i in 1:n )  x[i] <- sqrt(i)     ## efficient
x <- sqrt(1:n)                        ## even better
```

3. Verify before trusting a speed–up. Every time a slow implementation is replaced by a faster one, compare them on the same input. Use `all.equal`, which allows for a small tolerance, and not `identical` or `==`, because the order of the floating–point operations may have changed (see also the section on numerical overflows). Results that are equal in theory differ in the last bits when the operations are performed in a different order, which is exactly the situation when we compare a fast and a slow version of a function. Convergence checks in iterative algorithms suffer from the same problem and may never terminate.

```
0.1 + 0.2 == 0.3                          ## FALSE
isTRUE( all.equal(0.1 + 0.2, 0.3) )       ## TRUE
while ( u.new != u.old )                  ## may never stop
while ( sum( abs(u.new - u.old) ) > tol )  ## correct
all.equal(fast(x), slow(x))               ## TRUE, or a message
```

4. Avoid unnecessary calculations. In a discriminant analysis setting, for example, there is no need to calculate constant terms, such as $\log(2\pi)$, for each group and at every iteration. This only adds time, takes memory, and does not affect the algorithm or the result. The same holds for the Euclidean or the Hellinger distance: the square root and the multiplication by a common factor make no difference to the ordering of the distances and, if they can be avoided, they should be.
5. Try to make the mathematics, and hence the operations, as simple as possible. The partial correlation coefficient can be computed in many ways, two of which are via two regression models or via the simple correlation matrix. The second is much faster.
6. Suppose we have a function that uses many regression models, not necessarily from different downloadable *R* packages, but even from *R*'s built–in models. We should not create a separate function for each model. This seems like the obvious thing to do, simply because different regression models have different signatures and the hypothesis test for each of them is different: `lm` uses an F test, `glm` uses an F or a χ^2 test, and so on.

   The optimal solution is to write one function that includes all regression models and their arguments. This way we have one inclusive function for all models instead of many. Every time we want to add a new regression model we add it to this function and do not have to create a new one. This results in fewer files and less memory.

7. It is important to understand how *R* works, what operations do, and what we can and cannot do. The command `mahalanobis`, for example, uses the following line to calculate the distances.

```
rowSums(x %*% cov * x)
```

   The first key message is that the function performs one, and not two (as one would expect), matrix multiplications. Secondly, the user is given the chance to see that they can apply these types of operations. Thirdly, they are given the chance to make this function faster.
8. If we have a vector `x` of integer numbers, it is preferable to store it as `as.integer(x)` rather than `as.numeric(x)`. The former requires 4 bytes per element, the latter 8.
9. Copy–on–modify. When two names refer to the same object and one of them is modified, *R* silently copies the object first. Modifying a large object inside a function therefore duplicates it. The function `tracemem` prints a message every time an object is copied and is the simplest way to find unexpected copies.

```
x <- rnorm(1e7)
tracemem(x)
y <- x        ## no copy yet, x and y share the same memory
y[1] <- 0     ## now y is copied
x[1] <- 0     ## no copy, x has a single reference
```

10. The byte–code compiler. The `compiler` package translates *R* functions into byte–code, which runs faster, especially for loops and scalar arithmetic. Since *R* 3.4.0 the just–in–time compiler is enabled by default at its highest level, so every function is compiled the first time it is called and the user does not have to do anything. It can be checked and used explicitly as follows.

```
compiler::enableJIT(-1)   ## returns the current level, 3 = everything
f <- compiler::cmpfun(f)  ## compile a function explicitly
```

    Note that the compiler cannot vectorize a loop for us; the tips that follow remain necessary.

4 Simple functions

We continue with a list of simple functions that will help us make our code more efficient and faster.

11. Not all packages, or *R* itself, have the fastest implementations. The function `mean` is slower than `sum(x)/length(x)`, because `mean` is a generic function (method dispatch takes place) and performs two passes over the data in extended precision in order to reduce the rounding error, whereas `sum` is a `.Primitive` function, implemented directly in *C*, that performs a single pass. If we type `sum` we will see it is a `.Primitive` function, whereas `crossprod`, for example, is an `.Internal` function. Another example of a more advanced function is to perform many univariate logistic regressions and each time calculate the deviance of the model. If we create our own efficient functions, we may be surprised to see that they are faster than *R*'s built–in functions[1].
12. The commands `cor` and `cov` are two very fast functions that can be used to calculate the correlations or covariances between a vector and a matrix very efficiently. Many simple regression models can be computed using these two commands.
13. Search for functions that take less time. For example, `lm(y ~ x)` calls `lm.fit(x, y)` after building the design matrix from the formula. If we only require the coefficients, we should use `lm.fit` directly. The syntax is a bit different, since `x` must be the design matrix, but the speed is also very different, especially in large cases. The command `.lm.fit(x, y)` is even faster.
14. It's your algorithm. Tsagris (2017) showed that conditional G^2 tests of independence can become roughly 3–4 times faster if they are run using Poisson log–linear models instead of constructing the appropriate tables using `for` loops[2]. (Chatzipantsiou et al., 2019) proposed an extremely fast

[1] Note, however, that this is not always the case.
[2] This was observed in our Windows operating laptop, and hence speed–up factors may vary across machines.

algorithm for obtaining permutation–based and bootstrap–based p–values in the correlation and the two sample t–test cases, respectively. There are many ways to perform a computation; we should choose or implement the most appropriate and most efficient one. For example, in order to generate vectors from the multivariate normal distribution we may use the spectral or the Cholesky decomposition. The latter is faster. In general, we should try to find the optimal mathematical method for our problem.

15. Suppose we want to center some data. We can try with `apply`

```
cent <- function(x) x - mean(x)
a1 <- apply(x, 2, cent)
```

or use one of the following alternatives

```
a2 <- scale(x, center = TRUE, scale = FALSE)
m <- colMeans(x)
a3 <- sweep(x, 2L, m)
a4 <- t( t(x) - m ) ## see the section on matrices
dm <- dim(x)
n <- dm[1]
p <- dm[2]
a5 <- x - rep( m, rep(n, p) )
```

We generated a matrix with 10,000 rows and 20 columns. The time results using the package *microbenchmark* are as follows

```
Unit: microseconds
expr    min       lq      mean   median       uq     max neval
a1   1317.6  1494.80  2338.510  2215.30  2974.20  3878.8    20
a2    686.5   774.05  1019.975   890.95  1076.40  2968.4    20
a3    662.0   785.30  1146.100   897.40  1274.05  2912.3    20
a4    641.6   668.70   929.645   696.75   875.85  2539.2    20
a5    298.0   325.55   507.210   339.95   454.60  2610.5    20
```

Note that when measuring time with this or any similar package, a single laptop with fixed JIT and BLAS settings limits how far the ratios carry.

16. A very useful command is `tabulate`.

