Intro to R Programming: Lab 1 - Solutions

More on computers' mistakes

1.Let's implement the two formulae for the variance.

```
sum((x-mean(x))^2)/n                    # Formula on left hand side.
```

```
## [1] 0.009400396
```

```
sum(x^2)/n - mean(x)^2                  # Formula on right hand side.
```

```
## [1] 0.03125
```

The two results are clearly different. Let's compare these with the R built-in function `var`.

```
var(x)                                  # Use the built-in function.
```

```
## [1] 0.009409806
```

The right hand formula is numerically unstable, so you should not use it.

2. Solutions already in the lab script.

R as a calculator

3.

```
5-7/8
```

```
## [1] 4.125
```

```
(5-7)/8
```

```
## [1] -0.25
```

```
pi -333/123
```

```
## [1] 0.4342756
```

```
256^(1/4)
```

```
## [1] 4
```

```
exp(1)
```

```
## [1] 2.718282
```

```
(1/1000)^1000
```

```
## [1] 0
```

```
(1/56)^(1/4)
```

```
## [1] 0.3655552
```

4.

```
x <- 1                # definition of the variable x to be 1.
x <- x/2 + 1/x        # update of x
x
```

```
## [1] 1.5
```

Let's repeat the update a few times and let's compare it to $\sqrt{2}$.

```
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x
```

```
## [1] 1.414214
```

```
sqrt(2)
```

```
## [1] 1.414214
```

Logical Variables

5. We can use

```
c <- (a & b) | (!a & !b)
```

or

```
c <- a==b
```

Intro to R Programming: Lab 2 - Solutions

Task 1

```
x <- seq(1, 5, by=0.3)
mean(x)
```

```
## [1] 2.95
```

```
sd(x)
```

```
## [1] 1.25499
```

```
x.standardised <- (x-mean(x))/sd(x)
mean(x.standardised)
```

```
## [1] -1.676022e-16
```

```
sd(x.standardised)
```

```
## [1] 1
```

Task 2

```
x <- rnorm(100, mean=1, sd=1)
n <- length(x)
x.bar <- 1/n * sum(x)
differences <- x - x.bar
sx2 <- 1/(n-1) * sum(differences^2)
t <- sqrt(n) * x.bar / sqrt(sx2)
t
```

```
## [1] 10.44616
```

Using the built-in functions `mean` and `sd` we could have also used

```
n <- length(x)
t <- sqrt(n) * mean(x)/ sd(x)
t
```

```
## [1] 10.44616
```

Alternatively, we could have used the high-level function `t.test`

```
t.test(x)
```

```
##
##  One Sample t-test
##
## data:  x
## t = 10.446, df = 99, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.8422299 1.2372141
## sample estimates:
## mean of x
##  1.039722
```

Task 3

```
sum(2^(0:9))
```

```
## [1] 1023
```

```
sum(1/2^(1:1e5))
```

```
## [1] 1
```

Task 4

1. Running the code

```
x <- rnorm(100)
mean(x)
```

```
## [1] -0.08833979
```

```
median(x)
```

```
## [1] -0.1705871
```

shows that the sample mean and median (of data from the normal distribution) does not vary by all that much.

2. Looking at the Cauchy distribution

```
x <- rcauchy(100)
mean(x)
```

```
## [1] -0.1557537
```

```
median(x)
```

```
## [1] -0.04299161
```

gives a different picture. Whilst the median is relatively stable, the mean appears to be rather unstable. The reason for this is that the Cauchy distribution is heavy-tailed, i.e. samples drawn from it will have extreme outliers, which make the mean close to being useless.

3. We can use

```
x <- rnorm(100)
x.trimmed <- sort(x)[11:90]
mean(x.trimmed)
```

```
## [1] -0.02218451
```

or use the argument `trim` of the built-in function `mean`

```
x <- rcauchy(100)
mean(x,trim=0.1)
```

```
## [1] -0.02267795
```

Task 5

```
u <- 1:100
u[u<55] <- 0
u
```

```
##   [1]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
##  [18]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
##  [35]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
##  [52]  0  0  0 55 56 57 58 59 60 61 62 63 64 65 66 67 68
##  [69] 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
```

```
##  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

Task 6

