Statistical Computing Chapter_3

Henry Bourne

2022-10-19

Common R

In this chapter of the portfolio we will go through some common R functions including how to leverage vectorization, the R apply family, map, reduce, filter and how to do use parallel programming.

Vectorisation

When we use vectorization the same operation is applied to every number in a vector, vectorization is important as vector operations are much faster. The reason that using vectorization is faster is that vector operations are written in c++ which works much more quickly than in R. eg. lets say we want to sum together two lists of numbers l1 and l2:

```
l1 <- c(1, 2, 3, 4, 5)
l2 <- c(6, 7, 8, 9, 10)
# Without vectorization we would write:
out <- 0
for (i in 1:length(l1)) {
    out <- out + l1[i] + l2[i]
}
out
```

```
## [1] 55
```

```
# With vectorization we write:
out <- sum(l1, l2)
out
```

```
## [1] 55
```

R apply family, sweep, map, reduce and filter

The apply family consists of: apply(), lapply(), sapply(), and mapply(). The idea of this family of functions is to apply the same function to margins of an array or matrix and return a vector or array or list of values.

Starting with **apply()**; it takes three arguments: X, MARGIN, FUN. Where **X** is an array (dimension ≥ 2), **MARGIN** is a vector giving the dimensions the function will be applied over (eg. for a matrix: 1 indicates rows (output same length as number of rows), 2 indicated columns (output same as number of columns) and c(1,2) indicates rows and columns (output same dimension as matrix applied to)) and **FUN** is the function to be applied. eg. for a matrix of all 2's and applying the sum function to the rows:

```
x <- matrix(2, nrow = 2, ncol = 2)
apply(x, 1, sum)
```

```
## [1] 4 4
```

Can see that apply has returned a vector of the sum of the columns, what about if we apply sum to the whole matrix?

```
x <- matrix(2, nrow = 2, ncol = 2)
apply(x, c(1, 2), sum)
```

```
##      [,1] [,2]
## [1,]    2    2
## [2,]    2    2
```

Can see it has performed $sum(x_{i,j}), \forall i, j \in \{1, 2\}$, which is just the original matrix, to see this more clearly lets apply sqrt(),

```
apply(x, c(1, 2), sqrt)
```

```
##          [,1]     [,2]
## [1,] 1.414214 1.414214
## [2,] 1.414214 1.414214
```

Where apply is for use on data with dimension >1, **lapply()** is used to apply a function to each element in a list or vector and outputs a list. eg.

```
lapply(c(1, 2, 3, 4, 5), exp)
```

```
## [[1]]
## [1] 2.718282
##
## [[2]]
## [1] 7.389056
##
## [[3]]
## [1] 20.08554
##
## [[4]]
## [1] 54.59815
##
## [[5]]
## [1] 148.4132
```

sapply() works like lapply() but returns a more user-friendly (less complicated) output

```
sapply(c(1, 2, 3, 4, 5), exp)
```

```
## [1]   2.718282   7.389056  20.085537  54.598150 148.413159
```

The multivariate version of sapply() is **mapply()**, which we use when we have a function that has multiple arguments, it uses the first elements of each input as arguments for the function and then the second elements and so on... . We first give it the function and then all of our inputs, eg. lets decompose x from earlier and use mapply on its row vectors:

```
mapply(sum, x[1, ], x[2, ])
```

```
## [1] 4 4
```

Lets now look at Map, Reduce and Filter. But before we do note that we can define a function in R by writing *{function (<arguments>) <some computation>}*, this is called an anonymous function. We start with the **Map()** function, which takes a function and n vectors as input, where n is the number of arguments that the function takes. It then applies the function in the same way mapply does, in fact Map is a wrapper of mapply with SIMPLIFY=FALSE, ie. it will not simplify the output to a vector. eg. doing the same before as we did with mapply:

```r
Map(sum, x[1, ], x[2, ])
```

```
## [[1]]
## [1] 4
##
## [[2]]
## [1] 4
```

Reduce() takes a binary function (takes two arguments) and one vector, it then successively combines elements of the vector (and a possible given initial value, innit) using the function. eg.

```r
Reduce(sum, c(1, 2, 3, 4, 5))
```

```
## [1] 15
```

```r
Reduce(function(a, b) a * b, c(1, 2, 3, 4, 5))
```

```
## [1] 120
```

Finally we have **Filter()** which given a unary logical function (a function that takes one argument and returns a Boolean value) and a vector, goes through the vector applying the function and returns a vector with only the values that returned true, eg.

```r
Filter(function(a) a%%2 == 0, c(1, 2, 3, 4, 5))
```

```
## [1] 2 4
```

Note: we could also give it a function which returns numbers, as 0 is treated as FALSE and any other number is treated as TRUE.

Parrallel Programming

When we want to perform large amounts of computation a single CPU core might not be enough, when we are running on a single cpu we are **cpu-bound** meaning the time for compute is determined by the speed of this single CPU. When carrying out large amounts of computation we may also find we are **memory-bound** which means the time for compute is decided by the amount o free memory. We also can be **I/O-bound** which is where time for compute is limited by the time it takes to read/write from the disk. Finally we can also be **network-bound** which is where our speed is bounded by the time to transfer information across the network.