```
table(iris[, 5])
tabulate(iris[, 5])
```

Two differences between them are that `table` returns the values as names, whereas `tabulate` returns only the frequencies. `tabulate` works with factor variables as well. But if we have a numerical vector, we should make sure the numbers are consecutive and strictly positive, i.e. no zero is included.

```
x <- rep(0:5, each = 4)
table(x)
tabulate(x)  ## 0 is missing
x <- rep(c(1, 3, 4), each = 5)
table(x)
tabulate(x)  ## there is a 0 appearing indicating the absence of 2
```

The command `tabulate` is many times faster than `table`. For discriminant analysis algorithms `tabulate` is more useful, both because of speed and because it returns a 0 for a value with zero frequency. The drawback arises when we have negative numbers, zeros, or positive numbers that are not consecutive. If we want speed, we should formulate our data to match the requirements of `tabulate`.

17. To extract the mean vector of each group we can use a `for` loop or

```
a1 <- aggregate(x, by = list(ina), mean)
```

where `ina` is a numerical variable indicating the group. A faster alternative is the built–in command `rowsum`

```
a2 <- rowsum(x, ina) / as.vector( table(ina) )
## a bit faster option is a3 below
a3 <- rowsum(x, ina) / as.vector( tabulate(ina) )
```

Note that this command works also for matrices. Note though that `tabulate` returns the frequency of consecutive strictly positive integers and should be treated with caution as the previous tip mentions. An alternative would be to use the package data.table (Barrett et al., 2026)

```
library(data.table)
dt <- as.data.table(x)                  ## x is the numeric matrix
dt[, ina := ina]                        ## add the grouping variable
a1 <- dt[, lapply(.SD, sum),  by = ina] ## group sums
a2 <- dt[, lapply(.SD, mean), by = ina] ## group means
```

18. It is suggested to use `which.min(x)` and `which.max(x)` instead of `which( x == min(x) )` and `which( x == max(x) )` to find the position of the minimum and maximum, respectively. The former perform a single pass over the data, the latter two.
19. Incremental calculations. An example is the first–order partial correlations, which can be computed from the simple correlations with fewer calculations. In general, if our function can perform calculations incrementally, the boost in speed can be very high.
20. When calculating operations such as `sum(a * x)`, where `x` is a vector or a matrix and `a` is a scalar, write `a * sum(x)`. The two are equal by the distributive law of multiplication over addition, $\sum_{i=1}^{n} ax_i = a\sum_{i=1}^{n} x_i$, so we are free to change the order of the operations. In the first case, the scalar is multiplied with all elements of the vector (many multiplications), whereas in the second case the sum is calculated first and then a single multiplication between two numbers takes place.

```
sum(a * x)  ## n multiplications and n-1 additions
a * sum(x)  ## n-1 additions and 1 multiplication
```

21. Suppose we want to calculate the factorial of some integers and most (or all) of those integers appear more than once (Poisson, beta binomial, beta geometric, or negative binomial distribution, for example). Instead of performing the operation for each element, we can do it for the unique values only and weight each result by its frequency, as in the example below. Note, however, that this trick does not always pay off. It will work in the case where we have many repeated integers and a `for` or a `while` loop, and hence we have to calculate the factorials repeatedly.

```
x <- rpois(10000, 5)
sum( lgamma(x + 1) )
y <- sort( unique(x) )
ny <- as.vector( table(x) )
sum( lgamma(y + 1) * ny )
```

22. If we use the `glm` or `lm` commands multiple times, then we should write

```
glm(y ~ x, family = ..., y = FALSE, model = FALSE)
lm(y ~ x, y = FALSE, model = FALSE)
```

The two extra arguments `y = FALSE, model = FALSE` reduce the memory requirements of the fitted object.

23. When calculating $\log(1 + x)$ use `log1p(x)` instead of `log(1 + x)`. The advantage is accuracy: when x is very small, $1 + x$ is rounded to 1 and `log(1 + x)` returns 0, whereas `log1p(x)` returns the correct value. The same holds for $e^x - 1$ and `expm1(x)`.

```
x <- 1e-17
log(1 + x)   ## 0
log1p(x)     ## 1e-17
```

24. Vectorization can save a tremendous amount of time even in small datasets. Try to avoid `for` loops by using vectorized operations or matrix multiplications. For example, instead of

    ```
    for (i in 1:n) y[i] <- x[i]^2
    ```

    we can write

    ```
    y <- x^2
    ```

    This is a very easy example, but the point is made. Vectorization requires a lot of thinking and is not always applicable, but when it is, it can yield substantial computational savings.
25. A useful command is `outer`. An example is where we have two vectors `x` and `y` and we want to add each element of `y` to each element of `x`. The result is a matrix.

    ```
    z <- matrix(0, nrow = length(x), ncol = length(y) )
    for ( i in 1:dim(z)[1] )  z[i, ] <- x[i] + y
    ## The above task can take place a lot faster by typing
    outer(x, y, "+")
    ```

26. The command `sort` has an extra feature: by specifying `index.return = TRUE`, the outcome is a list with the sorted numbers along with their ordering indices. That is, `sort(x, index.return = TRUE)` sorts the numbers and returns their order as if we had used `order` separately.
27. We can partially sort a vector. In the *k–NN* algorithm, for example, we want the *k* smallest distances. There is no need to sort all the distances, only the *k* smallest: `sort(x, partial = 1:k)`.
28. Many functions call internal functions. `sort`, for example, calls `sort.int`, and `sample` calls `sample.int`. Calling the internal function directly avoids the argument checking of the wrapper.
29. Suppose we are given many matrices in a list, `A`, and wish to stack them one under the other into a new matrix. The command to do this efficiently is

    ```
    do.call(rbind, A)
    ```

30. Use `seq_len(n)` instead of `1:n` and `seq_along(x)` instead of `1:length(x)` in loops. When `n` is 0, `1:n` is `c(1, 0)` and the loop runs twice on non–existent elements, whereas `seq_len(0)` is empty and the loop is skipped.

    ```
    for ( i in 1:length(x) )    ## fails when x is empty
    for ( i in seq_along(x) )   ## correct
    ```

31. Use `vapply` instead of `sapply` when the function returns a value of known type and length. `sapply` inspects all the results in order to decide what to return, whereas `vapply` is told in advance, allocates the result once, and fails if a result has the wrong type.

    ```
    sapply(x, f)               ## classical
    vapply(x, f, numeric(1))   ## more efficient and safer
    ```

32. Use `anyNA(x)` instead of `any(is.na(x))`. The latter allocates a logical vector of the same length as `x` and scans it entirely, whereas the former stops at the first missing value.

    ```
    any( is.na(x) )  ## classical
    anyNA(x)         ## more efficient
    ```

33. When flattening a list with `unlist`, drop the names if they are not needed. Constructing the names is often more expensive than the flattening itself.

