Vectorization

> Talk is cheap. Show me the code.
> --- Linus Torvalds

While we already intesively talked about loops we have experienced and discussed some of their drawbacks. Especially in R loops are incomprehensible slow. Additionally they require to handle index or iterator values, which may lead to confusion. Luckily R provides an alternative that eradicates these problems and allows to eventually deprecated the usage of loops in R in favor of vectorized functions. If you understand the nuts and bolts of vectorization in R, the following functions allow you to write shorter, simpler, safer, and faster code.

*apply-Family

In the previous sections we already introduced some vectorized functions without explicitly metioning their rationale and that they are vectorized. The following functions all belong to the so called *apply-Family. Vectorization in R requires a thorough understanding of the available data structures, as the following functions iterate (automatically) over different slices of data strucutes and perform (loop-wise) repetitions on the data slices from vectors, matrices, arrays, lists and dataframes. More specifically, the family consists of `apply()`, `lapply()`, `sapply()`, `vapply()`, `tapply()`, `mapply()`, `rapply()`, and `eapply()`. The following snippet gives a very short overview over the functions we are going to discuss in more detail. All of these functions are provided by the R base system and are thus located in the library `base`.

```{}
# The *apply-Family of Functions in the Library base
base::apply     Apply Functions Over Array Margins
base::lapply    Apply a Function over a List or Vector
base::sapply    Simplified Version of lapply
base::vapply    Safer Version of sapply
base::tapply    Apply a Function Over a Ragged Array
base::mapply    Apply a Function to Multiple List or Vector Arguments
base::rapply    Recursively Apply a Function to a List
base::eapply    Apply a Function Over Values in an Environment
```

apply {-}
We already introduced `apply` and used it to apply a function to the rows or columns of a matrix, in the same fashion as functions like `rowMeans` or `colMeans` calculate a specific values for either a row or a column of a matrix. Generally speaking apply operates on (two dimensional) arrays, a.k.a. matrices. To get started we create a sample data set consisting of a matrix with 20 cells partitioned into five rows and four columns.

```{r}
mat <- matrix(c(1:10, 21:30), nrow = 5, ncol = 4)
mat
```

To mimic the functionality of `rowSums`, we now can use the apply function to find the sum over all elements of each row as follows.

```{r}
```

```
apply(X=mat, MARGIN=1, FUN=sum)
```

Notice that the funciton call to `apply` takes three arguments, where `X` is the data, `MARGIN` corresponds eiter to the rows as they are the first dimension of the data or to the columns, which correspond to the second dimension. `FUN` is the funciton that should be applied on the specified margin of the data. Note that the function in the snippet below is passed without parantheses (`sum` instead of `sum()`).

Remember that in R everything is a vector. Therefore, a matrix can be seen as a collection of line vectors when you cross the matrix from top to bottom (along `MARGIN=1`), or as a list of column vectors, spanning the matrix left to right (along `MARGIN=2`). The code in the above R chunk therefore translates directo to the instruction to: "apply the function sum to the matrix `mat` along the rows". Ursuprisingly this leads to a vector containing the sums of the values of each row. Mathematically speaking we would expect a column vector here, while R outputs a line vector. As R does not differentiate here while outputting these on the console this makes no difference for this case. The following picture illustrates the process.

```{r, echo=FALSE, fig.cap=NULL, out.width="70%"}
#knitr::include_graphics("gfx/CH06-ApplyExample.png")
```

lapply {-}
While matrices are an important and often used data structure they are not the only one. Quite often data comes as `list` and it may be a reasonable purpose to apply a function to every (sub-) element of a given list. As lists have no dimensions (see `dim`), the application of `apply` fails. Thefeore if you want to apply a specific function to every element of a list you have to use a list compatible version of apply, the `lapply`-function. The syntax is quite comparable to our usual `apply`, which can be seen when executing `?lapply`.

Using our toy example with the previously introduced matrix, we construct a list by spliting `mat` by row. Applying the function `sum` to this list should now result in the same values as the previous application of `apply` on `mat`.

```{r}
lst <- split(mat, 1:nrow(mat))   # Split mat by row
lst
```

Due to the flexibility and ubiquitousness of lists, `lapply` can be widely used and e.g. also works on dataframes in addition to lists. Additionally it is compatible with vectors, where the second most important part about `lapply`comes into place. Regardless if the data input `X` is a list, a dataframe or a vector, the returned data is always a `list`, which can be seen in the code below.

```{r}
lapply(X=lst, FUN=sum)
```

The following image shows the proces and illustrates how `lapply` works. As seen above the results are identical to the ones delivered by `apply` and the returned data structure is (as expected) a list.

```{r, echo=FALSE, fig.cap=NULL, out.width="50%"}
#knitr::include_graphics("gfx/CH06-LapplyExample.png")
```

sapply {-}
The function `sapply` takes the same inputs and behaves in the exact same mannor as `lapply`, but tries to simplify the result so that it returns an appropriate data structure instead of always returning a list. Applied to our example from above `sapply` returns a numeric vector.

```{r}
sapply(X=lst, FUN=sum)
```