```
a <- c(TRUE,FALSE)
b <- c(FALSE,FALSE)
c <- (a & !b)
d <- !(a | b)
c
```

```
## [1]  TRUE FALSE
```

```
d
```

```
## [1] FALSE  TRUE
```

Task 7

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
rep(1:3,times=4)
```

```
##  [1] 1 2 3 1 2 3 1 2 3 1 2 3
```

```
rep(1:3,each=2)
```

```
## [1] 1 1 2 2 3 3
```

```
c(1:20,19:1)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 19 18 17
## [24] 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

```
(1:10)^2
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

```
2^(0:10)
```

```
##  [1]    1    2    4    8   16   32   64  128  256  512 1024
```

Task 8 It is easiest to start with a diagonal matrix and then set the few off-diagonal entries which are not 0.

```
P <- diag(c(1,5,1,7,9))
P[1,4] <- -1
P[3,1] <- 3
P
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0   -1    0
## [2,]    0    5    0    0    0
## [3,]    3    0    1    0    0
## [4,]    0    0    0    7    0
## [5,]    0    0    0    0    9
```

1.

```
P[1,]
```

```
## [1]  1  0  0 -1  0
```

2.

```
P[,2]
```

```
## [1] 0 5 0 0 0
```

3.

```
t(P)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    3    0    0
## [2,]    0    5    0    0    0
## [3,]    0    0    1    0    0
## [4,]   -1    0    0    7    0
## [5,]    0    0    0    0    9
```

```
solve(P)
```

```
##      [,1] [,2] [,3]       [,4]      [,5]
## [1,]    1  0.0    0  0.1428571 0.0000000
## [2,]    0  0.2    0  0.0000000 0.0000000
## [3,]   -3  0.0    1 -0.4285714 0.0000000
## [4,]    0  0.0    0  0.1428571 0.0000000
## [5,]    0  0.0    0  0.0000000 0.1111111
```

4.

```
P[1,] <- 1:5
P
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    0    5    0    0    0
## [3,]    3    0    1    0    0
## [4,]    0    0    0    7    0
## [5,]    0    0    0    0    9
```

5.

```
P[P!=0] <- 1
P
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    1    1    1
## [2,]    0    1    0    0    0
## [3,]    1    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

Task 9 Multiplying to matrices is much slower than computing the product of a matrix (of the same size) and a vector. The first line of code multiplies the two matrices $\mathbf{A}$ and $\mathbf{B}$ first (which takes very long), and then multiplies the result by the vector $\mathbf{x}$. The second line of code never multiplies two matrices. The result of $\mathbf{B} \cdot \mathbf{x}$ is another vector of length 1000. $\mathbf{A}$ is then multiplied by this vector.

Task 10 All parts in the code below.

```
A <- rbind(c( 1,  2,  3),
           c( 2, 20, 26),
           c(3, 26, 70))       # Define matrix A
b <- c(4, 52, 31)              # Define vector b
```

```r
solve(A, b)                    # Solve Ax=b
```

```
## [1] -1  4 -1
```

```r
solve(A)%*%b                   # the same thing but slower (for large matrices)
```

```
##      [,1]
## [1,]   -1
## [2,]    4
## [3,]   -1
```

```r
L <- t(chol(A))                # Compute Choleski factor

L%*%t(L)                       # Should be the same as A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2   20   26
## [3,]    3   26   70
```

```r
v <- solve(L)%*%b              # Both lines should give the same (fwdsolve)
v
```

```
##      [,1]
## [1,]    4
## [2,]   11
## [3,]   -6
```

```r
v <- forwardsolve(L, b)        # (much faster for large matrices)
v
```

```
## [1]  4 11 -6
```

```r
z <- solve(t(L))%*%v           # Both lines should give the same (backsolve)
z
```

```
##      [,1]
## [1,]   -1
## [2,]    4
## [3,]   -1
```

```r
z <- backsolve(t(L), v)        # (much faster for large matrices)
z
```

```
## [1] -1  4 -1
```

```r
det(A)                         # Both line compute det(A)
```

```
## [1] 576
```

```r
prod(diag(L))^2
```

```
## [1] 576
```

Intro to R Programming: Lab 3 - Solutions

Notice that for all solutions, you will need to include the correct directory where your files are located. Here I assume that you told Rstudio the current working directory is the one where the files are located.

Task 1

The file `health.txt` is white-space separated and the first line contains the column names. The file does not contain any missing values.