```
unlist(x)                     ## classical
unlist(x, use.names = FALSE)  ## more efficient
```

34. The vectorized `ifelse` evaluates both branches in full and carries a large overhead. For a scalar condition use `if-else`, and for numeric vectors use arithmetic on the logical condition.

```
ifelse(test, 1, 2)            ## slow for a scalar test
if (test) 1 else 2            ## efficient
ifelse(x > 0, x, 0)           ## classical
x * (x > 0)                   ## more efficient
```

    Note however, that these two differ when x contains NA, so users should not swap them blindly.
35. For binning a numerical vector into intervals use `findInterval` instead of `cut`. `cut` constructs a factor with character labels, whereas `findInterval` returns the integer index of the interval directly, which is what is usually needed afterwards (for `tabulate` or `rowsum`).

```
as.integer( cut(x, breaks) )   ## classical
findInterval(x, breaks)        ## more efficient
```

5 Calculations involving matrices

R is not designed to handle large–scale datasets, yet there are many ways to handle matrices efficiently, and we present some tricks below.

36. Use `colMeans`, `rowMeans`, `colSums` and `rowSums` instead of `apply(x, 2, mean)`, `apply(x, 1, mean)`, etc., as they are extremely fast. In addition, many really fast functions can be built using these commands. In general, we should avoid `apply` and `aggregate` whenever possible. For the median, though, there is no built–in alternative, and `apply(x, 2, median)` should be used instead of a `for` loop going through every column of the matrix. The `for` loop is not slower, but the `apply` is neater.
37. In the same spirit as `.lm.fit`, the functions `.colSums`, `.rowSums`, `.colMeans` and `.rowMeans` are the internal versions of `colSums` etc. They skip the argument checks, at the price of passing the dimensions explicitly. The gain is noticeable when they are called many times on small matrices.

```
colSums(x)              ## classical
.colSums(x, n, p)       ## more efficient inside loops
```

38. If we have to use a `for` loop going through the rows of a matrix, consider transposing the matrix and then going through its columns.
39. If we are given a matrix and, inside a `for` loop, we would like to extract some specific columns/rows each time, we can instead store the indices of the columns/rows inside the `for` loop and, outside the loop, extract the columns/rows once and perform the operations we want.
40. If we want the number of rows or columns of a matrix $\mathbf{X}$, we should not use `nrow(X)` or `ncol(X)`, but `dim(X)[1]` or `dim(X)[2]`, as they are roughly 2 times faster. `nrow` and `ncol` are *R*–level functions that call `dim` and index the result, so we pay for an extra function call. Note though that the benefit is very tiny, since both functions require nanoseconds, and the time gain seem might seem negligible. We mention this tip here though to help the reader develop the efficient programming way of thinking.
41. Suppose we have a matrix $\mathbf{X}$ and we want the position of the maximum for each row. The obvious solution is `apply(X, 1, which.max)`. The efficient solution is `max.col(X)`. Note that by default `max.col` breaks ties at random; use `max.col(X, ties.method = "first")` to obtain the same result as `which.max`.
42. If we want to subtract a vector from a matrix in a row–wise fashion, we should be aware of the fact that *R* does it column–wise. This is because *R* reads, writes and stores data in a column–wise fashion. For example, let $\mathbf{X}$ be a matrix and $\mathbf{y}$ a vector whose length is equal to the number of columns of $\mathbf{X}$. We should type

```
t(X) - y
```

43. If we transpose a matrix and never use the initial matrix in the subsequent steps, it is best to delete the initial matrix, or even better, to store its transpose in the same object. That is, if we have a matrix **X**, we should type

```
Y <- t(X)  ## not suggested
X <- t(X)  ## suggested
```

    We repeat that this applies only when `X` is not used again in later steps. The reason is memory saving: if `X` is big and we keep a second object as big as the first one, we request extra memory for no reason.
44. Use the command `princomp` instead of `prcomp` for principal component analysis[3]. We generated a matrix with 10,000 rows and 50 columns to benchmark this and the results are presented below.

```
Unit: milliseconds
expr             min       lq      mean   median        uq      max  neval
prcomp(x)    43.0756  45.2539  46.48306  46.2098  47.53110  50.5531     20
princomp(x)  24.5183  26.3287  29.78383  28.4339  30.25745  51.3666     20
```

45. Create vectors or matrices of the required size from the start. Instead of making a vector longer each time using `c()`, create an empty vector with the required size. The commands `rbind` and `cbind` are useful, but come at a heavy price: each call allocates a new object and copies all the data, so they are very expensive when called many times (see the profiling example in the general advice).

```
x <- numeric(n)          ## vector of n zeros
x <- integer(n)          ## integer vector
x <- matrix(0, n, p)     ## n x p matrix
x <- vector("list", n)   ## list of n NULLs
```

46. For group covariance matrices the command `by` could be used. The matrices are stored in a list and the command `simplify2array` is then needed to convert the list to an array, in order to calculate, for example, the determinant of each matrix. A `for` loop is faster.
47. What if we have an array of matrices and want to calculate the sum or the mean of all the matrices? The obvious answer is `apply(x, 1:2, mean)`. Since *R* works column–wise, we can instead use `t( colSums( aperm(x) ) )` and `t( colMeans( aperm(x) ) )` for the sum and the mean, respectively.

