Is R really that slow?
Taking R to its very limits

Michail Tsagris and Manos Papadakis

mtsagris@yahoo.gr and papadakm95@gmail.com

Department of Computer Science, University of Crete, Herakleion, Greece

Abstract

R has many capabilities most of which are hidden, yet waiting to be discovered. For this reason we demonstrate some of them and provide tips for how to write faster codes without having to program in C++, yet using it implicitly.

Keywords: Fast, efficient, computational cost.

1 Introduction

We will show a few tips for faster computations. In small sample and or low dimensions you may see small differences, but in bigger datasets the differences arise. You might observe a time difference of only 5 seconds in the whole process. Differences from 40 down to 12 seconds for example, or to 22 seconds; still good. But not always this kind of differences. Some times, one tip gives you 1 second and then another tip 1 second and so on until you save 5 seconds. If you have 1000 simulations, then you save 5000 seconds. Even small decreases matter. Some times the speed-ups will appear in the large scale vectors, but not in the small samples. Perhaps for someone who needs a simple car, a very expensive car or a jeep type might not be of such use, especially if he or she does not go to the village or to off-road situation. But for the user who needs a jeep, every computational power, every second he/she can gain matters. The computer looks strong but bear in mind we are mostly interested in the relevant durations (which function is faster and by how much) and not the actual time per se. All the computations took place in a 64-bit desktop with an Intel Core i5-4960K CPU @ 3.5GHz processor and 32 GB RAM. This will be just another medium computer in a few years, given the technological rate of increase.

1.1 Duration of a processes

If you want to see how much time your process or computation or simulation needs, you can do the following in R

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

The elapsed is what you want. Alternatively you can download the package microbenchmark (Mersmann, 2015) which allows you to compare two or more functions measuring the time even in nanoseconds.

2 A few tips for faster implementations

2.1 Simple functions

The function *mean* is slower than *sum(x)/length(x)*. If you type *sum* you will see it is a *.Primitive* function, whereas *crossprod* and *colMeans* are both *.Internal* ones and note that *colMeans* and *colSums* are two really really fast function. Our 2 points are a) create your own functions, you will be surprised to see that you may do faster than R's built-in functions (it doesn't always work that way) and b) use *.Internal* functions whenever possible. An example of the point is the *var* function. Create your own and you will see it is faster.

Search for functions that take less time. For example, the command *lm.fit(x,y)* is a wrapper for *lm(y x)*, which means that the first one is used by the second one to give you the nice output. But, if you need only the coefficients, for example, then use the first one. The syntax is a bit different, the x must be the design matrix, but the speed is very different especially in the big cases.

Avoid using *apply* or *aggregate* we saw before whenever possible if possible. But, use *colMeans* or *colSums* instead of *apply(x, 2, mean)* to get the mean vector of a sample because it's faster. For the median though, you have to use *apply(x, 2, median)* instead of a *for* going to every column of the matrix. The for function is not slower, but the apply is knitter. However, we will get back to the median case in the end of this document.

Avoid unnecessary calculations. In a discriminant analysis setting for example there is no need to calculate constant parts, such as $\log{(2\pi)}$, every time for each group and every iteration. This only adds time and takes memory and does not affect the algorithm or the result.

Remove unnecessary parentheses as they make it harder for the compiler behind to check whether the parentheses open and close. Try to make the mathematics simple.

If you want to extract the number of rows or columns of a matrix x do not use *nrow(x)* or *ncol(x)*, but *dim(x)[1]* or *dim(x)[2]* as they are almost 2 times faster.

If you have a vector "x" and want to put it in a matrix with say 10 columns, do not write *as.matrix(x, ncol = 10)*, but *matrix(x, ncol = 10)*. The first method creates a matrix and puts the vector in. The second method, simply changes the dimension of x, instead of 1 column, it will now have 10. Again, about 2 times faster.

Instead of *log(det(A))*, you can type *determinant(A, logarithm = TRUE)* as it is slightly faster for small matrices. In the big matrices, say 100 and above the differences become negligible though.