```
health <- read.table("health.txt", header=TRUE)
str(health)
```

```
## 'data.frame':    169 obs. of  6 variables:
##  $ Country          : Factor w/ 169 levels "Albania","Algeria",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ Region           : Factor w/ 7 levels "East Asia & Pacific",..: 2 4 7 3 2 1 2 2 3 4 ...
##  $ Year             : int  1995 1995 1995 1995 1995 1995 1995 1995 1995 1995 ...
##  $ Population       : num  3141102 28291591 12105105 34855160 3223173 ...
##  $ LifeExpectancy   : num  71.9 68.5 42.1 72.6 68.6 ...
##  $ HealthExpenditure: num  0.282 0.621 0.207 6.154 0.257 ...
```

The file `cia.csv` is comma-separated and the first line contains the column names. The file contains missing values codes as "?".

```
cia <- read.csv("cia.csv", na.strings="?")
str(cia)
```

```
## 'data.frame':    255 obs. of  7 variables:
##  $ X                    : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Country              : Factor w/ 255 levels "Afghanistan",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ Continent            : Factor w/ 7 levels "Africa","Asia",..: 2 5 4 1 7 4 1 3 7 3 ...
##  $ Population           : int  33609937 15700 3639453 34178188 65628 83888 12799293 14436 NA 85632 ...
##  $ Life                 : num  44.6 NA 78 74 73.7 ...
##  $ GDP                  : num  1.28e+10 NA 1.35e+10 1.71e+11 4.62e+08 ...
##  $ MilitaryExpenditure: num  2.44e+08 NA 2.01e+08 5.65e+09 NA ...
```

Task 2

We can write the data frame `health` into a csv file as follows

```
write.table(health, file="health.csv", sep=",", row.names=FALSE, col.names=TRUE, na="*")
```

Task 3

```
health <- transform(health, ExpectancyGroup=cut(LifeExpectancy,
breaks=c(0, 40, 70, Inf), labels=c("low", "medium", "high")))
str(health$ExpectancyGroup)
```

```
##  Factor w/ 3 levels "low","medium",..: 3 2 2 3 2 3 3 2 3 3 ...
```

Task 4

1. You can read in the data using

```
maternity <- read.csv("maternity.csv", header=TRUE)
```

2.

```
maternity <- transform(maternity, smokeprop=Smoking/(Maternities-SmokingUnknown),
                                  bfprop=Breastfeeding/(Maternities-BreastfeedingUnknown))
```

3.

```
# Smoking
extremes <- c(which.min(maternity$smokeprop),which.max(maternity$smokeprop))
                                    # Determines index of smallest and largest obs.
maternity[extremes,c("HealthAuthority","smokeprop")]
```

```
##    HealthAuthority  smokeprop
## 56 Westminster PCT 0.03262643
## 32   Blackpool PCT 0.29350649
```

```
# Breastfeeding
extremes <- c(which.min(maternity$bfprop),which.max(maternity$bfprop))
maternity[extremes,c("HealthAuthority","bfprop")]
```

```
##          HealthAuthority    bfprop
## 37          Knowsley PCT 0.3716075
## 13 Haringey Teaching PCT 0.9560557
```

4.

```
# London
maternity.london <- subset(maternity, Region=="London")
mean(maternity.london$smokeprop)
```

```
## [1] 0.06274764
```

```
mean(maternity.london$bfprop)
```

```
## [1] 0.8847122
```

```
# North-West
maternity.nw <- subset(maternity, Region=="North West")
mean(maternity.nw$smokeprop)
```

```
## [1] 0.1737148
```

```
mean(maternity.nw$bfprop)
```

```
## [1] 0.6288463
```

Alternatively we can use the function `colMeans`.

```
colMeans(subset(maternity, Region=="London")[,c("smokeprop","bfprop")]) #London
```

```
##  smokeprop     bfprop
## 0.06274764 0.88471221
```

```
colMeans(subset(maternity, Region=="North West")[,c("smokeprop","bfprop")]) # North-West
```

```
## smokeprop    bfprop
## 0.1737148 0.6288463
```

The above code computes the average proportion of breastfeeding / smoking mothers averaged over all PCT's in London and the North West. Sctirctly speaking, if we want to compute the average proportion of breastfeeding / smoking mothers across London and the North West we have to compute a weighted mean to account for the fact that the PCT's have different numbers of maternities.

