12 Advanced topics

This chapter contains brief descriptions of more advanced uses of R. First, we cover more details surrounding packages. We then deal with two topics that are important for computational speed: parallelisation and matrix operations. Finally, there are some tips for how to play well with others (which in this case means using R in combination with programming languages like Python and C++).

After reading this chapter, you will know how to:

- Update and remove R packages,
- Install R packages from other repositories than CRAN,
- Run computations in parallel,
- Perform matrix computations using R, and
- Integrate R with other programming languages.

12.1 More on packages

12.1.1 Loading and auto-installing packages

The best way to load R packages is usually to use `library`, as we've done in the examples in this book. If the package that you're trying to load isn't installed, this will return an error message:

```
library("theWrongPackageName")
```

Alternatively, you can use `require` to load packages. This will only display a warning but won't cause your code to stop executing, which usually would be a problem if the rest of the code depends on the package[59]!

However, `require` also returns a `logical`: `TRUE` if the package is installed, and `FALSE` otherwise. This is useful if you want to load a package and automatically install it if it doesn't exist.

To load the `beepr` package, and install it if it doesn't already exist, we can use `require` inside an `if` condition, as in the code chunk below. If the package exists, the package will be loaded (by `require`) and `TRUE` will be returned, and otherwise `FALSE` will be returned. By using `!` to turn `FALSE` into `TRUE` and vice versa, we can make R install the package if it is missing:

```
if(!require("beepr")) { install.packages("beepr"); library(beepr) }
beep(4)
```

12.1.2 Updating R and your packages

You can download new versions of R and RStudio following the same steps as in Section 2.1. On Windows, you can have multiple versions of R installed simultaneously.

To update a specific R package, you can use `install.packages`. For instance, to update the `beepr` package, you'd run:

```
install.packages("beepr")
```

If you make a major update of R, you may have to update most or all of your packages. To update all your packages, you simply run `update.packages()`. If this fails, you can try the following instead:

```
pkgs <- installed.packages()
pkgs <- pkgs[is.na(pkgs[, "Priority"]), 1]
install.packages(pkgs)
```

12.1.3 Alternative repositories

In addition to CRAN, two important sources for R packages are Bioconductor, which contains a large number of packages for bioinformatics, and GitHub, where many developers post development versions of their R packages (which often contain functions and features not yet included in the version of the package that has been posted on CRAN).

To install packages from GitHub, you need the `devtools` package. You can install it using:

```
install.packages("devtools")
```

If you for instance want to install the development version of `dplyr` (which you can find at https://github.com/tidyverse/dplyr), you can then run the following:

```
library(devtools)
install_github("tidyverse/dplyr")
```

Using development versions of packages can be great, because it gives you the most up-to-date version of packages. Bear in mind that they are development versions though, which means that they can be less stable and have more bugs.

To install packages from Bioconductor, you can start by running this code chunk, which installs the `BiocManager` package that is used to install Bioconductor packages:

```
install.packages("BiocManager")
# Install core packages:
library(BiocManager)
install()
```

You can have a look at the list of packages at:

https://www.bioconductor.org/packages/release/BiocViews.html#___Software

If you for instance find the `affyio` package interesting, you can then install it using:

```
library(BiocManager)
install("affyio")
```

12.1.4 Removing packages

This is probably not something that you'll find yourself doing often, but if you need to uninstall a package, you can do so using `remove.packages`. Perhaps you've installed the development version of a package and want to remove it, so that you can install the stable version again? If you for instance want to uninstall the `beepr` package[60], you'd run the following:

```
remove.packages("beepr")
```

12.2 Speeding up computations with parallelisation

Modern computers have CPUs with multiple cores and threads, which allows us to speed up computations by performing them in parallel. Some functions in R do this by default, but far from all do. In this section, we'll have a look at how to run parallel versions of `for` loops and functionals.

12.2.1 Parallelising `for` loops

First, we'll have a look at how to parallelise a `for` loop. We'll use the `foreach`, `parallel`, and `doParallel` packages, so let's install them if you haven't already:

```r
install.packages(c("foreach", "parallel", "doParallel"))
```

To see how many cores are available on your machine, you can use `detectCores`:

```r
library(parallel)
detectCores()
```

It is unwise to use all available cores for your parallel computation – you'll always need to reserve at least one for running RStudio and other applications.