When it comes to calculating probabilities or p-values more specifically, do not do $1 - pchisq(stat, dof)$, but do $pchisq(stat, dof, lower.tail = FALSE)$ as is a bit faster. In the tens of thousands of repetitions (simulation studies for example or an algorithm that requires p-values repeatedly), the differences become seconds.

If you take your input matrix and transpose it and never use the initial matrix in the subsequent steps it is best to delete the initial matrix, or even better store its transpose in the same object. That is, if you have a matrix x, you should do the following

```
y <- t(x)  ## wrong
x <- t(x)  ## correct
```

We repeat that this in the case when *x* is not used again in latter steps. The reason for this is memory handling. If *x* is a big and you have a second object as big as the first one, you request your computer to use extra memory for nothing. If you have a few kilos to carry and want to change the bag, you just change the bag, you do not get another bag, put the same weight there and carry two bags. Even if you have a car, it is not wise to do so, especially if the weight is many tens of kilos.

When calculating operations such as *sum(a * x)*, where *x* is a vector and *a* is a scalar, a number, do *a * sum(x)*. 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 multiplication between two numbers take place.

Suppose for example you want to calculate the factorial of some integers and most (or all) of those integers appear more than once (Poisson, beta binomial, beta geometric, negative binomial distribution for example). Instead of doing the operation for each element, do it for the unique ones and simply calculate its result by its frequency. See the example below. **Note however**, that this trick does not always work. It will work in the case where you have many integers and a *for* or a *while* loop and hence you have to calculate factorials all the time.

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

Use the command *prcomp* instead of *princomp*. The first one should be used for principal component analysis when you have matrices with more than 100 variables. The more variables the bigger (40 times for example) the difference from doing *eigen(cov(x))*.

Pre-calculate any stuff you require inside the loops or will be used more than one time.

If you are to use the *glm* or *lm* commands multiple times, then you should do

```
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 glm object. We found this tip in the win-vector blog.

When calculating $\log(1+x)$ use *log1p(x)* and not *log(1+x)* as the first one is faster. A very nice function is *tabulate*.

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

Two differences between these two are that table gives you a name with the values, but tabulate gives your only the frequencies. Hence, *tabulate(x) = as.vector(table(x))*. In addition, if you use *tabulate*, you can do so with factor variables as well. But, if you have numbers, a numerical vector, 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
```

tabulate is definitely many times faster than *table*. For discrimination algorithms, *tabulate* might be more useful, because of speed, when counting frequencies, it could be more useful as well, as it will return a 0 value if a number has a zero frequency. The drawback arises when you have negative numerical data, data with a zero or positive numbers but not consecutive. If you want speed, formulate your data to match the requirements of *tabulate*. Avoid parentheses. See the examples below and redo them to convince yourselves.

Unit: nanoseconds

expr	min	lq	mean	median	uq	max	neval	cld
a1	587	880.0	2478.23	1173	1466.0	19648	100	a
a2	12024	13489.5	16246.23	14076	19941.0	29031	100	b
a3	24633	26979.0	32404.20	31378	34750.5	59530	100	c

```
microbenchmark( a1 = for (i in 1:100) 5, a2 = for (i in 1:100) (( 5 )),
a3 = for (i in 1:100) (((( 5 )))) )
```

a1 is with no parentheses, a2 is with 2 parentheses and a3 is with 4 parentheses.