```
# London
weighted.mean(maternity.london$smokeprop, w=maternity.london$Maternities)
```

```
## [1] 0.06111505
```

```r
weighted.mean(maternity.london$bf, w=maternity.london$Maternities)
```

```
## [1] 0.8877516
```

```r
# North-West
weighted.mean(maternity.nw$smokeprop, w=maternity.nw$Maternities)
```

```
## [1] 0.1705526
```

```r
weighted.mean(maternity.nw$bf, w=maternity.nw$Maternities)
```

```
## [1] 0.633896
```

The same weigting can (should?) also be used in parts (5) and (6) (not shown here).

5.

```r
# Deprivation at most 10
colMeans(subset(maternity, Deprivation<=10)[,c("smokeprop","bfprop")])
```

```
##  smokeprop     bfprop
## 0.07443845 0.84044513
```

```r
# Deprivation at least 40
colMeans(subset(maternity,  Deprivation>=40)[,c("smokeprop","bfprop")])
```

```
## smokeprop    bfprop
## 0.1128379 0.7224793
```

6.

```r
# Many smokers
mean(subset(maternity, smokeprop>0.25)$bfprop)
```

```
## [1] 0.5133014
```

```r
# Few smokers
mean(subset(maternity, smokeprop<0.15)$bfprop)
```

```
## [1] 0.7965106
```

Task 5

1.

```r
x <- log(alligator$Length)          # Define x
y <- log(alligator$Weight)          # Define y
```

2.

```r
# Coefficients
sxx <- sum((x - mean(x))^2)          # Compute sums of squares
sxy <- sum((x - mean(x)) * (y - mean(y)))
beta.1 <- sxy / sxx                  # Compute estimates
beta.0 <- mean(y) - beta.1 * mean(x)
c(beta.0,beta.1)
```

```
## [1] -8.488617  3.434303
```

```r
# Fitted values
y.hat <- beta.0 + beta.1 * x         # Fitted values

# Estimated variance
```

```r
sigma2 <- sum((y.hat-y)^2) / (length(y) - 2)
                                        # Estimate of residual variance
sigma2
```

```
## [1] 0.01524718
```

```r
# R^2
R2 <- 1 - sum((y.hat-y)^2) / sum((y-mean(y))^2)
                                        # Coefficient of determination
R2
```

```
## [1] 0.9806848
```

3.

```r
X <- cbind(1,x)                         # Create design matrix

XtX <- t(X)%*%X                         # Prepare calculation of beta
Xty <- t(X)%*%y

beta <- solve(XtX,Xty)                  # Compute beta
beta
```

```
##        [,1]
##   -8.488617
## x  3.434303
```

```r
y.hat <- X%*%beta                       # fitted values
```

Intro to R Programming: Lab 4 - Solutions

Notice that for all solutions, you will need to include the correct directory where your files are located. Here I assume that you told Rstudio the current working directory is the one where the files are located.

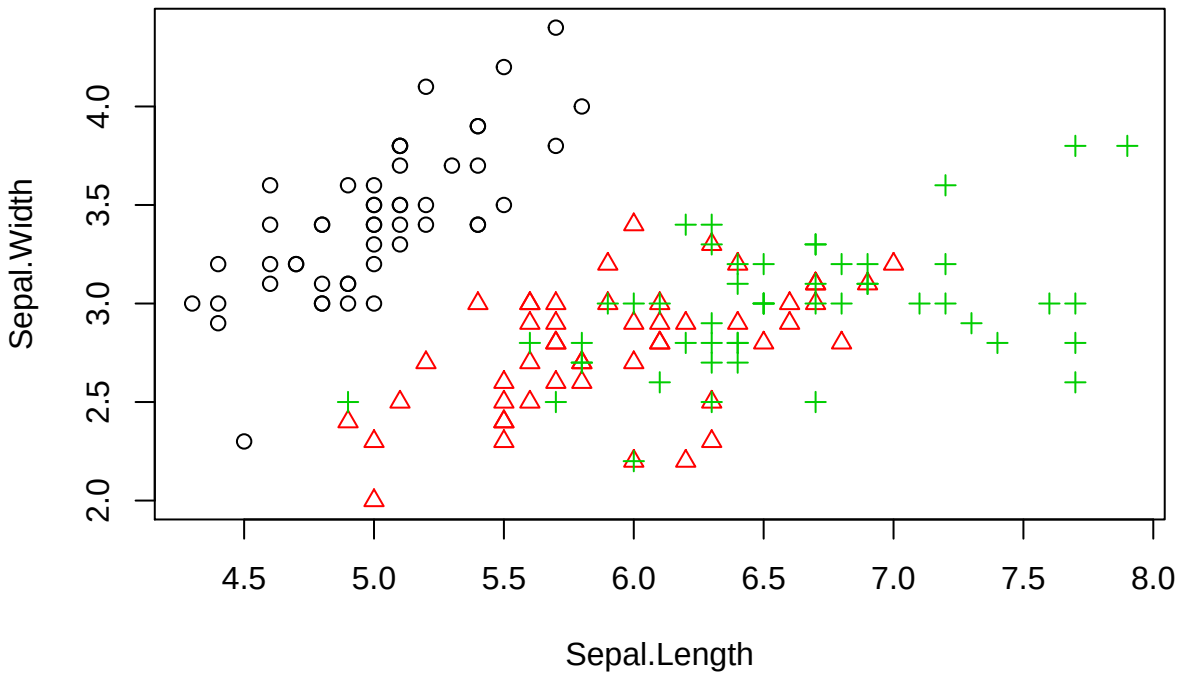
Task 1

1.

