1 Vectors and matrices

Variables are objects in R that store values; *vectors* are essentially lists of variables. Variables in R are initialized using the assignment operators `=` or `<-`, which are equivalent. For example,

```
x <- 3
print(x)
[1] 3
```

Above we have explicitly initialized `x`, but variables can be implicitly declared using the value of another variables or as the result of some function:

```
x <- 6
y <- x
print(y)
[1] 6

x <- 8
y <- x^2 - 2*x + 1
print(y)
[1] 49
```

All variables used above are *scalars*, that is, they are vectors of length 1. Longer vectors can be created by *concatenating* a list of scalars, or by explicit declaration:

```
## v is the concatenation of x1,x2,x3
x1 <- 3; x2 <- 12; x3 <- 7;
v <- c(x1, x2, x3)
print(v)
[1]  3 12  7

## create v explicitly
v <- c(3, 12, 7)
print(v)
[1] 3 12  7
```

Note that once you created a variable, that variable can essentially be used interchangeably with the number itself. A useful function for initializing variables is `seq(a,b,length=L)`, which will create a sequence of numbers beginning with `a` and ending with `b` of length `L`. Instead of specifying `length` you can specify the mesh size by using `seq(a,b,by=M)`:

```
v <- seq(1,2,length=10)
print(v)
[1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

```
v <- seq(1,2,by=.1)
print(v)
[1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

Once a vector is created it can accept the same mathematical operations as a scalar variable, and the result is a vector where that operation has been carried out on each element:

```
v <- seq(0, 5, by=1) ## is the same as v <- c(0:5)
print(v)
[1]  0  1  2  3  4  5

w <- v^2
print(w)
[1]  0  1  4  9 16 25

w <- exp(v)
print(w)
[1]   1.000000   2.718282   7.389056  20.085537  54.598150 148.413159

## If you want fewer decimal places (say, only 3) you would write
print( round(w, 3) )
[1]   1.000   2.718   7.389  20.086  54.598 148.413
```

Individual elements of `v` can be accessed by *indexing* `v`. Indices may also be themselves a vector:

```
v <- c(4, 5, 10, 19, 23, 4, 16)

## print the 6th element of v
print(v[6])

## print the 1st, 3rd, and 4th elements of v
index <- c(1, 3, 4)
print( v[index] )
[1]  4 10 19
```

One important thing here is that values in `index` cannot exceed `length(v)`. This would be like requesting the 8th element of a vector that is only of length 7. Above we have used indexing to select cases from a vector, but we can also use it to delete cases:

```
## select everything except 2nd, 5th, and 6th element from v
index <- c(2,5,6)
v[-index]
[1]  4 10 19 16
```

Other useful basic commands on vectors include

- `sort(x)` which returns the values in `x` sorted from lowest to highest
- `max(x)` and `min(x)` which return the max/min values in `x`
- `mean(x)`, `median(x)`, `sd(x)`

Exercise: Write a line of code that will generate a vector of the form $m^2, (m-1)^2, ..., 1, 0, 1, ..., (m-1)^2, m^2$ for an arbitrary input value, m. Test this code for $m = 5$ to see that

```
[1] 25 16  9  4  1  0  1  4  9 16 25
```

is returned.

Solution: This sequence is the square of the sequence $-m, -m+1, -m+2, ..., -1, 0, 1, ..., m-2, m-1, m$, so `seq(-m, m, by=1)^2` will suffice (remember to assign a value to `m` first).