Unit: nanoseconds

expr	min	lq	mean	median	uq	max	neval	cld
a1	587	880.0	2478.23	1173	1466.0	19648	100	a
a2	12024	13489.5	16246.23	14076	19941.0	29031	100	b
a3	24633	26979.0	32404.20	31378	34750.5	59530	100	c

2.2 Using *colMeans* and *colSums*

Next, suppose you want to center some data, you can try with *apply* for example

```
data = matrix(rnorm(1000 * 10), ncol = 10 )
m <- colMeans(data)
n <- nrow(data)  ;  p <- ncol(data)

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

or using this

```
a2 <- scale(data, center = TRUE, scale = FALSE)
a3 <- sweep(data, 2L, m)
a4 <- t( t(data) - m )
a5 <- data - rep( m, rep(n, p) ) ## looks faster
```

See also Gaston Sanchez's webpage for a comparison of these. We created this example to see for ourselves.

Unit: microseconds

expr	min	lq	mean	median	uq	max	neval
a1	204.981	278.7335	329.02278	339.8760	372.5730	588.551	100
a2	100.585	130.9360	153.60703	152.6365	172.8705	276.241	100
a3	81.817	114.2205	131.29665	128.0035	142.5195	446.912	100
a4	42.228	57.3310	82.81961	81.6700	96.3325	392.368	100
a5	14.076	20.0880	36.17844	29.3255	39.5890	425.798	100

Try the same experiment with a few thousands of columns and just a few rows. You will be surprised by the number of times the third way is faster than the fourth way.

If you want to extract the mean vector of each group you can use a loop (*for* function) 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) )
a3 <- rowsum(x, ina, reorder = FALSE) / as.vector( table(ina) )  ## faster
```

We found this suggestion here suggested by Gabor Grothendieck. Using the same dataset as before we created the vector *ina* which contains 5 different distinct values, each appearing 200 times.

Unit: microseconds

expr	min	lq	mean	median	uq	max	neval
a1	2538.657	2683.8150	2904.0747	2748.6235	2959.030	5815.715	100
a2	434.595	481.2215	529.1915	495.4445	523.156	2073.564	100

For the covariances the command by could be used but the matrices are stored in a list and then you need *simplify2array* to convert the list to an array in order to calculate for example the determinant of each matrix. The for loop is faster, at least that's what we have seen in our trials.

What if you have an array with matrices and want to calculate the sum or the mean of all the matrices? The obvious answer is to use *apply(x, 1:2, mean)*. R works in a column-wise fashion and not in a row-wise fashion. Instead of the *apply* you can try *t(colSums(aperm(x)))* and *t(colMeans(aperm(x)))* for the *sum* and *mean* operations 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) ) )
```

Unit: microseconds

expr	min	lq	mean	median	uq	max	neval
a1	35173.46	39991.249	44639.0034	44563.589	47045.502	77162.324	100
a2	728.43	883.853	988.6688	928.573	1062.588	1440.146	100

2.3 Calculations involving matrices

If you want the matrix of distances, with the zeros in the diagonal and the upper triangular do not use the command *as.matrix(dist(x))* but use *dist(x, diag = TRUE, upper = TRUE)*. Suppose you want the Euclidean distance of a single vector from many others (say thousands for example). The inefficient way is to calculate the distance matrix of all points and take the row which corresponds to your vector. The efficient way is to use the Mahalanobis distance with the identity matrix and the covariance matrix.

```
x <- MASS::mvrnorm(1, numeric(50), diag( rexp(50,0.4)) ) ##  vector in $R^50$.
y <- MASS::mvrnorm(1000, numeric(50), diag( rexp(50,0.4)) ) ##  vector in $R^50$.
a1 <- dist( rbind(x, y) )  ## inefficient way
Ip <- diag(50)
a2 <- mahalanobis( y, center = x, cov = Ip, inverted = TRUE )  ## better way
```

Unit: milliseconds

expr	min	lq	mean	median	uq	max	neval
a1	99.94572	105.60132	110.94383	108.68982	112.31262	147.5095	100
a2	2.14336	2.42312	3.04217	2.60904	3.14480	29.0234	100

Can we make the above faster? The answer is yes, by avoiding the matrix multiplications. You see the matrix multiplications are performed in C++ using a *for* loop. Even though it's fast, FORTRAN can make it faster.