`sapply` can be fored to behave exactly like lapply and also return a list by setting the argument `simplify` to `FALSE`.

```{r}
res1 <- lapply(X=lst, FUN=sum)
res2 <- sapply(X=lst, FUN=sum, simplify = FALSE)
identical(res1, res2)
class(res2)   # sapply also returns a list if forced to act like lapply
```

The simplification performed by `sapply` can also be applied manually. R offers the commands `unlist` or `simplify2array`, that perform similar simplification operations. The code below shows that results obtained by `lapply` are identical to the ones generated by `sapply`, after passing them to the function `simplify2array`.

```{r}
res3 <- sapply(X=lst, FUN=sum)
res4 <- simplify2array(lapply(X=lst, FUN=sum))
identical(res3, res4)
```

While the manual simplification is possible, it should be strictly avoided. Using the built in capabilities of the functions makes the code more readable and may be more robust. It may cover additional cases that may not be covered when own functions or manual processes are used and therefore does not fail surprisingly.

vapply {-}
`vapply` is similar to `sapply` and therefore somehow identical to `lapply`, but it requires to specify what type of data is expected as return value. Therefore `vapply` supports the additional argument `FUN.VALUE`, that allows to specify the expected return value. For the example used above we expect `l/s/vapply` to return a single numeric value for each list value, therefore `FUN.VALUE = numeric(1)`.

```{r}
vapply(X=lst, FUN=sum, FUN.VALUE=numeric(1))
```

If the value specified by `FUN.VALUE` and the actual value returned by `vapply` do not match an error is returned.

```{r, error=TRUE}
vapply(X=lst, FUN=sum, FUN.VALUE=character(1))
vapply(X=lst, FUN=sum, FUN.VALUE=numeric(2))
```

Deciding which of these three functions `lapply`, `sapply` or `vapply` to use is obviously highly dependent on the context. While `lapply` always provides consistent results, the usage of `sapply` often helps to avoid annoiying transformations. When the input suffers from some inconsistencies `vapply` is the way to go, as it easily allows for checking special data types or even more complex data structures.

tapply {-}
While every considered `*apply`-function up to now only supports one data input, `tapply` supports two of them, where the additional argument resembles an `INDEX` or grouping variable. `tapply` splits the provided data by the grouping values and applies the specified function to these created groups. The values for `INDEX` can be constructed based on factor levels, which means the provided values need to be a factor or must work when (automatically) coerced to a factor.

The following code transforms our sample data `mat` from the other examples to a data frame with three columns. The first columns conatins the actual values that we previously found within `mat`, the second column indicates the column index where a specific `value` was placed in `mat` and the thrid coolumn inicates the row index, where the `value` was located.

```{r}
df <- data.frame(value=as.vector(mat),    # Transform example mat into a
data
                 row=rep(1:5, times=4),   # frame that contains row and
column
                 col=rep(1:5, each=4))    # indices in addition to the
value.

head(df, n=3) # First 3 rows of the data frame
tail(df, n=3) # Last 3 rows of the data frame
```

The value in the top left corner (first row and first column) of `mat` is 1. Corresponding column indices are therefore `col=1` and `row=1`. This tupel forms the first row of the created dataframe `df`. All other values from `mat` are handled in the same way. As `mat` is a 5 by 4 matrix, we get a dataframe with 20 rows.

We can now use `tapply`, the values from within `mat` and one of the column or row indices as grouping `INDEX` to calculate the `sum` of all values that belong to a specified group. Previously we calculated the sum of all values from a specified row. To obtain these exact same results, we can use the following function call.

```{r}
tapply(X=df$value, INDEX = df$row, FUN = sum)
```

<!-- The following figure illustrates the process of the previous function call. -->

<!-- ```{r, echo=FALSE, fig.cap=NULL, out.width="50%"} -->
<!-- #knitr::include_graphics("gfx/CH06-TapplyExample.png") -->
<!-- ``` -->

mapply {-}
`mapply` stands for 'multivariate' apply. Its purpose is to be able to vectorize arguments to a function that is not usually accepting vectors as arguments. `mapply` applies a function to multiple Input arguments. The Inputs can either be lists or vectors. For a small example we define the following three vectors.

```{r}
n <- rep(x = 15, n = 10)
m <- rep(x =  0, n = 10)
s <- seq(from = 1, to = 100, length.out = 10)
```

The following example generates $n = 15$ (`n`) normally distributed random numbers with mean $\mu = 0$ (`m`) and varying standard deviation σ (`s`).While the arguments for `n` are for all iterations equal to `r n[1]` and for `m` equal to `r m[1]` the values for `s` differ for each group of `r n[1]` numbers.

```{r}
y <- mapply(FUN = rnorm, n = n, mean = m, sd = s)
colnames(paste0("s=",s))
round(y, digits = 4)
```
The outcome is illustated using the following plot.

```{r}
plot(NA, ylim = range(y), xlim=c(1,10),
     ylab = "Random Number",
     xaxt = 'n', xlab= "",
     main = "Normally Distributed Random Numbers")
     mtext("with varying Standard Deviation")

z <- sapply(1:10,
       function(x,y){points(x = rep(x,length(y[,x])),
                            y = y[,x])},
       y = y)
```

