Ecffient calculations

Vectorized computations

- The efficiency of calculations depends on how you perform them.
- Vectorized calculations, for example, avoid going trough individual vector or matrix elements and avoid for loops. Though very efficient, vectorized calculations cannot always be used.
- On the other hand, users having a Pascal or C programming background often forget to apply vectorized calculations where they could be used. We therefore give a few examples to demonstrate its use.

A weighted average

- Take advantage of the fact that most calculations and mathematical operations already act on each element of a matrix or vector. For example, log(x), sin(x) calculate the log and sin on all elements of the vector x.
- For example, to calculate a weighted average W

$$W = \frac{\sum_i x_i w_i}{\sum_i w_i}$$

- in R of the numbers in a vector x with corresponding weights in the vector w, use: ave.w <- sum(x*w)/sum(w)

Replacing numbers

- Suppose we want to replace all elements of a vector which are larger than one by the value 1. You could use the following construction (as in C or Fortran)
- ## timing the calculation using Sys.time
- tmp <- Sys.time()
- x <- rnorm(15000)
- for (i in 1:length(x)){
- if(x[i] > 1){ x[i] <- 1}
- }
- Sys.time - tmp
- Time difference of 0.2110000 secs

Replacing numbers

- However, the following construction is much more efficient; The second construction works on the complete vector x at once instead of going through each separate element.
- tmp <- Sys.time()
- x <- rnorm(15000)
- x[x>1] <- 1
- Sys.time() - tmp
- Time difference of 0.0400002 secs

The ifelse function

- Suppose we want to replace the positive elements in a vector by 1 and the negative elements by -1. When a normal `if- else' construction is used, then each element must be used individually.
- tmp <- Sys.time()
- x <- rnorm(15000)
- for (i in 1:length(x)){
- if(x[i] > 1){x[i] <- 1}
- else{x[i] <- -1}
- }
- Sys.time() - tmp

The ifelse function

- In this case the function ifelse is more efficient. The function ifelse has three arguments. The first is a test (a logical expression), the second is the value given to those elements of x which pass the test, and the third argument is the value given to those elements which fail the test.
- tmp <- Sys.time()
- x <- rnorm(15000)
- x <- ifelse(x>1,1,-1)
- Sys.time()-tmp

The cumsum function

- To calculate cumulative sums of vector elements use the function cumsum. For example:
- x <- 1:10
- y <- cumsum(x)
- y
- [1] 1 3 6 10 15 21 28 36 45 55

- The function cumsum also works on matrices in which case the cumulative sums are calculated through each column. Use cumprod for cumulative products, cummin for cumulative minimums and cummax for cumulative maximums.

The apply and outer functions
The apply function

- This function is used to perform calculations on parts of arrays. Specifically, calculations on rows and columns of matrices. (For list and data.fram, use lapply and sapply which will be introduced later.)
- To calculate the means of all columns in a matrix, use the following syntax:
- M <- matrix(rnorm(10000),ncol=100)
- apply(M,1,mean)
- The first argument of apply is the matrix, the second argument is either 1 or 2. If one chooses 1 then the mean of each row will be calculated, if one chooses 2 then the mean will be calculated for each column. The third argument is the name of a function that will be applied to the columns or rows.

The apply function

- The function apply can also be used with a function that you have written yourself.
- Extra arguments to your function must now be passed trough the apply function.
- The following construction calculates the number of entries that is larger than a threshold d for each column in a matrix.

The apply function

- tresh <- function(x,d){
-   sum(x>d)
- }

- M <- matrix(rnorm(10000),ncol=100)
- apply(M,1,tresh,0.6)

The lapply and sapply functions

- These functions are suitable for performing calculations on the components of a list. Specifically, calculations on the columns of a data frame.
- If, for instance, you want to find out which columns of the data frame cars are of type numeric then proceed as follows:
- data(cars)
- lapply(cars, is.numeric)

The lapply and sapply functions

- The function sapply can be used as well:
- sapply(cars, is.numeric)

- The function sapply can be considered as the `simplied' version of lapply.
- The function lapply returns a list and sapply a vector (if possible). In both cases the first argument is a list (or data frame) , the second argument is the name of a function.
- Extra arguments that normally are passed to the function should be given as arguments of lapply or sapply.

The lapply and sapply functions

- data(ChickWeight)
- mysummary <- function(x){
-   if(is.numeric(x))
-     return(mean(x))
-   else
-     return(NA)
- }

- sapply(ChickWeight,mysummary)

The outer function

- The function outer performs an outer-product given two arrays (vectors).
- This can be especially useful for evaluating a function on a grid without explicit looping.
- The function has at least three input-arguments: two vectors x and y and the name of a function that needs two or more arguments for input.
- For every combination of the vector elements of x and y this function is evaluated. Some examples are given by the code below.

The outer function

- # Ex1 #
- x <- 1:3
- y <- 1:3
- z <- outer(x,y,FUN="-")
- z

- ## Ex2 ##
- x <- c("A", "B", "C", "D")
- y <- 1:9
- z <- outer(x, y, paste, sep = "")
- z

The outer function

- ### Ex3 ###
- x <- seq(-4,4,l=50)
- y <- x
- myf <- function(x,y){
-   sin(x)+cos(y)
- }
- z <- outer(x,y, FUN = myf)
- persp(x,y,z, theta=45, phi=45, shade = 0.45)