```
x <- array( dim = c(1000, 10, 10) )
for (i in 1:10) x[, , i] <- matrix( rnorm(1000 * 10), ncol = 10 )
a1 <- apply(x, 1:2, mean)
a2 <- t( colMeans( aperm(x) ) )
```

48. If we want to calculate the logarithm of the determinant of a matrix **X**, instead of `log( det(X) )` we can type `determinant(X, logarithm = TRUE)`, as it is slightly faster for small matrices. For big matrices, say of dimensions 100×100 or more, the difference becomes negligible.
49. Suppose we want the Euclidean distances of a single vector from many others (say thousands). The inefficient way is to calculate the distance matrix of all points and take the row that corresponds to our vector. An efficient way is to use the Mahalanobis distance with the identity matrix as the covariance matrix.

```
x <- rnorm(50)
y <- matrix( rnorm(1000 * 50), ncol = 50 )
a1 <- dist( rbind(x, y) )  ## inefficient way
Ip <- diag(50)
```

[3]Note however, that `princomp` works from the covariance matrix and its eigendecomposition, which loses precision relative to the SVD of the centered data that `prcomp` uses, and the difference shows up on ill–conditioned or nearly collinear data that applied users might meet.

```
                          ## below is a better way
a2 <- sqrt( mahalanobis( y, center = x, cov = Ip, inverted = TRUE ) )
```

Another way is the following

```
z <- t(y) - x             ## R recycles column--wise
a3 <- sqrt( colSums(z^2) )
```

50. Calculating $\mathbf{X}^\top \mathbf{Y}$ as `t(X) %*% Y` instead of `crossprod(X, Y)` forces *R* to allocate a transposed copy of $\mathbf{X}$ before the multiplication, whereas `crossprod` passes the transposition flag to the underlying BLAS routine and no copy is made. Note however, that these margins depend on the linear algebra backend and can narrow or reverse under an optimized multithreaded BLAS.
51. Similarly, the product $\mathbf{X}^\top \mathbf{X}$ of an $n \times p$ matrix is computed faster by `crossprod(X)` than by the explicit matrix multiplication, since the result is known to be symmetric.

```
t(X) %*% Y       ## classical
crossprod(X, Y)  ## more efficient
X %*% t(Y)       ## classical
tcrossprod(X, Y) ## more efficient
t(X) %*% X       ## classical
crossprod(X)     ## more efficient
```

52. Let $\mathbf{X}$ and $\mathbf{m}$ be a matrix and a vector, and suppose we want the sum of their elementwise product. There are two ways to do it.

```
sum(m * t(x))
sum(x %*% m)  ## more than 2 times faster
```

53. When working with arrays it is more efficient to have them transposed. For example, if we have K covariance matrices of dimension $p \times p$, we would naturally create an array of dimensions `c(p, p, K)`. Make its dimensions `c(K, p, p)` instead. If we want, for example, to divide each matrix by a different scalar, in the first case we have to use a `for` loop, whereas in the transposed case we simply divide the array by the vector of scalars.
54. If we want to invert a positive definite matrix (e.g. a covariance matrix) we should use `chol2inv( chol( X ) )`, as it is faster than `solve(X)`.
55. To invert a matrix (not necessarily positive definite) and multiply the result with a vector or another matrix, there are two ways

```
solve(X) %*% Y  ## classical
## a much more efficient way is to not invert the matrix X
solve(X, Y)
```

56. The quantity $\text{tr}\left(\mathbf{X}^\top \mathbf{X}\right)$ can be evaluated either via

```
sum( diag( crossprod(X) ) )
```

or, faster, via

```
sum(X * X) ## or
sum(X^2)
```

Note that for a square matrix $\text{tr}\left(\mathbf{X}^2\right)$ is a different quantity, equal to `sum(X * t(X))`.

57. Similarly, the trace $\text{tr}\left(\mathbf{X}^\top \mathbf{Y}\right)$ can be computed either via

```
sum( diag( crossprod(X, Y) ) )  ## just like before
```

or, faster, via

```
sum(X * Y)  ## faster, like before
```

58. In the same spirit, if we want the diagonal of the product of two matrices, type

```
diag( tcrossprod(X, Y) )  ## for example
rowSums(X * Y)            ## this is faster
```

59. Suppose we have two matrices **A**, **B** and a vector **x** and we want **ABx** (the dimensions must match, of course).

```
A %*% B %*% x    ## inefficient way
A %*% (B %*% x)  ## efficient way
```

In the first case we perform a matrix–by–matrix multiplication followed by a matrix–by–vector multiplication. In the second case we perform two matrix–by–vector multiplications, which require far fewer operations, and avoid extra calculations.

60. For the reconstruction of a matrix from its eigendecomposition, there are two ways to do the multiplication

```
s <- matrix( rnorm(100 * 100), ncol = 100 )
s <- crossprod(s)
eig <- eigen(s)
vec <- eig$vectors
lam <- eig$values
a1 <- vec %*% diag(lam) %*% t(vec)
a2 <- vec %*% ( t(vec) * lam )  ## faster way
```

61. The exponential term of the multivariate normal density can be calculated either using matrices or simply with the command `mahalanobis`. If we have many observations, many dimensions, or many groups, this can save a **lot** of time.

