Statistical Programming — Worksheet 2

By the end of the practical you should feel confident writing and calling functions, and using `if()`, `for()` and `while()` constructions.

You should complete and understand questions 1–6 for next time.

1. Review

(a) Let $t = 2$: create a vector with $(i+1)$th entry $\frac{e^{-t}t^i}{i!}$ for $i = 0, \ldots, 10$ (you might want to use the function `factorial()` for this).

```
t <- 2
n <- 10
sq <- 0:n
exp(-t) * t^sq/factorial(sq)
```

(b) Write a function with arguments `t` and `n` that evaulates $\sum_{i=0}^{n} \frac{e^{-t}t^i}{i!}$.

```
parsum = function(t, n) {
    sq <- seq(from = 0, to = n)
    exp(-t) * sum(t^sq/factorial(sq))
}
```

Note this is the same as $P(N \leq n)$ where $N \sim Poisson(t)$, which is coded as `ppois()` in R:

```
ppois(15, lambda = 10)
parsum(t = 10, 15)
```

But notice that our version doesn't work well for large `n`: try `parsum(t=200, 200)`.

(c) Write your function again using a `for()` loop. Do not use vectors, or the `sum()` function. Check it gives the same answers as (b).

```
parsum2 = function(t, n) {
    out <- 0
    for (i in 0:n) {
        out <- out + exp(-t) * t^i/factorial(i)
    }
    out
}
```

2. Solving a Quadratic. Write a function with three arguments `a`, `b` and `c`, that returns the **real** roots of the equation $ax^2 + bx + c = 0$, if any. Your function should behave well if $a = 0$ and return an empty vector when there are no real roots.

```
roots <- function(a, b, c) {
    dis <- b^2 - 4 * a * c
    ## if discriminant is negative, no solutions
```

```
    if (dis < 0)
        return(numeric(0))
    if (a == 0) {
        ## if a=0 reduce to linear case
        if (b != 0)
            return(-c/b) else return(NaN)  # if a=b=0 this is degenerate
    }
    ## quadratic formula
    (-b + c(1, -1) * sqrt(dis))/(2 * a)
}
```

3. **Sieve of Eratosthenes.** The Sieve of Eratosthenes is a method for finding all the prime numbers less than some specified n. Here is an outline of the algorithm:

   - Create a vector `x` of integers from 2 to n, and an empty vector `p`.
   - Given `x`, append the first element (say `z`) to `p`; then remove any multiples of `z` (including `z` itself) from `x`.
   - Stop when `x` is empty, and return `p`.

   Write a function to implement this of Eratosthenes. It should take one argument `n`, and return all the primes up to `n`.

```
sieve <- function(n) {
    x <- 2:n
    p <- numeric(0)
    while (length(x) > 0) {
        p = c(p, x[1])
        x = x[x%%x[1] != 0]
    }
    p
}
sieve(1000)
```

   Try it for $n = 10^3, 10^4, 10^5$. [Optional: can you see any way to speed this procedure up?]

   Once `x[1]` is greater than $\sqrt{n}$ it's clear that we won't find more prime factors, so we could just stop and return `c(p,x)` at this point.

4. **Random Walks.** Write a function `rndwlk`, with an argument `k`, that simulates a symmetric random walk (see lecture), stopping when the walk reaches k (or $-k$). After stopping it should return the entire walk.

```
rndwlk = function(k) {
    curr = 0  # current position
    out = 0  # vector of all positions
    while (abs(curr) < k) {
        curr = curr + sample(c(1, -1), 1)  # new position
        out = c(out, curr)  # add to vector
```

```
    }
    out
}
```

Try calling `plot(rndwlk(10))` a few times to see how it looks.

5. **Simulating Discrete Distributions.** In lectures you've seen that we can sample X from a discrete distribution on $\{1, \ldots, k\}$ as follows: let $p_i = P(X = i)$. Then:

   - generate $U \sim \text{Unif}[0, 1]$;
   - set $X = \min\{i \mid \sum_{j=1}^{i} p_j \geq U\}$.

   Write a function that, given `p` containing $(p_1, \ldots, p_k)$ can simulate X from this distribution. You may find the functions `cumsum()` and `which()` useful.

```
simDisc <- function(p) {
    U <- runif(1)
    cp <- cumsum(p)  # get cumulative distribution
    which(cp > U)[1]  # return which entry is first > U
}
```

Modify your function so that it takes an argument `n`, and produces a vector of `n` i.i.d. values from the distribution p. Comment on how you could check that your function worked as expected.

