In this session, I want to give a few interesting ideas to make your R program faster.

We have mentioned this in last lab. Using for loops in R can be slow. Some of the basic operations can be accelerated by using the `apply` function family. Suppose we want to calculate the sum of exponentials on each row of a matrix. The naive way of doing this by the following:

```r
> naive.func1 <- function(X,nr,nc){
+   result <- rep(0,nr)
+   for(i in 1:nr){
+     for(j in 1:nc){
+       result[i] <- result[i]+exp(X[i,j])
+     }
+   }
+   return(result)
+ }
>
We can be a bit more efficient if we use the `sum` function.

```r
> naive.func2 <- function(X,nr){
+   result <- rep(0,nr)
+   for(i in 1:nr){
+     result[i] <- sum(exp(X[i,]))
+   }
+   return(result)
+ }
```

But a better way might be using the functions we mentioned last time.

```r
> apply.func <- function(X){
+   return(apply(exp(X),1,sum))
+ }
> rowsum.func <- function(X){
```
+     return(rowSums(exp(X)))
+ }
>
Now let’s compare the timing performance and accuracy.

> set.seed(500)
> X <- matrix(rnorm(1000000),1000,1000)
> system.time(r1 <- naive.func1(X,1000,1000))
user    system elapsed
 2.090   0.015   2.173

> system.time(r1 <- naive.func2(X,1000))
user    system elapsed
 0.031   0.004   0.034

> system.time(r1 <- apply.func(X))
user    system elapsed
 0.101   0.006   0.106

> system.time(r1 <- rowsum.func(X))
user    system elapsed
 0.016   0.001   0.017

> set.seed(500)
> X <- matrix(rnorm(1000000),20000,50)
> system.time(r1 <- naive.func1(X,20000,50))
user    system elapsed
 1.942   0.015   2.002

> system.time(r1 <- naive.func2(X,20000))
user    system elapsed
 0.080   0.005   0.083

> system.time(r1 <- apply.func(X))
user    system elapsed
 0.080   0.004   0.083

> system.time(r1 <- rowsum.func(X))
user    system elapsed
 0.019   0.001   0.020
Is there a way to improve for loops, if the operation is really complex? There is a specific way that I saw in some online tutorials.

```r
naive.func <- function(n){
  tmp <- rep(0,n)
  for(k in 1:n){
    tmp[k] <- rnorm(1)
  }
  return(tmp)
}
```

```r
lapply.func <- function(n){
  tmp <- unlist(lapply(1:n,function(x) rnorm(1)))
  return(tmp)
}
```

```r
set.seed(500)
system.time(r1 <- naive.func(1000000))

user  system elapsed
4.789 0.271  5.195

set.seed(500)
system.time(r1 <- lapply.func(1000000))

user  system elapsed
5.581 0.368  6.047

## No big difference!
```

### 0.1 Vectoring Computation

Check the slides from John Chambers and Narasimhan Balasubramanian. Here we introduce an example from their ideas.

What we want to get is the matrix below:

```
[1,] 101  0  0  0
[2,] 102 201  0  0
[3,] 103 202 301  0
[4,] 104 203 302 401
[5,] 105 204 303 402
[6,]  0 205 304 403
[7,]  0  0 305 404
[8,]  0  0  0 405
```

We start from the following matrix
The target matrix is easy to get by the simplest way of matrix calculation. But suppose now we have a matrix with a lot of rows and/or columns, then for loop would be extremely slow. The most intuitive way to do it is to move 2nd column down 1, 3rd column down 2, etc. So, let’s try to work in this way by vectorizing.

```r
> n <- nrow(x)
> p <- ncol(x)
> xx <- matrix(0, n + p, p)
> xx[1:nrow(x), ] <- x
> xx

[1,] 101 201 301 401
[2,] 102 202 302 402
[3,] 103 203 303 403
[4,] 104 204 304 404
[5,] 105 205 305 405
[6,] 0   0   0   0
[7,] 0   0   0   0
[8,] 0   0   0   0
[9,] 0   0   0   0
```

Remember we mentioned the single indexing of matrix elements last time. We can truncate the matrix according to its single index by directly working on the length. Notice we cannot use the function `dim()` directly.

```r
> length(xx) <- length(xx) - p
> dim(xx) <- c(n + p - 1, p)
> xx

[1,] 101 0   0   0
[2,] 102 201 0   0
[3,] 103 202 301 0
[4,] 104 203 302 401
[5,] 105 204 303 402
[6,] 0   205 304 403
[7,] 0   0   305 404
[8,] 0   0   0   405
```

>
Well, this is smart but it is hard to appreciate it as it seems useless in application. We now proceed show its merits, again the example is from John Chambers and Narasimhan Balasubramanian.

0.2 Convolve

Convolution is an important mathematical computing method. For the details, just Google it. We first show the most straightforward way to write the function in R, which you could just take as the definition.

```r
> convolveSlow <- function(x, y) {
+ nx <- length(x)
+ ny <- length(y)
+ xy <- rep(0, nx + ny - 1)
+ for (i in seq(1, nx)) {
+   xi <- x[i]
+   for (j in seq(1, ny)) {
+     ij <- i + j - 1
+     xy[ij] <- xy[ij] + xi * y[j]
+   }
+ }
+ return(xy)
+ }
```

Then, we write another version to do the same computation by vectorizing. To get a better insight of what we do here, note that we use the trick from the first example.