```
x <- matrix( rnorm(1000 * 20), ncol = 20 )
m <- colMeans(x)
dm <- dim(x)
n <- dm[1]
p <- dm[2]
s <- cov(x)
a1 <- diag( (x - rep(m, rep(n, p)) ) %*% solve(s)
            %*% t(x - rep(m, rep(n, p)) ) )
a2 <- diag( t( t(x) - m ) %*% solve(s) %*% ( t(x) - m ) )
a3 <- mahalanobis(x, m, s)  ## much faster
```

62. Suppose we want to compute the trace of the square of the correlation matrix **R**. It is well known from linear algebra (Strang, 2023) that for a symmetric matrix **A**, its eigendecomposition is given by $\mathbf{A} = \mathbf{\Gamma}\mathbf{\Lambda}\mathbf{\Gamma}^\top$, where $\mathbf{\Gamma}$ is the matrix of eigenvectors and $\mathbf{\Lambda}$ is a diagonal matrix that contains the p eigenvalues $\lambda_1, ..., \lambda_p$. Note that $\text{tr}(\mathbf{A}) = \sum_{j=1}^{p} \lambda_j$ and $\text{tr}(\mathbf{A}^2) = \sum_{j=1}^{p} \lambda_j^2$. This trick allows us to avoid computing the correlation matrix, multiplying it by itself and then summing its diagonal elements.

R's built–in function `prcomp()` computes principal components by singular value decomposition of the (centered) data matrix in a fast and memory–efficient manner. Its drawback is that it computes p eigenvectors and p eigenvalues, when in fact at most $n - 1$ eigenvalues are non–zero, since the rank of the correlation matrix is at most $n - 1$ when $n < p$. Hence, instead of `prcomp()`, the function `hd.eigen()` available in *Rfast* can be applied. This function is optimized for the case $n < p$ and returns the first n eigenvectors and eigenvalues. Thus, `hd.eigen()` is not only faster than `prcomp()` in high–dimensional settings, but also requires less memory. The function takes the original data ($\mathbf{Y}$), standardizes them ($\mathbf{Z}$), and performs the eigendecomposition of the matrix $\mathbf{Z}\mathbf{Z}^\top$. The sum of the squared eigenvalues divided by $(n-1)^2$ equals $\text{tr}(\mathbf{R}^2)$.

63. Prefer matrices to data frames for numerical work. A data frame is a list of columns, so every operation on it goes through method dispatch, and extracting a row constructs a new data frame. If all the columns are numeric, convert once at the start of the function.

```
for (i in 1:n)  f( df[i, ] )    ## very slow
x <- as.matrix(df)
for (i in 1:n)  f( x[i, ] )     ## much faster
```

64. Use `drop = FALSE` when subsetting matrices. By default `X[, 1]` returns a vector, so a function that extracts `X[, ind]` will silently receive a vector when `ind` has length 1, and `dim` or `colSums` will fail. This tip is about correctness rather than speed.

```
X[, ind]                ## vector when length(ind) == 1
X[, ind, drop = FALSE]  ## always a matrix
```

65. If a linear system involves a triangular matrix, e.g. the Cholesky factor of a covariance matrix, do not use `solve`, which treats the matrix as a general one. `backsolve` and `forwardsolve` exploit the triangular structure and solve the system in $O(p^2)$ instead of $O(p^3)$ operations.

```
L <- chol(S)                ## upper triangular, S = t(L) %*% L
solve(L, b)                 ## classical
backsolve(L, b)             ## more efficient
forwardsolve( t(L), b )     ## for the lower triangular factor
```

66. When the matrix is symmetric, declare it. By default `eigen` checks for symmetry by comparing all pairs of elements; declaring it skips the check, uses the faster symmetric routine, and guards against a false negative due to rounding errors in a computed covariance matrix. If only the eigenvalues are needed, say so as well.

```
eigen(S)                                      ## classical
eigen(S, symmetric = TRUE)                    ## more efficient
eigen(S, symmetric = TRUE, only.values = TRUE) ## eigenvalues only
```

67. Sparse matrices. If most elements of a matrix are zero (contingency tables with many categories, adjacency matrices, design matrices with many dummy variables), the package Matrix (Bates et al., 2026) stores only the non–zero elements, and the cost of multiplication and `solve` scales with the number of non–zeros. Sparsity is lost by operations that fill in the zeros, e.g. adding a scalar or centering.

```
library(Matrix)
xs <- Matrix(x, sparse = TRUE)
crossprod(xs)                    ## works as for a dense matrix
solve(xs, b)
```

68. Single precision. *R* stores real numbers in double precision (8 bytes). The package float (Schmidt, 2025) provides 32–bit (4–byte) matrices with the usual arithmetic and linear algebra defined on them. This halves the memory and roughly doubles the throughput of BLAS operations, at the cost of about 7 significant digits instead of 16. It is inappropriate for iterative algorithms where rounding errors accumulate.

```
library(float)
xf <- fl(x)               ## half the memory of x
crossprod(xf)
dbl( crossprod(xf) )      ## back to double precision
```

69. Multiple responses in one call. If we need the regression of many response variables on the same design matrix (genome–wide analyses, permutation tests), we should not loop over the responses.

`lm.fit` and `.lm.fit` accept a matrix of responses and fit all the regressions with a single QR decomposition of `x`. If the responses arrive one at a time, compute the QR decomposition once and reuse it.

```
for (j in 1:q)  b[, j] <- lm.fit(x, Y[, j])$coefficients  ## classical
b <- lm.fit(x, Y)$coefficients                             ## efficient
q <- qr(x)           ## decompose once
qr.coef(q, y1)       ## reuse for each response
qr.coef(q, y2)
```

70. Keep distances as a `dist` object. The function `dist` returns the lower triangle of the distance matrix as a vector of $n(n-1)/2$ elements. Converting it to a full matrix more than doubles the memory, and many functions (`hclust`, `cmdscale`) accept the `dist` object directly. If the full matrix is required, *Rfast* computes it directly in *C++*.

```
d <- as.matrix( dist(x) )           ## classical, n x n matrix
d <- dist(x)                        ## n(n-1)/2 vector, half the memory
d <- Rfast::Dist(x)                 ## full matrix, much faster
d <- Rfast::Dist(x, result = "vector")  ## lower triangle only
d <- Rfast::dista(x, y)             ## distances between rows of x and y
```

71. To sum many matrices of the same dimensions stored in a list `A`, use `Reduce`. This applies the operator cumulatively and allocates one result, instead of copying all the matrices into an array first.

```
apply( simplify2array(A), 1:2, sum )  ## classical
Reduce("+", A)                        ## more efficient
```

72. The function `quantile` is surprisingly slow, because it sorts the whole vector and handles nine quantile definitions. If a single quantile is needed, use partial sorting. The k–th smallest value is `sort(x, partial = k)[k]`, and the default (type 7) quantile of order α interpolates between two order statistics.