To run the steps of a `for` loop in parallel, we must first use `registerDoParallel` to *register* the parallel backend to be used. Here is an example where we create three workers (and so use three cores in parallel[61]) using `registerDoParallel`. When we then use `foreach` to create a `for` loop, these three workers will execute different steps of the loop in parallel. Note that this wouldn't work if each step of the loop depended on output from the previous step. `foreach` returns the output created at the end of each step of the loop in a `list` (Section 5.2):

```
library(doParallel)
registerDoParallel(3)

loop_output <- foreach(i = 1:9) %dopar%
{
      i^2
}

loop_output
unlist(loop_output) # Convert the list to a vector
```

If the output created at the end of each iteration is a vector, we can collect the output in a `matrix` object as follows:

```
library(doParallel)
registerDoParallel(3)

loop_output <- foreach(i = 1:9) %dopar%
{
      c(i, i^2)
}

loop_output
matrix(unlist(loop_output), 9, 2, byrow = TRUE)
```

If you have nested loops, you should run the outer loop in parallel, but not the inner loops. The reason for this is that parallelisation only really helps if each step of the loop takes a comparatively long time to run. In fact, there is a small overhead cost associated with assigning different iterations to different cores, meaning that parallel loops can be slower than regular loops if each iteration runs quickly.

An example where each step often takes a while to run is simulation studies. Let's rewrite the simulation we used to compute the type I error rates of different versions of the t-test in Section 7.2.2 using a parallel `for` loop instead. First, we define the function as in Section 7.2.2 (minus the progress bar):

```r
# Load package used for permutation t-test:
library(MKinfer)

# Create a function for running the simulation:
simulate_type_I <- function(n1, n2, distr, level = 0.05, B = 999,
                            alternative = "two.sided", ...)
{
      # Create a data frame to store the results in:
      p_values <- data.frame(p_t_test = rep(NA, B),
                             p_perm_t_test = rep(NA, B),
                             p_wilcoxon = rep(NA, B))

      for(i in 1:B)
      {
            # Generate data:
            x <- distr(n1, ...)
            y <- distr(n2, ...)

            # Compute p-values:
            p_values[i, 1] <- t.test(x, y,
                                alternative = alternative)$p.value
            p_values[i, 2] <- perm.t.test(x, y,
                                alternative = alternative,
                                R = 999)$perm.p.value
            p_values[i, 3] <- wilcox.test(x, y,
                                alternative = alternative)$p.value
      }

      # Return the type I error rates:
      return(colMeans(p_values < level))
}
```

Next, we create a parallel version:

```r
# Register parallel backend:
library(doParallel)
registerDoParallel(3)

# Create a function for running the simulation in parallel:
simulate_type_I_parallel <- function(n1, n2, distr, level = 0.05,
                                     B = 999,
                                     alternative = "two.sided", ...)
{

      results <- foreach(i = 1:B)  %dopar%
      {
            # Generate data:
            x <- distr(n1, ...)
            y <- distr(n2, ...)

            # Compute p-values:
            p_val1 <- t.test(x, y,
                                  alternative = alternative)$p.value
            p_val2 <- perm.t.test(x, y,
                                  alternative = alternative,
                                  R = 999)$perm.p.value
            p_val3 <- wilcox.test(x, y,
                                  alternative = alternative)$p.value

            # Return vector with p-values:
            c(p_val1, p_val2, p_val3)
      }

      # Each element of the results list is now a vector
      # with three elements.
      # Turn the list into a matrix:
      p_values <- matrix(unlist(results), B, 3, byrow = TRUE)

      # Return the type I error rates:
      return(colMeans(p_values < level))
}
```

We can now compare how long the two functions take to run using the tools from Section 6.6 (we'll not use `mark` in this case, as it requires both functions to yield identical output, which won't be the case for a simulation):

```
time1 <- system.time(simulate_type_I(20, 20, rlnorm,
                                      B = 999, sdlog = 3))
time2 <- system.time(simulate_type_I_parallel(20, 20, rlnorm,
                                               B = 999, sdlog = 3))

# Compare results:
time1; time2; time2/time1
```

As you can see, the parallel function is considerably faster. If you have more cores, you can try increasing the value in `registerDoParallel` and see how that affects the results.

12.2.2 Parallelising functionals

The `parallel` package contains parallelised versions of the `apply` family of functions, with names like `parApply` , `parLapply` , and `mclapply` . Which of these you should use depends in part on your operating system, as different operating systems handle multicore computations differently. Here is the first example from Section 6.5.3, run in parallel with three workers:

```
# Non-parallel version:
lapply(airquality, function(x) { (x-mean(x))/sd(x) })

# Parallel version for Linux/Mac:
library(parallel)
mclapply(airquality, function(x) { (x-mean(x))/sd(x) },
         mc.cores = 3)

# Parallel version for Windows (a little slower):
library(parallel)
myCluster <- makeCluster(3)
parLapply(myCluster, airquality, function(x) { (x-mean(x))/sd(x) })
stopCluster(myCluster)
```

Similarly, the `furrr` package lets us run `purrr` functionals in parallel. It relies on a package called `future`. Let's install them both:

```
install.packages(c("future", "furrr"))
```

To run functionals in parallel, we load the `furrr` package and use `plan` to set the number of parallel workers:

```
library(furrr)
# Use 3 workers:
plan(multisession, workers = 3)
```

We can then run parallel versions of functions like `map` and `imap`, by using functions from `furrr` with the same names, only with `future_` added at the beginning. Here is the first example from Section 6.5.3, run in parallel:

```
library(magrittr)
airquality %>% future_map(~(.-mean(.))/sd(.))
```

Just as for `for` loops, parallelisation of functionals only really helps if each iteration of the functional takes a comparatively long time to run (and so there is no benefit to using parallelisation in this particular example).