Much like vectors are lists of scalars, *matrices* are lists of vectors. They can be created explicitly or by concatenating vectors. Suppose we want to create a variable containing the matrix

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

We could create this explicitly by specifying the values for each entry ($A_{11} = 1, A_{12} = 4$, *etc.*), or by binding a number of vectors:

```
## Initialize to a matrix of 0's, specifying dimensions with nrow and ncol
A <- matrix(0, nrow=3, ncol=3)
print(A)
      [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0

## specify each entry
A[1,1] = 1
A[1,2] = 4
A[1,3] = 7
A[2,1] = 2
A[2,2] = 5
A[2,3] = 8
A[3,1] = 3
A[3,2] = 6
A[3,3] = 9

print(A)
     [,1] [,2] [,3]
```

```
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

## Create A by concatenating vectors
v1 <- seq(1, 3, by=1) ## same as c(1:3)
v2 <- seq(4, 6, by=1) ## same as c(4:6)
v3 <- seq(7, 9, by=1) ## same as c(7:9)
A <- cbind( v1, v2, v3 )
colnames(A) = NULL ## don't worry about this
```

The command `cbind` creates a matrix where each of the vectors passed to it are the columns of the matrix. Notice that matrices are indexed in the same way as vectors, with the first and second index referring to the row and column numbers, respectively. If one of the two indices are left blank then that entire row/column is returned:

```
# get the second column of A
A[,2]
[1] 4 5 6

# get the third row of A
A[3,]
[1] 3  6  9
```

Similarly to vectors, usual mathematical operations on a matrix are carried out elemen-twise. One final basic matrix command that is useful is the *transpose* function, which returns a matrix with the rows and columns reversed from the input matrix:

```
t(A)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Exercise: Use the `seq`, `cbind` and `t()` commands along with usual mathematical operators to create a 5-by-5 matrix of the form $A_{ij} = j^i$.

Solution: One way of doing this is:

```
v1 <- c(1:5)
v2 <- c(1:5)^2
v3 <- c(1:5)^3
v4 <- c(1:5)^4
v5 <- c(1:5)^5

t( cbind(v1,v2,v3,v4,v5) )
```

However a far more clever alternative is

```
v <- c(1:5)
t( matrix( rep(v, 5), 5, 5) ) ^ v
```

2 For loops

Most of the programming tasks in this course will require the use of a for loop. A for loop essentially iterates a process a number of times, indexed by a counter variable. The simplest example of a for loop is one which prints the numbers in a vector successively:

```
v <- seq(0, 1, length=11)
for(i in 1:11)
{
   print(v[i])
}
[1] 0
[1] 0.1
[1] 0.2
[1] 0.3
[1] 0.4
[1] 0.5
[1] 0.6
[1] 0.7
[1] 0.8
[1] 0.9
[1] 1
```

The counter in this example was the numbers `1:11`, and the process to be completed at each iteration was to print the i'th element of `v`. The counter need not be a contiguous sequence of numbers as above, for example:

```
index <- c(1,2, 5, 8, 13, 22)
for(i in index)
{
   print(i)
}
[1] 1
[1] 2
[1] 5
[1] 8
[1] 13
[1] 22
```

So the counter iterates over the values in the `index` vector sequentially. To loop over a process that is indexed by more than one number we need to use one loop within another, commonly referred to as a *nested* loop. We can do the exercise at the end of the last section by using a nested loop:

```
A = matrix(0, 5, 5)
for(i in 1:5)
{
   for(j in 1:5)
   {
      A[i,j] = j^i
   }
}
print(A)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    1    4    9   16   25
[3,]    1    8   27   64  125
[4,]    1   16   81  256  625
[5,]    1   32  243 1024 3125
```

Exercise: Use a for loop and the indexing commands we learned to create a vector of length 30 with entries $v_1 = 0$, $v_2 = 1$ and for $i \geq 3$, $v_i = \frac{1}{2}(v_{i-1} + v_{i-2})$.

Solution:

```
## initialize v
v <- rep(0, 30)

## set the first two entries
v[1:2] <- c(0,1)

## fill in the rest
for(i in 3:30)
{
   v[i] <- .5*(v[i-1] + v[i-2])
}
```

3 Conditional execution

Many tasks in programming require there to be different instructions carried out for different inputs. For example, suppose we want a program that tells us whether a given number is negative or positive. In R we can accomplish this using an if-then statement:

```
## declare x to be some number first
if(x < 0)
{
   print("x is negative")
}  else
{
   print("x is positive")
}
```

One key point of syntax here is that the `else` has to be written on the same line as the brace which ends the `if` statement. We haven't accounted for the possibility that $x = 0$ here, but we can with a slight motification:

```
## declare x to be some number first
if(x < 0)
{
   print("x is negative")
}  else if(x > 0)
{
   print("x is positive")
}  else if(x == 0)
{
   print("x is 0")
}
```

Another key point of syntax is the difference between `==` and `=`. For example, `x=1` assigns the value 1 to x; `x==1` returns `TRUE` is x is indeed 1, and `FALSE` otherwise.

A while loop is a form of repeated conditional execution. Essentially a while loop executes what is inside the body of the loop until a certain condition is met. For example, suppose we wanted to print integers starting at 0 until they exceed 10:

```
## counter variable
k <- 0

while(k <= 10)
{

   print(k)

   ## increase k by 1 at each iteration
   k <- k + 1

}
```

The loop above checks the value of `k` at each iteration, and breaks out of the loop at soon as it finds that `k` has exceeded 10. To further illustrate the while loop we will work through a process of writing a simple program to calculate the largest factor of an integar k, other than k itself. If k is an even number, then the answer is of course $k/2$, so we assume the input number here is odd.

```
## the number to get the largest factor of
m = 7
```

```
## grid of candidate values for the largest factor
## factors of m (other than m) cannot be greater than m/2
s <- seq(0, floor(m/2), by=1) ## floor(x) rounds x down to the nearest integer

## counter variable used to index s within the loop
## we want to start with the largest values in s
k <- length(s)

## variable will be true if we find a factor of m
bool <- FALSE

## Loop will break when bool becomes TRUE
while( (bool == FALSE) & (k > 1) )
{

   ## see if s[k] is a factor of m
   bool <- ( (m/s[k]) == floor(m/s[k]) )

   ## decrease counter variable
   k <- k - 1

}

sprintf("The largest factor of %i other than itself is %i", m, s[k+1])
```

Essentially what we've done here is check every number less than $m/2$ to see whether it is a factor of m. The first time we find a factor of m, it must be the largest factor since we check in decreasing order, and the loop is broken. A similar program can be used to check whether a given number is prime, since if the largest factor of m other than itself is 1, the number is prime.

4 Random Number Generation

At the heart of many programming tasks in this course is simulating random variables. R has a number of functions that can be used to generate from various distributions:

```
# u contains n random variables uniformly distributed on (a,b)
n <- 10
a <- 1
b <- 5
u <- runif(n, min=a, max=b)

# u contains n normally distributed variables with mean mu and variance sigma
n <- 10
mu <- 0
sigma <- 2
```

```
u <- rnorm(n, mean=mu, sd=sqrt(sigma))

# u contains n exponentially distributed variables with parameter lambda
n <- 10
lambda <- 5
u <- rexp(n, rate=lambda)

# u contains n binomial random variables with parameters k, p
n <- 10
k <- 20
p <- .5
u <- rbinom(n, size=k, prob=p)
```

For each of the random number generating functions above (which begin with `r`), there is a corresponding probability density function, CDF function, and quantile function (inverse CDF), which begin with `d`, `p`, and `q`, respectively. For example the normal density, CDF, and quantile functions are `dnorm`, `pnorm`, and `qnorm`.

We can use simulation to estimate quantities that may or may not be easy to calculate by hand. For example, recall that the Binomial(k, p) distribution as probability mass function

$$p(x) = \binom{k}{x} p^x (1-p)^{k-x}$$

Using this we can explicitly calculate, for example, $P(X > 3)$ when k and p are known. Suppose that $k = 10$ and $p = .35$ can also use R to calculate this by simulation:

