Statistical Programming:
Week 2 Lab

Gordon Ross

gordon.ross@ed.ac.uk

Answers are given in blue text

Exercise 1: A useful R function is seq(), which can create a vector following an arithmetic progression. The help page (?seq) gives full details. Using this function, write R code to find the sum of all even numbers between 2 and 100.

```
x <- seq(2,100,by=2)
sum(x)
```

Exercise 2: Generate 500 random numbers from the standard Normal distribution (use the rnorm() function) with mean 0 and standard deviation 1, and store these in a vector. Now plot them as a histogram, then calculate their mean and standard deviation.

:

```
x <- rnorm(500,0,1)
hist(x)
mean(x)
sd(x)
```

Exercise 3: Generate 10 random numbers from the Exponential distribution (use the rexp() function) with parameter $\lambda = 1$. Now do this again (i.e enter the same command into R). Note that you get 10 different numbers each time. This is because you are generating random numbers, so they wont be the same.

In some cases, it is useful to tell R to generate the same numbers each time you run the command. This allows you to check your work if you (eg) use R on different days, or want to compare your answers to someone else. To do this in R, we use the set.seed() function, which takes a number which basically initialises the random number generator so it starts from the same place each time.

Try typing:

```
set.seed(1)
rexp(10,0.1)
```

You should get the numbers:

```
7.551818 11.816428  1.457067  1.397953  4.360686 28.949685 12.295621
5.396828  9.565675  1.470460
```

If you type the above again (including the set.seed(1) command) then you will always get these numbers.

Exercise 4: Write a for loop that iterates over the numbers 1 to 7 and prints the cube of each number using print(). Now, write the same code without using a loop.

```
for (i in 1:7) {
  print(i^3)
}

#or without a loop
x <- 1:7
print(x^3)
```

Exercise 5: In the lectures we discussed functions that take a single argument. However, functions can take multiple arguments, for example:

```
f <- function(x,y) {
  return(x+y)
}
```

Write a function which takes arguments x and n and returns:

$$1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{3} + \ldots + \frac{x^n}{n}$$

We can do it this way without a loop, using vectorised code. A version using a loop is also fine

```
f <- function(x,n) {
 temp <- x^(1:n) #vectorised! check what this does
 temp2 <- 1:n
 return( 1 + sum(temp/temp2))
}
```

Exercise 5: Suppose $x_0 = 1$ and $x_1 = 2$, and:

$$x_i = x_{i-1} + \frac{2}{x_{i-1}}, \quad i = 2, 4, \ldots$$

Write a function testloop(n) which takes the argument $n > 2$ and returns the first n values of this sequence as a vector, i.e. $x_0, x_1, \ldots, x_{n-2}$.

Hint: first assign x to be a vector of length n using $x \leftarrow numeric(n)$ and then use a loop

```
testloop<- function(n) {
  #it is good practice to check the arguments passed into functions are correct
  if (n <= 2) {
    return("error: n needs to be greater than 2")
   }

  x <- numeric(n) #assign a vector of length n
  x[1] <- 1;
  x[2] <- 2

  for (i in 3:n) {
    x[i] <- x[i-1] + 2/x[i-1]
  }
  return(x)
 }
```

Exercise 6: Re-write the following to eliminate the loops, using `apply`.

```
X <- matrix(runif(100000),1000,100)
z <- rep(0,1000)
for (i in 1:1000)
 {
      for (j in 1:100)
       {
         z[i] <- z[i] + X[i,j]
       }
 }
```

This function uses the 'if' command to decide which method to use. If we call the function with method=1 it uses a loop. If we call with method=2, it uses apply.

```
f <- function(method)
{
    X <- matrix(runif(100000),1000,100)
    if(method==1)
    {
        z <- rep(0,1000)
        for (i in 1:1000)
```

```
        {
            for (j in 1:100)
            {
                z[i] <- z[i] + X[i,j]
            }
        }
    }
    if(method==2)
    {
        z <- apply(X,1,sum)
    }
    return(z)
}
```

Exercise 7: Re-write the following, replacing the loop with efficient (i.e. vectorised) code.

```
n    <- 100000
z    <- rnorm(n)
zneg <- NULL
j    <- 1

for (i in 1:n)
 {
  if (z[i]<0)
   {
       zneg[j] <- z[i]
       j <- j + 1
   }
 }


   zneg <- z[z<0]
```

Exercise 8: Run the following code.

```
set.seed(1)
n  <- 1000
A  <- matrix(runif(n*n),n,n)
x  <- runif(n)
```

Evaluate $x^T Ax, tr(A), tr(A^T W A)$, where W is a diagonal matrix with $W_{ii} = x_i$.

Some hints: the t() function in R transposes a matrix. The diag() function can do two things: if an integer is passed as an argument, it creates a new diagonal matrix of rank n. If a matrix is passed as an argument then it returns the diagonal of the matrix (remember that 'tr' above means the trace of the matrix, which is the sum of its diagonal.

```
x  <- matrix(x,ncol=1)
t(x)%*%A%*%x
sum(diag(A))
W  <- as.numeric(x)*diag(1,nrow=length(x))
sum(diag(t(A)%*%W%*%A))
```

As a small point of efficiency, note that rather than evaluating $tr(A^TWA)$ directly we could instead rewrite this as

```
sum(A*(W%*%A))
```

which uses element-wise multiplication. This avoids the need to compute the first matrix product and is hence substantially faster to execute. If you have not seen much matrix algebra before then dont worry about this extra detail!