12.3 Linear algebra and matrices

Linear algebra is the beating heart of many statistical methods. R has a wide range of functions for creating and manipulating matrices, and doing matrix algebra. In this section, we'll have a look at some of them.

12.3.1 Creating matrices

To create a `matrix` object, we can use the `matrix` function. It always coerces all elements to be of the same type (Section 5.1):

```
# Create a 3x2 matrix, one column at a time:
matrix(c(2, -1, 3, 1, -2, 4), 3, 2)

# Create a 3x2 matrix, one row at a time:
# (No real need to include line breaks in the vector with
# the values, but I like to do so to see what the matrix
# will look like!)
matrix(c(2, -1,
          3, 1,
         -2, 4), 3, 2, byrow = TRUE)
```

Matrix operations require the dimension of the matrices involved to match. To check the dimension of a `matrix`, we can use `dim`:

```
A <- matrix(c(2, -1, 3, 1, -2, 4), 3, 2)
dim(A)
```

To create a unit matrix (all 1s) or a zero matrix (all 0s), we use `matrix` with a single value in the first argument:

```
# Create a 3x3 unit matrix:
matrix(1, 3, 3)

# Create a 2x3 zero matrix:
matrix(0, 2, 3)
```

The `diag` function has three uses. First, it can be used to create a diagonal matrix (if we supply a vector as input). Second, it can be used to create an identity matrix (if we supply a single number as input). Third, it can be used to extract the diagonal from a square matrix (if we supply a matrix as input). Let's give it a go:

```r
# Create a diagonal matrix with 2, 4, 6 along the diagonal:
diag(c(2, 4, 6))

# Create a 9x9 identity matrix:
diag(9)

# Create a square matrix and then extract its diagonal:
A <- matrix(1:9, 3, 3)
A
diag(A)
```

Similarly, we can use `lower.tri` and `upper.tri` to extract a matrix of `logical` values, describing the location of the lower and upper triangular part of a matrix:

```r
# Create a matrix_
A <- matrix(1:9, 3, 3)
A

# Which are the elements in the lower triangular part?
lower.tri(A)
A[lower.tri(A)]

# Set the lower triangular part to 0:
A[lower.tri(A)] <- 0
A
```

To transpose a matrix, use `t` :

```r
t(A)
```

Matrices can be combined using `cbind` and `rbind` :

```
A <- matrix(c(1:3, 3:1, 2, 1, 3), 3, 3, byrow = TRUE)  # 3x3
B <- matrix(c(2, -1, 3, 1, -2, 4), 3, 2)               # 3x2

# Add B to the right of A:
cbind(A, B)

# Add the transpose of B below A:
rbind(A, t(B))

# Adding B below A doesn't work, because the dimensions
# don't match:
rbind(A, B)
```

12.3.2 Sparse matrices

The `Matrix` package contains functions for creating and speeding up computations with sparse matrices (i.e., matrices with lots of 0s), as well as for creating matrices with particular structures. You likely already have it installed, as many other packages rely on it. `Matrix` distinguishes between sparse and dense matrices:

```
# Load or/and install Matrix:
if(!require("Matrix")) { install.packages("Matrix"); library(Matrix) }

# Create a dense 8x8 matrix using the Matrix package:
A <- Matrix(1:64, 8, 8)

# Create a copy and randomly replace 40 elements by 0:
B <- A
B[sample(1:64, 40)] <- 0
B

# Store B as a sparse matrix instead:
B <- as(B, "sparseMatrix")
B
```

To visualise the structure of a sparse matrix, we can use `image`:

```
image(B)
```

An example of a slightly larger, 72×72 sparse matrix is given by `CAex`:

```
data(CAex)
CAex
image(CAex)
```

`Matrix` contains additional classes for, e.g., symmetric sparse matrices and triangular matrices. See `vignette("Introduction", "Matrix")` for further details.