```
z <- y - x
a <- sqrt( colSums(z^2) )
```

Try both ways and see. Check the spatial median Section here where we have kept two functions, one with the Mahalanobis and one with the above trick. Put large data and check the time required by either function; you will be amazed.

We found this article (pages 18-20) by Douglas Bates very useful and in fact we have taken some tips from there.

Suppose **X** and **m** are a matrix and a vector and want to multiply them. There are two ways to do it.

```
sum(x %*% m)  ## a bit faster
sum(m * x)
```

Suppose you want to calculate the product of an $n \times p$ matrix $\mathbf{X}^T\mathbf{X}$ for example. The command *crossprod(X)* will do the job faster than if you do the matrix multiplication.

When working with arrays it is more efficient to have them transposed. For example, if you have K covariance matrices of dimension $p \times p$, you would create an array of dimensions $c(p, p, K)$. Make its dimensions $c(K, p, p)$. If you want for example to divide each matrix with a different scalar (number) in the first case you will have to use a for loop, whereas in the transposed case you just divide the array by the vector of the numbers you have.

```
solve(X) %*% Y  #### classical
solve(X, Y)     #### much more efficient because it does not invert the matrix X
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
```

Douglas Bates mentions, in the same article, that calculating $\mathbf{X}^T\mathbf{Y}$ in R as $t(X)\% * \%Y$ instead of *crossprod(X,Y)* causes X to be transposed twice; once in the calculation of $t(X)$ and a second time in the inner loop of the matrix product. The *crossprod* function does not do any transposition of matrices. Let us see a comparison now.

```
x = matrix( rnorm(100 * 10), ncol = 10 )
a1 = t(x) %*% x
a2 = crossprod(x)
```

```
Unit: microseconds
 expr    min     lq     mean median      uq    max neval
   a1 10.850 11.290 14.13776 11.730 14.9555 62.462   100
   a2  4.105  4.399  5.40775  4.692  5.1320 17.009   100
```

Sticking with *solve(X)*, if you want to only invert a matrix then you should use *chol2inv(chol(X))* as it is faster.

```
x = matrix( rnorm(100 * 10), ncol = 10 )
s = cov(x)
a1 = solve(s)
a2 = chol2inv( chol( s ) )
```

```
Unit: microseconds
 expr    min     lq     mean  median      uq     max neval
   a1 14.370 15.249 17.22281 15.835 16.4220 112.608   100
   a2  6.451  7.331  8.57772  7.625  8.0645  66.861   100
```

The trace of the square of a matrix tr $\left(\mathbf{A}^2\right)$ can be evaluated either via

```
sum( diag( crossprod(A) ) )
```

or faster via

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

Let us now calculate times. The second way is faster, simply because the elements of the matrix are square and then are summed. In the first way, two identical matrices are required and multiplied and then sum over the elements of the new matrix.

```
x = matrix( rnorm(100 * 100), ncol = 100 )
a1 = sum( diag( crossprod(x) ) )
a3 = sum(x^2)  ## fastest
a3 = sum(x * x)
```

```
Unit: microseconds
 expr     min       lq      mean  median       uq      max neval
   a1 378.292 400.8720 423.22317 413.481 426.6775 1277.393   100
   a2  15.249  17.8885  21.61856  19.648  22.5805   43.987   100
   a3  15.836  18.1815  32.79716  20.235  27.1255  899.688   100
```

If you want to calculate the following trace involving a matrix multiplication tr $\left(\mathbf{X}^T\mathbf{Y}\right)$ you can do either

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

or faster

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

Moving in the same spirit, suppose you want the diagonal of the crossproduct of two matrices, then do

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

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

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

The explanation for this one is that in the first case you have a matrix by matrix by vector calculations. In the second case you have a matrix by vector which is a vector and then a matrix by a vector. You do less calculations. The final tip is to avoid unnecessary and/or extra calculations and try to avoid doing calculations more than once.

As for the eigen-value decomposition, 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
```

```
Unit: microseconds
 expr      min       lq      mean    median        uq      max neval
   a1 1315.222 1340.148 1409.3404 1358.6230 1388.3875 2359.775   100
   a2  671.247  684.443  744.2076  695.0005  712.3015 1674.746   100
```