Exercise 9: Computers do not represent most real numbers exactly. Rather, a real number is approximated by the nearest real number that can be represented exactly (floating point number), given some scheme for representing real numbers as fixed length binary sequences. Often the approximation is not noticeable, but it can make a big difference relative to exact arithmetic (imagine that you want to know the difference between 2 distinct real numbers that are approximated by the same binary sequence, for example). One consequence of working in finite precision arithmetic is that for any number x, there is a small number ϵ such that for all e for which $|e| \leq |\epsilon|$, $x + e$ is indistinguishable from x.

- Try out the following code to find the size of this number, when $x = 1$.

```
eps <- 1
x   <- 1
while (x+eps != x)
 {
   eps <- eps/2
 }
eps/x
```

- Confirm that the final `eps` here is close to the largest e for which x and $x + e$ give rise to the same floating point number.
- `2*eps` is stored in `R` as `.Machine$double.eps`. Confirm this.
- Confirm that you get the same `eps/x` value for $x = 1/8, 1/4, 1/2, 1, 2, 4$ or 8.
- Now try some numbers which are not exactly representable as modest powers of 2, and note the difference.

Sol:

```
## Qu: How could we improve the code below?
eps1 <- 1
eps2 <- 1
eps3 <- 1
eps4 <- 1
eps5 <- 1
```

```
x1   <- 1/8
x2   <- 1/16
x3   <- 1/32
x4   <- 1/64
x5   <- 1/128
i    <- 0
j    <- 0
k    <- 0
l    <- 0
s    <- 0
while (x1+eps1 != x1)
{
    i <- i+1
    eps1 <- eps1/2
}
while (x2+eps2 != x2)
{
    j <- j+1
    eps2 <- eps2/2
}
while (x3+eps3 != x3)
{
    k <- k+1
    eps3 <- eps3/2
}
while (x4+eps4 != x4)
{
    l <- l+1
    eps4 <- eps4/2
}
while (x5+eps5 != x5)
{
    s <- s+1
    eps5 <- eps5/2
}
print(c(i,j,k,l,s))
```

Exercise 10: A random walk is a mathematical object which describes a path that consists of a succession of random steps. For example, the path traced by a molecule as it travels in a liquid or a gas, the search path of a foraging animal, superstring behavior, the price of a fluctuating stock and the financial status of a gambler can all be approximated by random walk models, even though they may not be truly random in reality. As illustrated by those examples, random walks have applications to many scientific fields including ecology, psychology, computer science, physics, chemistry, and biology, and also to economics. Random walks explain the observed behaviors of many processes in these fields, and thus serve as a fundamental model for the recorded stochastic activity.

A random walk at time t can be expressed as

$$S_t = S_{t-1} + B_t,$$

where $B_t = \pm 1$, iid, with probability $1/2$.

What is the expected value and standard deviation of S_t? Run the following code and discuss the output of the program with your classmates. Write a function in `R` that implements the code for varying p. Call the function and inspect the cases $p > 0.5$ and $p < 0.5$. Give a high-level description (4-5 steps) of the code.

```
## ------------------------------------------------
## Brownian motion as scaling limit of random walk
## ------------------------------------------------
N <- 2^(0:14)
p <- 1/2
X <- rbinom(N[length(N)],1,p)
X[X==0] <- -1                    #+-1 spins, w.p. 1/2

for(k in 1:(length(N)-1))
{
    print(paste("k is:", k))
    l <- 1
    for(i in seq(N[k],N[k+1],len=100))
    {
        if(k > 1) l<-sqrt(N[k])
        plot(1:N[k],l*cumsum(X[1:N[k]]),type="l",
             col=1,lwd=2,xlim=c(1,i),ylim=c(-i,i),
             ylab="walk",xlab="time")
    }
    Sys.sleep(1)
    lines((N[k]:N[k+1]),
          l*cumsum(X[1:N[k+1]])[N[k]:N[k+1]],col=2,lwd=2)
    Sys.sleep(1)
    for(i in seq(N[k],N[k+1],length=100))
    {
        plot((1 : N[k+1]),
             sqrt(i)*cumsum(X[ 1 : N[k+1]]),
             type="l",
             col=1,lwd=2,xlim=c(1,N[k+1]),ylim=c(-N[k+1],N[k+1]),
             ylab="walk",
             xlab="time")
    }
}
```

The point of this question is mainly to explore how a more complicated piece of code runs. However for the mean/variance of S_t note that we can write:

$$S_t = S_0 + \sum_{i=1}^{t} B_i$$

where S_0 is the initial value and the B_i variables are independent. The mean/variance can then be found using standard results about sums of independent random variables

Exercise 11: The empirical cumulative distribution function for a set of measurements $\{x_i : i = 1, \ldots, n\}$ is

$$\hat{F}(x) = \frac{\{\#x_i \leq x\}}{n}$$

where $\{\#x_i \leq x\}$ denotes the number of x_i values less than x.

- Write an R function which takes an un-ordered vector of observations `x` and returns the values of the empirical c.d.f. for each value, in the order corresponding to the original `x` vector (`?sort,?rank`).
- Modify your function to take an extra argument `plot.cdf`, which when true will cause the empirical c.d.f. to be plotted as a step function over a suitable x range.

To test your code, generate random samples using `rnorm`, `runif`, etc.

Sol:

```
edf <- function(x)
{
    z <- (rank(x)-1)/length(x)
    return(z)
}

edf <- function(x,plot.cdf=FALSE)
{
    z <- (rank(x)-1)/length(x)
    if(plot.cdf==TRUE)
    {
        plot(sort(x),sort(rank(x)/length(x)))
    }
    return(z)
}
```