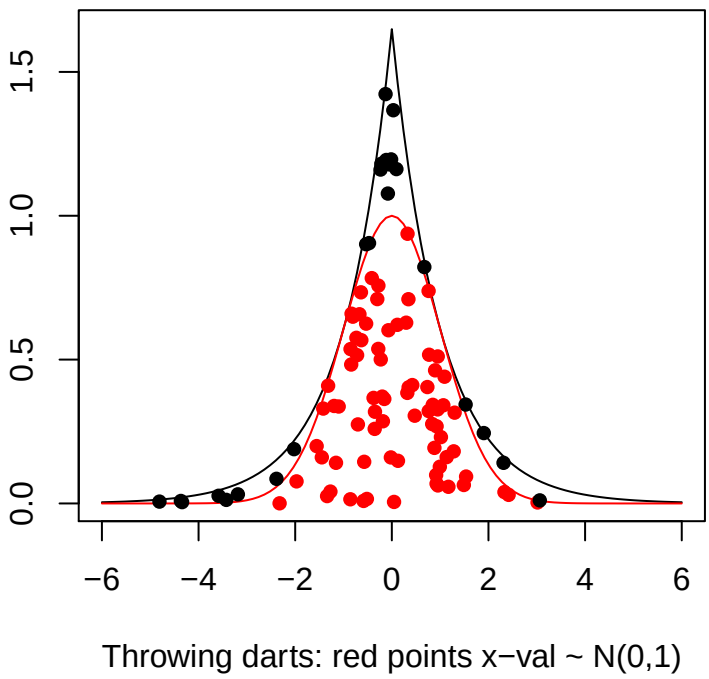
```
simDisc2 <- function(n, p) {
    U <- runif(n)
    out <- numeric(n)  # numeric vector of length n
    cp <- cumsum(p)
    for (i in 1:n) {
        out[i] <- which(cp > U[i])[1]
    }
    out
}
```

We can generate a large number of values and compare empirical (i.e. observed) frequencies with the probabilities. The CLT tells us how close the numbers should be.

6. **Rejection Sampling**. (*You wont have seen this in simulation, so consider this an intro - the algorithm below is well defined and you can certainly implement it*) We will write an R function to simulate $X \sim N(0,1)$ using rejection sampling with the double exponential proposal. That is, from a random variable Y with density

$$f_Y(y) = \exp(-|y|), \qquad y \in \mathbb{R}.$$

```r
#a plot for some intuition
#Throwing darts
x=seq(-6,6,length.out = 101)
plot(x,exp(-abs(x)+1/2),type='l',col=1,
     main="",
     xlab="Throwing darts: red points x-val ~ N(0,1)",
     ylab="",cex.lab=1)

lines(x,exp(-x^2/2),col=2)
for (k in 1:100) {
  y=sample(c(-1,1),1)*rexp(1);
  u=runif(1); #lines(c(y,y),c(0,exp(-abs(y)+1/2)),lwd=0.1);
  points(y,u*exp(-abs(y)+1/2),
         pch=16,cex=1,
         col=1+(u*exp(-abs(y)+1/2)<exp(-y^2/2)))
}
```



(i) Write a function to simulate n i.i.d. values of Y. *[Hint: you might want to start thinking about how to simulate an exponential random variable.] Here are a couple of possible solutions. You could also use a for-loop, though that is less efficient and harder to read.*

```
rdbexp <- function(n) {
    mag <- log(runif(n))  # gets an exponential by inversion
    sign <- sample(c(-1, 1), n, replace = TRUE)
    out <- mag * sign
    return(out)
}
```

An alternative to `sample()` *would be* `Y <- 2*round(runif(n)) - 1`.

(ii) Write a function implementing rejection for X. The algorithm is:

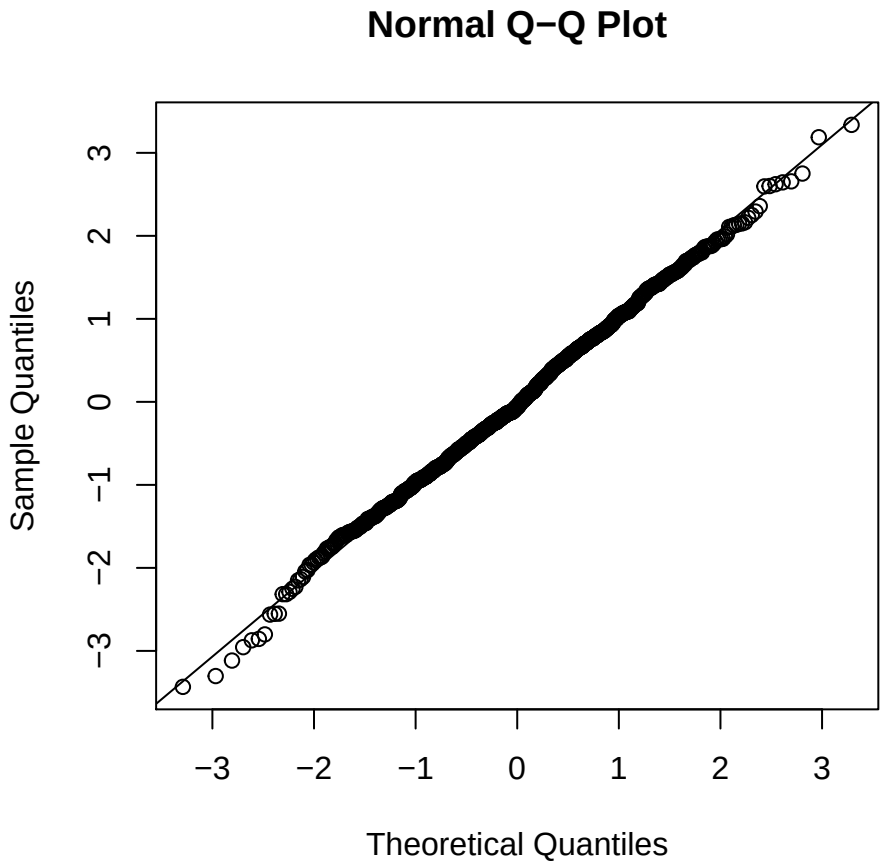
1. simulate $Y \sim \exp(-|x|)$ and $U \sim U(0,1)$;
2. if $U < \exp(-Y^2/2 + |Y| - 1/2)$ accept $X = Y$ and stop. Otherwise repeat 1.

[Hint: you can do this using a while statement. You should call the function you wrote in (a) to simulate Y. Your function should have no inputs, and return the simulated value of X.]

```
my_rnorm <- function(n) {
    out <- numeric(n)

    for (i in 1:n) {
        finished <- FALSE

        while (!finished) {
            ## try a Y, see if
            Y <- rdbexp(1)
            U <- runif(1)
            finished <- (U < exp(-Y^2/2 + abs(Y) - 0.5))
        }
        out[i] <- Y
    }

    return(out)
}
```

(iii) Test your rejection sampler by simulating 1000 samples and checking they are normal using the `qqnorm()` function.

```
X <- my_rnorm(1000)
qqnorm(X)  # should be roughtly straight
qqline(X)  # to make the comparison easier
```



7. Double for() Loop. Using two `for()` loops, write a function with an argument `n`, which constructs the $n \times n$ matrix with entries $a_{ij} = i - j$.

```
addMat <- function(n) {
    out <- matrix(0, n, n)
    for (i in 1:n) {
        for (j in 1:n) {
            out[i, j] = i - j
        }
    }
    out
}
addMat(4)
```

8. Moving Averages

(a) Write a function to calculate the moving averages of length 3 of a vector $(x_1, \ldots, x_n)^T$. That is, it should return a vector $(z_1, \ldots, z_{n-2})^T$, where

$$z_i = \frac{1}{3}\left(x_i + x_{i+1} + x_{i+2}\right), \qquad i = 1, \ldots, n-2.$$

Call this function `ma3()`.

```
ma3 = function(x) {
    n = length(x)
    x1 = x[-(1:2)]
    x2 = x[-c(1, n)]
    x3 = x[-c(n - 1, n)]
    (x1 + x2 + x3)/3
}

x = rnorm(100)
plot(ma3(x), type = "l")
```

(b) Write a function which takes two arguments, `x` and `k`, and calculates the moving average of `x` of length `k`. [Use a `for()` loop.]

```
ma = function(x, k) {
    n = length(x)
    out = x[-(1:(k - 1))]/k
    for (i in 2:k) {
        out = out + x[seq(from = k + 1 - i, to = n + 1 -
            i)]/k
    }

    out
}

max(abs(ma(x, 3) - ma3(x)))
```

(c) How does your function behave if k is larger than (or equal to) the length of x? You can tell it to return an error in this case by using the `stop()` function. Do so.

(d) How does your function behave if $k = 1$? What should it do? Fix it if necessary. *It should just return `x`, but it may cause the `for()` loop to misbehave if you used `1:(k-1)` in it.*

```
ma = function(x, k) {
    if (k == 1)
        return(x)
    n = length(x)
    out = x[-(1:(k - 1))]/k
    for (i in 2:k) {
        out = out + x[seq(from = k + 1 - i, to = n + 1 -
            i)]/k
    }

    out
}

max(abs(ma(x, 3) - ma3(x)))
```