The exponential term in the multivariate normal can be either calculated using matrices or simply with the command *mahalanobis*. If you have many observations and many dimensions and or many groups, this can save you a looot of time (we have seen this).

```
x <- matrix( rnorm(1000 * 20), ncol = 20 )
m <- colMeans(x)
n <- nrow(x)
p <- ncol(x)
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
```

```
Unit: microseconds
 expr       min        lq      mean    median        uq       max neval
   a1 12566.307 13131.399 14383.944 13564.821 13777.133 37264.327   100
   a2 14018.770 14839.574 16883.285 15192.499 15802.164 40466.607   100
   a3   529.021   630.633   671.828   649.547   692.801  1334.283   100
```

2.4 Numerical optimisation

The *nlm* is much faster than *optim* for optimization purposes but *optim* is more reliable and robust. Try in your examples or cases, if they give the same results and choose. Or use first *nlm* followed by *optim*.

If you have a function for which some parameters have to be positive, do not use constrained optimization, but instead put an exponential inside the function. The parameter can take any values in the whole of *R* but inside the function its exponentiated form is used. In the end, simply take the exponential of the returned value. As for its variance use the Δ method (Casella and Berger, 2002). If you did not understand this check the MLE of the inverted Dirichlet distribution and the Dirichlet regression (ϕ parameter) here.

Speaking of Dirichlet distribution, there are two ways to estimate the parameters of this distributions. Either with the use *nlm* or via the Newton-Raphson algorithm. We did some simulations and saw the Newton-Raphson can be at least 10 times faster. The same is true for the circular regression (Presnell et al., 1998) when comparing *nlm* with the E-M algorithm as described by Presnell et al. (1998). Switching to E-M or the Newton-Raphson and not relying on the *nlm* command can save you a looot of time. If you want to write a code and you have the description of the E-M or the Newton-Raphson algorithm available, because somebody did it in a paper for example, or you can derive it yourself, then do it.

If you have an iterative algorithm, such as Newton-Raphson, E-M or fixed points and you stop when the vector of parameters does not change any further, do not use *rbind*, *cbind* or *c()*. Store only two values, *vec.old* and *vec.new*. What we mean is, do not do for example

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

So, every time keep two vectors only, not the whole sequence of vectors. The same is true for the log-likelihood or whatever you have. Unless you want a trace of how things change, then ok, keep everything. Otherwise, apart from begin faster it also helps the computer run faster since less memory is used.

2.5 Vectorisation

Vectorization is a big thing. It can save tremendous amount of time even in the small datasets. Try to avoid *for* loops by using matrix multiplications. For example, instead of

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

you can use

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

Of course, this is a very easy example, but you see my point. This one requires a lot of thinking and is not always applicable. But, if it can be done, things can be super faster. See the bootstrap correlation coefficient for example, where I have two functions, *boot.correl* with a *for* loop and *bootcor*, which is vectorised.

2.6 Parallel computing

Before we begin with the functions, we would like to say a few words about the parallel computing in R. If you have a machine that has more than 1 cores, then you can put them all to work simultaneously and speed up the process a lot. If you have tricks to speed up your code that is also beneficiary. We have started taking into account tricks to speed up my code as I have mentioned before.

Panagiotis Tzirakis (master student at the department of computer science of the university of Crete in Herakleion) has showed me how to perform parallel computing in R. He is greatly acknowledged not only by us, but also by the readers of these notes, since they will save time as well.