```
k <- 10
p <- .35

#### exact probability
# using the built-in binomial CDF function
1-pbinom(3, size=k, prob=p)
[1] 0.486173

# calculating it ourselves
prob <- 0
for(x in 4:10) prob <- prob + choose(k, x)*(p^x)*((1-p)^(k-x))
[1] 0.486173

#### simulated probability
# nreps: number of simulated variables-- nreps larger --> more accurate
nreps = 10000
X = rbinom(nreps, size=k,  prob=p)
mean(X > 3)
[1] 0.4864
```

One can also use simulation to estimate the mean and variance of a random variable. Suppose a random variable X has an exponential distribution with parameter λ and are interested in its mean and variance. Recall that the exponential distribution has probability density function

$$p(x) = \lambda e^{-\lambda x}$$

from the definition of the expectation of a random variable,

$$\begin{aligned} E(X) &= \int_0^\infty x p(x) dx = \int_0^\infty x \lambda e^{-\lambda x} dx \\ &= \left(-x e^{-\lambda x} + \int e^{-\lambda x} dx \right) \Big|_0^\infty \\ &= \left(-x e^{-\lambda x} - \frac{e^{-\lambda x}}{\lambda} \right) \Big|_0^\infty \\ &= 0 - \frac{-1}{\lambda} = \frac{1}{\lambda} \end{aligned}$$

So $E(X) = 1/\lambda$. A similar calculation shows that $E(X^2) = 2/\lambda^2$, so

$$\text{var}(X) = E(X^2) - E(X)^2 = \frac{1}{\lambda^2}$$

Now we can use R to verify this calculation.

```
lambda <- 5

## exact values
EX <- 1/lambda
VarX <- 1/(lambda^2)
print(c(EX, VarX))
[1] 0.20 0.04

## sample mean and variance
X <- rexp(10000, lambda)
c(mean(X), var(X))
[1] 0.20292313 0.04186984
```

In this example the calculation was just a matter of using integration by parts. In other examples the calculation can be literally impossible, and simulation is the only way of getting a handle on it. For example, suppose $X \sim N(\theta, 1)$ and we were interested in $E(Y)$ and var(Y) where

$$Y = \frac{e^X}{1 + e^X}$$

This is the inverse-logistic function, which maps real numbers to the interval $(0, 1)$ and is used in logistic regression to map the fitted values back to probabilities (outside the scope of this course). The the expectation of Y is,

$$E(Y) = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-(x-\theta)^2/2} \frac{e^x}{1+e^x} dx$$

which is not expressible in closed form. Similarly the variance is not expressible. However, we can get a handle on this by simulation:

```
## mean of X
mu <- 0

## generate normal random variables
X <- rnorm(10000, mean=mu, sd=1)

## calculate Y
Y <- exp(X)/(1 + exp(X))

## sample mean and variance
c( mean(Y), var(Y) )
[1] 0.49825832 0.04389209
```

To get an idea of how the mean and variance of Y depends on θ we can estimate these quantities by simulation across a grid of values for θ and plot the results:

```
## Let mu go from -5 to 5
theta <- seq(-5, 5, length=100)

## Storage for the estimated means and variances
EY <- rep(0, 100) ## same as seq(0, 0, length=100)
VarY <- rep(0, 100)
for(i in 1:100)
{

      ## generate the underlying normal variables
      X <- rnorm(10000, mean=theta[i], sd=1)

      ## Calculate Y
      Y <- exp(X)/(1 + exp(X))

      ## store the sample mean and variance
      EY[i] <- mean(Y)
      VarY[i] <- var(Y)

}

## plot EY against theta
plot( theta, EY,  xlab=expression(" " *theta), ylab="EY",
    main=expression("EY vs. " *theta), col=2 )
```

```
lines(theta,  EY, col=2)