12.3.3 Matrix operations

In this section, we'll use the following matrices and vectors to show how to perform various matrix operations:

```
# Matrices:
A <- matrix(c(1:3, 3:1, 2, 1, 3), 3, 3, byrow = TRUE)  # 3x3
B <- matrix(c(2, -1, 3, 1, -2, 4), 3, 2)               # 3x2
C <- matrix(c(4, 1, 1, 2), 2, 2)             # Symmetric 2x2

# Vectors:
a <- 1:9                     # Length 9
b <- c(2, -1, 3, 1, -2, 4) # Length 6
d <- 9:1                     # Length 9
y <- 4:6                     # Length 3
```

To perform element-wise addition and subtraction with matrices, use `+` and `-`:

```
A + A
A - t(A)
```

To perform element-wise multiplication, use `*`:

```r
2 * A  # Multiply all elements by 2
A * A  # Square all elements
```

To perform matrix multiplication, use `%*%`. Remember that matrix multiplication is non-commutative, and so the order of the matrices is important:

```r
A %*% B  # A is 3x3, B is 3x2
B %*% C  # B is 3x2, C is 2x2
B %*% A  # Won't work, because B is 3x2 and A 3x3!
```

Given the vectors `a`, `b`, and `d` defined above, we can compute the outer product $a \otimes b$ using `%o%` and the dot product $a \cdot d$ by using `%*%` and `t` in the right manner:

```r
a %o% b  # Outer product
a %*% t(b) # Alternative way of getting the outer product
t(a) %*% d # Dot product
```

To find the inverse of a square matrix, we can use `solve`. To find the generalised Moore-Penrose inverse of any matrix, we can use `ginv` from `MASS`:

```r
solve(A)
solve(B)  # Doesn't work because B isn't square

library(MASS)
ginv(A)  # Same as solve(A), because A is non-singular and square
ginv(B)
```

`solve` can also be used to solve equations systems. To solve the equation $Ax = y$:

```r
solve(A, y)
```

The eigenvalues and eigenvectors of a square matrix can be found using `eigen`:

```r
eigen(A)
eigen(A)$values   # Eigenvalues only
eigen(A)$vectors  # Eigenvectors only
```

The singular value decomposition, QR decomposition, and the Choleski factorisation of a matrix are computed as follows:

```r
svd(A)
qr(A)
chol(C)
```

`qr` also provides the rank of the matrix:

```r
qr(A)$rank
qr(B)$rank
```

Finally, you can get the determinant[62] of a matrix using `det`:

```r
det(A)
```

As a P.S., I'll also mention the `matlab` package, which contains functions for running computations using MATLAB-like function calls. This is useful if you want to reuse MATLAB code in R without translating it row-by-row. Incidentally, this also brings us nicely into the next section.

12.4 Integration with other programming languages

R is great for a lot of things, but it is obviously not the best choice for every task. There are a number of packages that can be used to harvest the power of other languages, or to integrate your R code with code that you or others have developed in other programming languages. In this section, we'll mention a few of them.

12.4.1 Integration with C++

C++ is commonly used to speed up functions, for instance involving loops that can't be vectorised or parallelised due to dependencies between different iterations. The `Rcpp` package (Eddelbuettel & Balamuta, 2018) allows you to easily call C++ functions from R, as well as calling R functions from C++. See `vignette("Rcpp-introduction", "Rcpp")` for details.

An important difference between R and C++ that you should be aware of is that the indexing of vectors (and similar objects) in C++ starts with 0. So the first element of the vector is element 0, the second is element 1, and so forth. Bear this in mind if you pass a vector and a list of indices to C++ functions.

12.4.2 Integration with Python

The `reticulate` package can be used to call Python functions from R. See

```
vignette("calling_python", "reticulate")
```

for some examples.

Some care has to be taken when sending data back and forth between R and Python. In R `NA` is used to represent missing data and `NaN` (not a number) is used to represent things that should be numbers but aren't (e.g., the result of computing `0/0`). Perfectly reasonable! However, for reasons unknown to humanity, popular Python packages like Pandas, NumPy and SciKit-Learn use `NaN` instead of `NA` to represent missing data – but only for `double` (`numeric`) variables. `integer` and `logical` variables have no way to represent missing data in Pandas. Tread gently if there are `NA` or `NaN` values in your data.

Like in C++, the indexing of vectors (and similar objects) in Python starts with 0.

12.4.3 Integration with Tensorflow and PyTorch

Tensorflow, Keras, and PyTorch are popular frameworks for deep learning. To use Tensorflow or Keras with R, you can use the `keras` package. See `vignette("index", "keras")` for an introduction and Chollet & Allaire (2022) for a thorough treatise. Similarly, to use PyTorch with R, use the `torch` package. In both cases, it can take some tampering to get the frameworks to run on a GPU.

12.4.4 Integration with Spark

If you need to process large datasets using Spark, you can do so from R using the `sparklyr` package. It can be used both with local and cloud clusters, and (as the name seems to imply) it is easy to integrate with `dplyr`.

59. And why else would you load it…?↩

60. Why though?!↩

61. If your CPU has three or fewer cores, you should lower this number.↩

62. Do you *really* need it, though?↩