The idea behind is to use a library that allows parallel computing. Panayiotis suggested me the doParallel package (which uses the *foreach* package) and that is what I will use from now on. Below are some instructions on how to use the package in order to perform parallel computing. In addition, we have included the parallel computing as an option in some functions and in some others we have created another function for this purpose. So, if you do not understand the notes below, you can always see the functions throughout this text.

```
 ## requires(doParallel)
 Create a set of copies of R running in parallel and communicating
 ## over sockets.
 cl <- makePSOCKcluster(nc) ## nc is the number of cluster you
 ## want to use
 registerDoParallel(cl) ## register the parallel backend with the
 ## foreach package.
 ## Now suppose you want to run R simulations, could be
 ## R=1000 for example
 ## Divide the number of simulations to smaller equally
 ## divided chunks.
 ## Each chunk for a core.
 ba <- round( rep(R/nc, nc) )
 ## Then each core will receive a chunk of simulations
 ww <- foreach(j = 1:nc,.combine = rbind) %dopar% {
 ## see the .combine = rbind. This will put the results in a matrix.
 ## Every results will be saved in a row.
 ## So if you have matrices, make them vectors. If you have lists
 ## you want to return,
 ## you have to think about it.
 a <- test(arguments, R = ba[j], arguments)$results
 ## Instead of running your function "test" with R simulations
 ## you run it with R/nc simulations.
 ## So a stores the result of every chunk of simulations.
 return(a)
  }
stopCluster(cl) ## stop the cluster of the connections.
```

To see your outcome all you have to press is *ww* and you will see something like this

```
result.1 .....
result.2 .....
result.3 .....
result.4 .....
```

So, the object *ww* contains the results you want to see in a matrix form. If every time you want a number, the *ww* will be a matrix with 1 column. We will see more cases later on. Note that f you choose to use parallel computing for something simple, multicore analysis might take the same or a bit more time than single core analysis only because it requires a couple of seconds to set up the cluster of the cores. In addition, you might use 4 cores, yet the time is half than with 1 core. This could be because not all 4 cores work at 100% of their abilities. Of course you can always experiment with these things and see.

2.7 Efficiently written functions in R packages

The multinomial regression is offered in the package VGAM (Yee, 2010), but it also offered in the package nnet (Venables and Ripley, 2002). The implementation in the second package is much faster. The same is true for the implementation of the ordinal logistic regression in the VGAM and in the ordinal (Christensen, 2015). The latter package does it much faster. Also, the package *fields* (Nychka et al., 2015) has a function called *rdist* which is faster than the built-in *dist* in R.

Many fast functions can also be found in the package Rfast (Papadakis et al., 2016). This package contains many fast or really fast functions, either written in C++ or simply using R functions exploiting the row/colMeans and row/colSums functions. The function *colMedians* for example is much faster than *apply(x, 2, median)*. The same is true for the *colVars*. Functions for matrices, distribution fitting, utility functions and many more are there and we keep adding functions. We have also implemented regression functions as well, which can handle large sample sizes (50,000 or more, for example) efficiently. Some of the functions are solely in R so you can access them directly, but most of them are written in C++. The codes are accessible in the source files of the package.

Acknowledgements

Marco Maier from the Institute for Statistics and Mathematics in Vienna, Panagiotis Tzirakis (former master student at the computer science department in Herakleion), Giorgos Athineou (former master student at the computer science department in Herakleion), Kleio Lakiotaki (post-doc at the department of computer science in Herakleion).

References

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

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/.

Mersmann, O. (2015). *microbenchmark: Accurate Timing Functions*. R package version 1.4-2.1.

Nychka, D., Furrer, R., and Sain, S. (2015). *fields: Tools for Spatial Data*. R package version 8.2-1.

Papadakis, M., Tsagris, M., Dimitriadis, M., Tsamardinos, I., Fasiolo, M., Borboudakis, G., and Burkardt, J. (2017). *Rfast: Fast R Functions*. R package version 1.7.5.

Presnell, B., Morrison, S. P., and Littell, R. C. (1998). Projected multivariate linear models for directional data. *Journal of the American Statistical Association*, 93(443):1068–1077.

Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S*. Springer, New York, fourth edition. ISBN 0-387-95457-0.

Yee, T. W. (2010). The VGAM package for categorical data analysis. *Journal of Statistical Software*, 32(10):1–34.