## plot VarY against theta
plot( theta, VarY,  xlab=expression(" " *theta), ylab="VarY",
   main=expression("VarY vs. " *theta), col=3 )
lines(theta,  VarY, col=3)
```

In the plots below we can see that $E(Y)$ increases as a function of θ and is $1/2$ when $\theta = 0$. It is also apparent that $\mathrm{var}(Y)$ is maximized at $\theta = 0$.

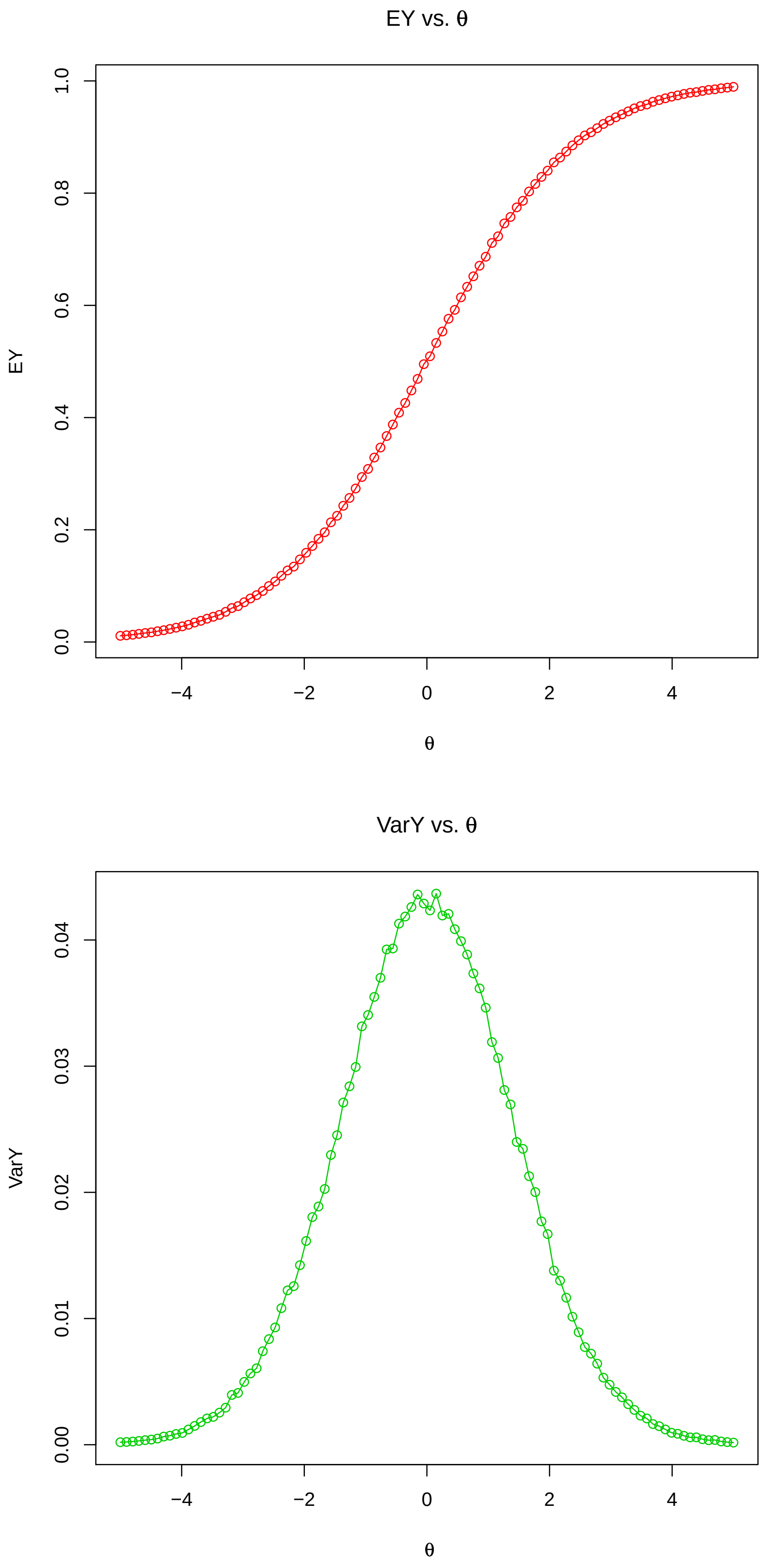

Figure 1: Mean and variance of Y as a function of θ