Parallel programming aims to distribute computation across many different cores (in a single processer) or even across multiple processers. By doing this we can speed up computation by spreading the workload across multiple cores in multiple processers. This is where supercomputers get their ability for fast compute, not from having very fast processers (although they do), but from being built up of a large amount of processers. Note, however, that running n processers will not mean computation will be n times as fast, the gain in speed with increasingly adding processers tends to diminish.

In R we can use **mclapply()** (if not on windows), it works just like lapply() except that its parallelized, we can specify the number of cores we want to use by setting mc.cores. eg.

```r
library(parallel)  # We make sure that we have the parallel package installed and if not return an erro
n <- detectCores()
mclapply(c(1:10000), log, mc.cores = n)
```

There is also mcmapply() which works just like mapply() except parallelized,

```r
x <- matrix(2, nrow = 1000, ncol = 1000)
mcmapply(sum, x[1, ], x[2, ], mc.cores = n)
```

We also have **foreach** which we use semantically as below:

```
library(foreach)
foreach(i = 1:1000) %do% {
    log(i)
}
```

What foreach does is given something to iterate through and an expression to evaluate for each expression it evaluates the expression sequentially. Note, this isn't running in parallel, in order for us to do this we must register the number of cores and instead of %do% use %dopar%, in this case it will evaluate the expression for each iteration in parallel:

```
library(doParallel)  # Make sure we have the doParallel package
```

```
## Loading required package: iterators
```

```
registerDoParallel(n)  # We register the number of cores

foreach(i = 1:1000) %dopar% {
    log(i)
}
stopImplicitCluster()  # This cleans up the cluster
```

Using the above

We will now take what we've learned above and put it to use. What we would like to do is write a function for the stattools package that runs as quickly as possible. To do this we will compare different methods for carrying out the same computation and evaluate which is fastest, we will then use the fastest as our implementation for the function in the package.

The function we will be looking at is *feat_trans*, which given data, D, and dimension for feature transform, b, returns the feature transform of D by degree b. First we will write the function using a for loop:

```
feat_trans.for <- function(D, b) {
    x_ft <- matrix(NA, nrow = length(D), ncol = b)
    for (i in 1:b) {
        for (j in 1:length(D)) {
            x_ft[j, i] <- D[j]^i
        }
    }
    x_ft
}
```

Secondly we write the same function but using vectorization:

```
feat_trans.v <- function(D, b) {
    x_ft <- matrix(D, nrow = length(D), ncol = b, byrow = FALSE)
    for (i in 1:b) {
        x_ft[, i] <- x_ft[, i]^i
    }
    x_ft
}
# Note that we still have to use a for loop here
```

Thirdly we will write feat_trans using the help of apply:

```
feat_trans.apply <- function(D, b) {
    x_ft <- matrix(D, nrow = length(D), ncol = b, byrow = FALSE)
```

```
    x_ft <- t(apply(x_ft, 1, function(a) a^(1:b)))
    x_ft
}
```

Lastly we will parallelize the function using foreach:

```
feat_trans.parallel <- function(D, b) {
    require(doParallel)
    require(parallel)
    n <- detectCores()
    registerDoParallel(n)

    x_ft <- matrix(D, nrow = length(D), ncol = b, byrow = FALSE)

    x_ft <- foreach(i = 1:b, .combine = "cbind") %dopar% {
        x_ft[, i]^i
    }
    stopImplicitCluster()  # This cleans up the cluster
    x_ft
}
```

Now we will compare the performance of these functions by creating a large matrix and performing a feature transform of a large degree.

```
x <- runif(n = 1e+05, min = -1000, max = 1000)
b <- 1000

system.time(feat_trans.for(x, b))
```

```
##    user  system elapsed
##   5.064   0.188   5.251
```

```
system.time(feat_trans.v(x, b))
```

```
##    user  system elapsed
##   1.763   0.332   2.095
```

```
system.time(feat_trans.apply(x, b))
```

```
##    user  system elapsed
##   2.602   0.836   3.439
```

```
system.time(feat_trans.parallel(x, b))
```

```
##    user  system elapsed
##   4.620   5.332   2.948
```

We see that the slowest by far was *feat_trans.for*, which purely used for loops. The next slowest was *feat_trans.apply* followed by *feat_trans.parallel*, leaving *feat_trans.v* as the fastest. Note that our vectorised implementation is faster than our implementation with apply, where the only difference between the two is one uses a for loop to apply the powers and one uses apply, meaning the for loop computes faster than apply. Also notice that even though *feat_trans.parallel* is parallelizing the computation it is still slower than our vectorised implementation, showing us that in this case (and in general) parallelization doesn't necessarily lead to faster compute times.

Hence, in our package we will implement the vectorised version in order to achieve faster compute times (do note, however, that its not all about speed as often readable code can be more important).