`mapply`can be used to behave like nested `for` loops as the iterators can simply be anticipated and all iterator combinatinos can be generated in advance, e.g. with `expand.grid`. The following code shows how to iterate over all cells of a $5 \times 10 $ matrix using `mapply`.

```{r}
nrow <- 5
ncol <- 10
m <- matrix(1:(nrow*ncol), nrow = nrow, ncol = ncol)
ic <- expand.grid(1:nrow, 1:ncol)
afun <- function(r,c,m){m[r,c]^2}
```

```
mapply(afun, r = ic[,1], c = ic[,2], MoreArgs = list(m))
```

rapply {-}

`rapply` is recursive version of `lapply`, that applies a function recursively to every list element.

```{r}
l <- list(a = list(a1 = 1:10 , a2 = 20:30),
          b = list(b1 = 1:100, b2 = 5))

rapply(object = l, f = sum)
```

eapply {-}

`eapply` applies a function to every (visible = not hidden) element in an environment.

```{r}
env <- new.env()
env$a <- 1:10
env$b <- exp(-3:3)
env$c <- c(TRUE, FALSE, FALSE, TRUE)

eapply(env, mean)
```

Technical Background

The paradigm of vectorization is closedly linked to functional programming and one can find a few reasons (e.g. execution speed in some cases), to make use of them. A very comparable set of functions an some additional patterns are implementd in the `purrr` package that is part of the `tidyverse`.

Parallelization using vectorized functions

A very convenient method of parallelization is by utilizing the `parallel` package that provides parallelized versions of functions from the `*apply`-family. However as it uses a technique called *forking* the usage of these function is limited to operating systems that are based on Unix such as Linux or MacOS.

```{r}
library(parallel)
num.cores <- detectCores()
num.cores
```

As you can see this document was generated on a computer with `r detectCores()` computing cores. Depending on the specific processor these may either be real cores or logical cores. The number only differs if a real core can handle two tasks (threads) at once and thus behaves somewhat like two actual cores. In the case of my computer the CPU has `r detectCores(logical = F)` real cores.

```{r}
regnames <- names(mtcars[,-1])
regoptions <- rep(list(c(T,F)),length(regnames))
regselector <- as.matrix(expand.grid(regoptions))
colnames(regselector) <- regnames
```

```{r}
modellist <- apply(regselector,1,function(x){as.formula(paste(c("mpg ~ 1",
regnames[x]), collapse = " + "))})
```

```{r}
system.time(res.sequential <- lapply(modellist, function(f)
{lm(f,data=mtcars)}))
system.time(res.parallel <- mclapply(modellist, function(f)
{lm(f,data=mtcars)},mc.cores=num.cores))

```

```{r}
rsquared.sequential <- sapply(res.sequential, function(x){summary(x)
$r.squared})
rsquared.parallel <- sapply(res.parallel, function(x){summary(x)
$r.squared})
table(rsquared.sequential == rsquared.parallel)
```

```{r}
summary(rsquared.sequential)
summary(rsquared.parallel)
```

<!-- ## Parallelization using foreach -->

<!-- ```{r} -->
<!-- library(foreach) -->
<!-- library(doParallel) -->
<!-- ``` -->

<!-- ```{r} -->
<!-- for (idx in 1:3) { -->
<!--   print(sqrt(idx)) -->
<!-- } -->
<!-- ``` -->

<!-- ```{r} -->
<!-- library(foreach) -->
<!-- foreach (i=1:3) %do% { -->
<!--   sqrt(i) -->
<!-- } -->
<!-- ``` -->

<!-- ```{r} -->
<!-- registerDoParallel(num.cores) -->
<!-- system.time({ -->
<!--   res.dopar <- foreach (idx=1:length(modellist), .combine = c)
%dopar% { -->
<!--   list(lm(modellist[[idx]], data=mtcars)) -->
<!--   } -->
<!-- }) -->
<!-- ``` -->

<!-- ```{r} -->
<!-- rsquared.dopar <- sapply(res.dopar, function(x){summary(x)
$r.squared}) -->
<!-- table(rsquared.dopar == rsquared.parallel) -->
<!-- summary(rsquared.dopar) -->
<!-- ``` -->

Exercises {-}

```{r, echo=FALSE, fig.cap=NULL, out.width="100%"}
knitr::include_graphics("gfx/CH00-ExercisesHeader.png")
```

<!--
For example, in the mtcars dataset:

```{r}
library(datasets)
tapply(mtcars$wt, mtcars$cyl, mean)
```
The tapply function first groups the cars together based on the number of cylinders they have, and then calculates the mean weight for each group.
-->