```
quantile(x, alpha)                        ## classical
h <- (n - 1) * alpha + 1
k <- floor(h)
y <- sort(x, partial = c(k, k + 1))
q <- y[k] + (h - k) * (y[k + 1] - y[k])   ## more efficient
```

For the median, `median` already uses this trick, and `colMedians` of *Rfast* applies it column–wise.

6 Numerical optimization

73. The command `nlm` is usually faster than `optim` for optimization purposes, but `optim` is more reliable and robust. In some cases `optim` with the argument `method = "BFGS"` can be faster, but these depend upon the function being minimized.
74. If we have a function for which some parameters have to be positive, we should not use constrained optimization, but instead exponentiate the parameter inside the function. The parameter can take any value on the real line $\mathbb{R}$, but inside the function its exponentiated form is used. In the end, we simply take the exponential of the returned value. For its variance we use the δ–method (Casella & Berger, 2002). The trick is to use a link function, similarly to generalised linear models.
75. There are two ways to estimate the parameters of a distribution, the Dirichlet for example: either with the use of `nlm`, or via the Newton–Raphson algorithm. We performed some simulations and saw that Newton–Raphson can be at least 10 times faster. The same is true for the circular regression of Presnell et al. (1998) when comparing `nlm` with the E–M algorithm described by the same authors. Switching to E–M or Newton–Raphson, and not relying on `nlm`, can save a lot of time. If we have the description of the E–M or the Newton–Raphson algorithm available, we should implement it. Of the two, Newton–Raphson is faster.

76. If we have an iterative algorithm, such as Newton–Raphson, E–M or fixed–point iteration, and we stop when the vector of parameters does not change any further, we should not use `rbind`, `cbind` or `c()` to store the whole sequence. Instead, we store only two values, `vec.old` and `vec.new`. That is, we should not type

```
u[i, ] <- u[i - 1, ] + W %*% B  ## not efficient
```

but

```
u.new <- u.old + W %*% B        ## efficient
```

So, at every iteration we keep two vectors only, not the whole sequence. The same is true for the log–likelihood or the criterion of interest. Unless we want to keep track of how the values change, our advice is to keep only the current and the previous value. Apart from being faster, this also uses less memory.

7 Numerical overflows

Numerical instabilities occur frequently when using real, not simulated, data. We mention these tricks because one should not sacrifice numerical stability for the sake of speed. When trying to speed up our code we experienced numerical instabilities and overflows, and we would like to share some of the issues we faced.

77. The fitted values of a logistic regression model make use of the inverse logit function $\frac{e^x}{1+e^x}$. If x is a very large number, both the numerator and the denominator overflow to `Inf` and the result is `NaN`. If, however, we write the formula in its equivalent form $\frac{1}{1+e^{-x}}$, the result is 1, as it should be.

```
x <- 1000
exp(x) / (1 + exp(x))   ## NaN
1 / (1 + exp(-x))       ## 1
```

78. The same is true for the Poisson regression, or other regressions that use the log link function. In those cases e^x is used. It might be useful to consider using e^{-x} if there is a possibility of having very large values, or simply standardize the predictor variables.
79. Kernel functions are of the form $e^{\frac{f(\mathbf{x})}{h}}$, where h is a scalar and $\mathbf{X}$ is a matrix. In order to speed up the calculations one could pre–compute $e^{f(\mathbf{x})}$ and then raise the result to the power $1/h$ for the different values of h. This can easily result in overflow and `Inf` values and hence caution must be taken.
80. The function $\log(1+e^x)$ behaves like x for large x; in double precision the two become equal for $x \gtrsim 37$, and e^x overflows for $x > 709$, so that even `log1p(exp(x))` returns `Inf`. The stable way to compute it is $\max(x, 0) + \log\left(1 + e^{-|x|}\right)$.

```
log1p( exp(x) )                    ## Inf for x > 709
pmax(x, 0) + log1p( exp(-abs(x)) )  ## never overflows
```

81. The product $\prod_{i=1}^{n} x_i$, where $0 < x_i \leq 1$, underflows to 0 very fast. In order to avoid this, we should compute $e^{\sum_{i=1}^{n} \log x_i}$, or better, work with the logarithm of the product throughout.

```
prod(x)             ## 0
exp( sum(log(x)) )  ## correct
```

82. We use the logarithm of the p–values and not the p–values themselves. This is very beneficial when our algorithm calculates and compares p–values. p–values underflow to zero when they are near $1e-308$ (and subnormals to $5e-324$), so *R* cannot sort them correctly since they are all considered equal. The logarithms of the p–values can be sorted correctly.

```
pchisq(stat, dof, lower.tail = FALSE)                ## can be 0
pchisq(stat, dof, lower.tail = FALSE, log.p = TRUE)  ## sortable
```

83. The log–sum–exp trick. Quantities of the form $\log \sum_{i=1}^{n} e^{x_i}$ appear in normalizing constants, mixture likelihoods, the softmax function and the E–step of the E–M algorithm. Computing them literally overflows when some $x_i > 709$ and underflows to $-\infty$ when all $x_i < -745$. The stable formula is $m + \log \sum_{i=1}^{n} e^{x_i - m}$ with $m = \max_i x_i$, since then the largest exponent is $e^0 = 1$.

```
log( sum( exp(x) ) )          ## overflows
m <- max(x)
m + log( sum( exp(x - m) ) )  ## stable
matrixStats::logSumExp(x)     ## the same, and row/column versions
```

84. Work with log–densities. All the density functions of *R* accept `log = TRUE` and compute the logarithm directly, whereas `log(dnorm(...))` underflows to $-\infty$ in the tails. Similarly, `lgamma`, `lbeta`, `lchoose` and `lfactorial` should be used instead of the logarithm of `gamma`, `beta`, `choose` and `factorial`, which overflow. Exponentiate, if at all, only at the very end.

```
log( dnorm(x, m, s) )        ## -Inf for x far in the tail
dnorm(x, m, s, log = TRUE)   ## correct
log( choose(1100, 550) )     ## Inf
lchoose(1100, 550)           ## correct
```

85. Integer overflow. Integers in *R* are 32–bit, so the largest one is `.Machine$integer.max` $= 2,147,483,647$. Arithmetic between integers that exceeds this returns `NA` with a warning. This typically arises when `n` comes from `length`, `dim` or `nrow`, which return integers. For exact arithmetic beyond 2^{53} the package bit64 (Chirico & Oehlschlägel, 2026) provides 64–bit integers.

```
n <- dim(x)[1]         ## an integer, say 50000
n * n                  ## NA with a warning
as.numeric(n) * n      ## 2.5e9
```

8 Reading large–scale datasets

86. For reading large text files, `read.csv` and `read.table` are slow: they read the file line by line and guess the type of every column. The function `fread` of the package data.table reads the file with multiple threads and is typically more than 10 times faster; `read_csv` of readr (Wickham et al., 2026) is a second option.