```r
> convolveV <- function(x, y) {
+ nx <- length(x)
+ ny <- length(y)
+ nxy <- nx + ny - 1
+ mat <- outer(x, y)
+ newmat <- matrix(0, nrow = nx + ny, ncol = ny)
+ newmat[1:nrow(mat), ] <- mat
+ length(newmat) <- length(mat) - ny
+ dim(newmat) <- c(nx, ny)
+ conv <- rowSums(newmat)
+ return(conv)
+ }
```

Now let’s compare the two implementations. First of all, we have to make sure that the computation is consistent between the two.
> set.seed(200)
> n <- 200
> a <- rnorm(n)
> b <- rnorm(n)
> vV <- convolveV(a, b)
> vSlow <- convolveSlow(a, b)
> sum(abs(vSlow - vV))

[1] 8.11351e-13

> ## Reasonably good!

The accuracy is consistent up to some numerical errors. Now we want to compare their efficiency. We count the running time of the two methods according to the length of vectors.

> set.seed(500)
> Vec.time <- Slow.time <- NULL
> for(n in seq(200,1000,by=50)){
+  a <- rnorm(n)
+  b <- rnorm(n)
+  t.s <- system.time(convolveSlow(a,b))
+  t.v <- system.time(convolveV(a,b))
+  Slow.time <- c(Slow.time,sum(t.s[c(1,2,4,5)]))
+  Vec.time <- c(Vec.time,sum(t.v[c(1,2,4,5)]))
+ }
> timing.df <- data.frame(time=c(Vec.time,Slow.time),method=c(rep("Vector",length(Vec.time)),rep("Slow",length(Slow.time))),size=rep(seq(200,1000,by=50),2))
> library(lattice)
> xyplot(time~size,data=timing.df,lwd=2,type="l",auto.key=TRUE,groups=method)

OK, so it can be seen that vectorizing programing gives impressive improvement on the speed, and in general, this is also true. However, there is no free lunch. What are the drawbacks of vectorizing? Just name a few obvious ones:

1. Interpretability.
2. Memory.

0.3 Call other programs from R

R may be inadequate for many tasks in practice, because it is only designed for statistical analysis, but it is likely that you have to deal with other computational perspective. So a natural way for this purpose is to use other programs. In R, we can call a few commonly used programing language/software with some well-written packages, so you don’t have to come up with your own glue.
Matlab

Matlab is commonly used for complex matrix computation and optimization. Also, many computer scientists tend to write and post their code of papers in matlab. Calling matlab from R is not very mature yet. But the simplest way is to use the package R.matlab. It opens matlab server from R and passes R object to matlab. Notice that passing objects between matlab and R is very important. The major drawback of generating and running a separate matlab script is the difficulty to pass objects in a flexible way. That is why R.matlab is my favorite way for this task, though it is not the only option. For other options, check [https://mandymejia.wordpress.com/2014/08/18/three-ways-to-use-matlab-from-r/](https://mandymejia.wordpress.com/2014/08/18/three-ways-to-use-matlab-from-r/).

First, we have to setup connection between R and matlab server.

```r
> library(R.matlab)
> Matlab$startServer()
[1] 0

> matlab <- Matlab()
> isOpen <- open(matlab)
> if (!isOpen){
+ throw("MATLAB server is not running: waited 30 seconds.")
+ Sys.sleep(20)
+ }
> 
> Then we can create the objects we need for the calculation, and pass it to matlab.

> x <- matrix(c(1,2,3,4),2,2)
> setVariable(matlab, x = x)
> evaluate(matlab, "x")

Use the function `evaluate` to start matlab operation. The corresponding matlab code is written as a string. After that, you can fetch the object from matlab by using `getVariable`.

```r
> evaluate(matlab, "y=x; z=y*x;")
> z <- getVariable(matlab, "z")
> z

```
Python

Python is another commonly used language in scientific computing and data processing. In particular, I found it is very useful in parsing raw data from text files, due to its flexible libraries and functions for text mining. Also, it is very popular in computer science so many researchers in that area like to post their python code. The package rPython can be used to call python from R (only available for Linux and Mac). The mechanism is similar as R.matlab, you pass your R object to python by python.assign, execute the python command by python.exec, then get your result back by python.get.

```
> library(rPython)
> python.call( "len", 1:3 )

[1] 3

> a <- 1:4
> b <- 5:8
> python.assign( "a", a)
> python.assign("b", b)
> python.exec("c=a+b")
> python.get("c")

[1] 1 2 3 4 5 6 7 8
```

Notice that python uses array addition by default. For numerical addition, you have to specify it differently. For instance, use the python library numpy.

```
> python.exec("import numpy")
> python.exec("an = numpy.array(a)")
```
As can be seen, the object passing is still very cumbersome. Of course, it is trivial to do all of these by executing one line each time. If you need a group of operations, just write a function for it. `python.call` is the R function to call python with objects as inputs.

```python
> python.exec("def np_add(a,b): import numpy; return(numpy.ndarray.tolist(numpy.array(a)+numpy.array(b)))")
> python.call("np_add",a,b)
[1]  6  8 10 12
```

Another useful one is `python.method.call`. Many objects in python has its only object oriented methods. This can be called by `python.method.call`.

```python
> python.assign( "a", "Hello world!" )
> python.method.call("a","split"," ")
```

```
[1] "Hello" "world!"
```

`python.load` can be used to execute an entire python file.
Figure 1: Timing comparison