```
x <- read.csv("data.csv")             ## classical
x <- data.table::fread("data.csv")    ## much faster
```

9 Parallel computing in *R*

If we have a machine with more than one core, we can put them all to work simultaneously and speed up the process considerably. Combining this with the tricks above is even more beneficial.

The idea is to use a library that allows parallel computing. We make use of the built–in parallel package. Below are some instructions on how to use it. Several of our packages include parallel computing as an option, so if the notes below are not clear, one can always consult those functions. Suppose we are given a matrix with discrete data and wish to perform maximum likelihood estimation for the zero inflated Poisson distribution for each column of the given matrix. The relevant function (`zip.mle`) is available in the package Rfast (Papadakis et al., 2025).

```
n <- dim(x)[2]
res <- matrix(nrow = n, ncol = 4)
for ( i in 1:n )  res[i, ] <- unlist( Rfast::zip.mle(x[, i], 1e-6) )
colnames(res) <- c('iters', 'loglik', 'lambda', 'pi')
```

Using the parallel package the above code translates into the following.

```
cl <- makeCluster(ncores)  ## ncores is the number of cores to use
clusterExport(cl, "x")
clusterEvalQ(cl, library(Rfast))
res <- parSapply(cl, 1:n, function(i) unlist(Rfast::zip.mle(x[, i], 1e-6)))
stopCluster(cl)
res <- t(res)
colnames(res) <- c('iters', 'loglik', 'lambda', 'pi')
```

87. On Linux and macOS, `mclapply` of the built–in package parallel package is often simpler and faster than a socket cluster: it forks the current *R* process, so the workers see all its objects without any copying or exporting, and there is no cluster to set up or stop.

    ```
    res <- parallel::mclapply(1:B, f, mc.cores = nc)  ## Linux / macOS
    ```

    The previous example for instance would be

    ```
    res_list <- mclapply(1:n, function(i) unlist(Rfast::zip.mle(x[, i], 1e-6)),
    mc.cores = ncores)
    res <- do.call(rbind, res_list)
    colnames(res) <- c('iters', 'loglik', 'lambda', 'pi')
    ```

10 Efficiently written functions in *R* packages

Multinomial regression is offered in the package VGAM (Yee, 2010), but also in the package nnet (Venables & Ripley, 2002). The implementation in the second package is much faster. The same is true for ordinal logistic regression in VGAM and in ordinal (Christensen, 2015); the latter is much faster. Also, the package *fields* (Nychka et al., 2015) has a function called `rdist` which is faster than the built–in `dist`. A fast implementation of the distance correlation is available in the package dcov (Weiqiang, 2020), and more memory– and computationally–efficient energy–related functions can be found in the package estats[4] (Tsagris & Papadakis, 2026a). These are just some examples where a function is available in more than one package, but the implementations do not share the same computational cost.

Regression models, distance matrix computations, statistical tests, utility functions and many more can also be found in the packages Rfast (Papadakis et al., 2025) and Rfast2 (Papadakis et al., 2026). These two packages contain many fast or really fast functions, either written in *C*++ or written in *R* exploiting fast built–in functions. The packages matrixStats (Bengtsson, 2025) and collapse (Krantz, 2026) offer similar collections of row– and column–wise functions written in *C* or *C*++, and they complement each other.

11 More advanced programming tips

Some programming languages offer advanced features. We mention some of those that *R* supports, along with their strengths and benefits.

88. Auto–printing our own class in our own style. In *R* we can print a variable using the functions `print` and `cat`, or by just typing the variable's name and pressing enter. But what if we do not like the way *R* prints it? That is the point of auto–printing. *R* supports a general way to print our own class through its default function `print`. The method is very simple, with one requirement: we must change the class.

[4] See (Tsagris & Papadakis, 2026b) for the mathematical and computational tricks.

```
print.new_class_name <- function(x, ...) {
  cat("auto-printing my variable with class 'new_class_name'
  and value: ", x)
}
```

Example:

(a) Create a variable: `x <- 0`
(b) Change its class: `class(x) <- "anything"`
(c) Create the function to print the variable:

```
print.anything <- function(x, ...) {
  cat("Auto-printing variable of class anything: ", x)
}
```

(d) Type: `print(x)`, or simply `x`.
(e) That's it.

89. Using `if-else` in one line. Another feature of *C++* is the ternary operator (which cannot be overloaded). It is an `if-else` statement with a different syntax and one more capability: it returns the value of the selected branch, i.e. it is an `if-else` with assignment. *R* supports the same thing with its default `if-else` syntax, which can reduce the length of the code.
Example:

```
if ( is_true ) {
  x <- 1
} else {
  x <- 2
}
```

The above example is very simple but needs 5 lines. Let us reduce it:

```
x <- if (is_true)  1  else  2
```

In a single line we have the same code. Be careful with the syntax, and make sure the last statement of each branch is the one we want to assign to `x`. For more than one command use curly brackets. This works because `if-else` in *R* is like a function that always returns its last evaluated expression, without the `return` keyword. *Note:* the same holds for `if-else-if`. This is a matter of style; there is no speed difference, since it is the same `if` construct.

90. Iterating over a vector. If we want to iterate over a vector and use its elements, we can write:

```
x <- rnorm(10)
for ( i in seq_along(x) ) {
  cat(x[i])
}
```

This is the classic way. Another way is:

```
x <- rnorm(10)
for ( i in x ) {
  cat(i)
}
```

The second way is equivalent to the first, but instead of iterating over the indices we iterate over the elements themselves. This eliminates one indexing operation per element, i.e. `length(x)` operations in total. The gain is small though.

91. Use "`[[`" instead of "`$`" for programmatic access to lists and data frames. The "`$`" operator performs partial matching on the name, which is a source of bugs in functions, and on data frames it dispatches

to a method. "[[" matches exactly, takes the name as a variable, and is faster inside loops. The internal `.subset2` does the same without any dispatch.

```
x$val                ## returns x$value if "val" is a unique prefix
x[["value"]]         ## exact, and the name can be a variable
.subset2(x, "value")  ## no dispatch at all
```

12 Conclusions

We have provided a long list of efficient coding tips to help the user make their programs faster. In some cases, the user can benefit from functions that use less memory, which is another important feature apart from speed. We also provided tips for avoiding numerical instabilities and overflows. These tips should not go unattended, as one can easily face such errors when trying to optimize the efficiency of their code.

Acknowledgements

We would like to thank Stefanos Fafalios, Christina Chatzipantsiou, Marco Maier, Panagiotis Tzirakis, Giorgos Athineou, Kleio Lakiotaki, Kurt Hornik, Uwe Ligges and the *R* Core team.

Conflicts of Interests

M. Tsagris and M. Papadakis are the maintainers of the *R* packages Rfast, Rfast2 and estats, which are recommended in this paper. The authors declare no other competing interests.

References

Barrett, T., Dowle, M., Srinivasan, A., Gorecki, J., Chirico, M., Hocking, T., Schwendinger, B., & Krylov, I. (2026). *data.table: Extension of 'data.frame'* [R package version 1.18.6.1]. https://doi.org/10.32614/CRAN.package.data.table

Bates, D., Maechler, M., & Jagan, M. (2026). *Matrix: Sparse and Dense Matrix Classes and Methods* [R package version 1.7-5]. https://doi.org/10.32614/CRAN.package.Matrix

Bengtsson, H. (2025). *matrixStats: Functions that Apply to Rows and Columns of Matrices (and to Vectors)* [R package version 1.5.0]. https://doi.org/10.32614/CRAN.package.matrixStats

Burns, P. (2011). *The R Inferno*. https://www.burns-stat.com/pages/Tutor/R_inferno.pdf

Casella, G., & Berger, R. L. (2002). *Statistical Inference*. Duxbury Pacific Grove, CA.

Chatzipantsiou, C., Dimitriadis, M., Papadakis, M., & Tsagris, M. (2019). Extremely efficient permutation and bootstrap hypothesis tests using R. *Journal of Modern Applied Statistical Methods*, *18*(2), eP2898. https://doi.org/10.22237/jmasm/1604189940

Chirico, M., & Oehlschlägel, J. (2026). *bit64: A S3 Class for Vectors of 64bit Integers* [R package version 4.8.6]. https://doi.org/10.32614/CRAN.package.bit64

Christensen, R. H. B. (2015). *ordinal—Regression Models for Ordinal Data* [R package version 2015.6-28]. http://www.cran.r-project.org/package=ordinal/

Gillespie, C., & Lovelace, R. (2016). *Efficient R programming: a practical guide to smarter programming*. O'Reilly Media, Inc.

Hester, J., & Vaughan, D. (2025). *bench: High Precision Timing of R Expressions* [R package version 1.1.4]. https://doi.org/10.32614/CRAN.package.bench

Krantz, S. (2026). collapse : Advanced and Fast Statistical Computing and Data Transformation in *R*. *Journal of Statistical Software*, *116*(1). https://doi.org/10.18637/jss.v116.i01

Mersmann, O. (2015). *microbenchmark: Accurate Timing Functions* [R package version 1.4-2.1]. http://CRAN.R-project.org/package=microbenchmark

Nychka, D., Furrer, R., & Sain, S. (2015). *fields: Tools for Spatial Data* [R package version 8.2-1]. http://CRAN.R-project.org/package=fields

Papadakis, M., Tsagris, M., Dimitriadis, M., Fafalios, S., Fasiolo, M., Jacob, M., Borboudakis, G., Burkardt, J., & Zou, C. (2025). *Rfast: A Collection of Efficient and Extremely Fast R Functions* [R package version 2.1.5.2]. http://CRAN.R-project.org/package=Rfast

Papadakis, M., Tsagris, M., Fafalios, S., Dimitriadis, M., Lasithiotakis, M., & Kontemeniotis, N. (2026). *Rfast2: A Collection of Efficient and Extremely Fast R Functions II* [R package version 0.1.5.6]. http://CRAN.R-project.org/package=Rfast2

Presnell, B., Morrison, S. P., & Littell, R. C. (1998). Projected Multivariate Linear Models for Directional Data. *Journal of the American Statistical Association*, *93*(443), 1068–1077. https://doi.org/10.1080/01621459.1998.10473768

Schmidt, D. (2025). *float: 32-Bit Floats* [R package version 0.3-3]. https://doi.org/10.32614/CRAN.package.float

Strang, G. (2023). *Introduction to Linear Algebra (6th edition)*. Wellesley-Cambridge Press, USA.

Tsagris, M. (2017). Conditional Independence test for categorical data using Poisson log-linear model. *Journal of Data Science*, *15*(2), 345–354. https://arxiv.org/pdf/1706.02046.pdf

Tsagris, M., & Papadakis, M. (2018). *Taking R to its limits: 70+ tips*. PeerJ. https://doi.org/10.7287/peerj.preprints.26605v1

Tsagris, M., & Papadakis, M. (2026a). *estats: Fast and Light-Weight Energy Statistics* [R package version 1.1].

Tsagris, M., & Papadakis, M. (2026b). Fast and light-weight energy statistics using the *R* package estats. *Journal of Statistical Computation and Simulation*, *96*(7), 1589–1605. https://doi.org/10.1080/00949655.2025.2601701

Venables, W. N., & Ripley, B. D. (2002). Modern Applied Statistics with S. In *Statistics and Computing*. Springer New York. https://doi.org/10.1007/978-0-387-21706-2

Hang Weiqiang <e0010758@u.nus.edu>. (2020). dcov: A Fast Implementation of Distance Covariance [Dataset]. In *CRAN: Contributed Packages*. The R Foundation. https://doi.org/10.32614/cran.package.dcov

Wickham, H. (2015). *Advanced R*. CRC Press.

Wickham, H., Hester, J., & Bryan, J. (2026). *readr: Read Rectangular Text Data* [R package version 2.2.0]. https://doi.org/10.32614/CRAN.package.readr

Yee, T. W. (2010). The VGAM Package for Categorical Data Analysis. *Journal of Statistical Software*, *32*(10). https://doi.org/10.18637/jss.v032.i10

COLOPHON Typeset with Typst 0.15.0 from the authors' LaTeX source. Template jrepstat.typ at commit 6dc131e. Set in STIX Two Text, STIX Two Math, Public Sans and JetBrains Mono. Built 24 November 2026.