```
iris <- read.csv("iris.csv")
```

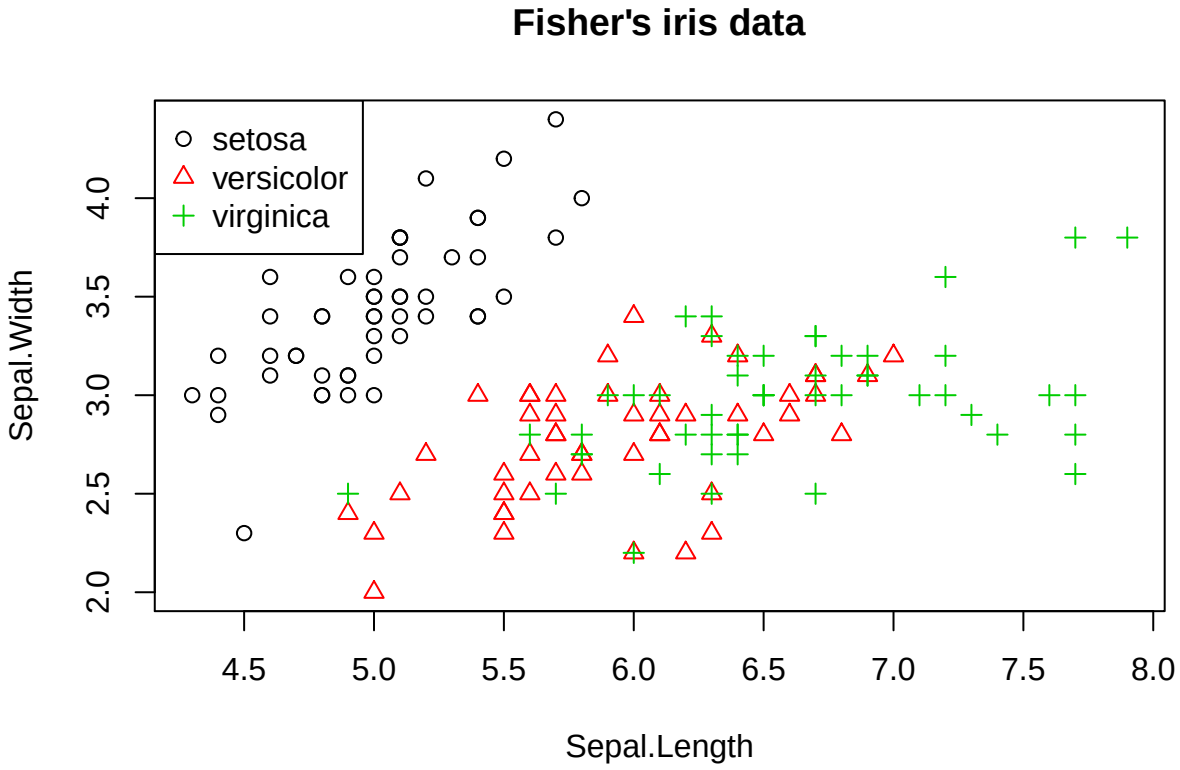
2.

```
plot(Sepal.Width~Sepal.Length, data=iris,
     col=unclass(Species),
     pch=unclass(Species))
```



3.

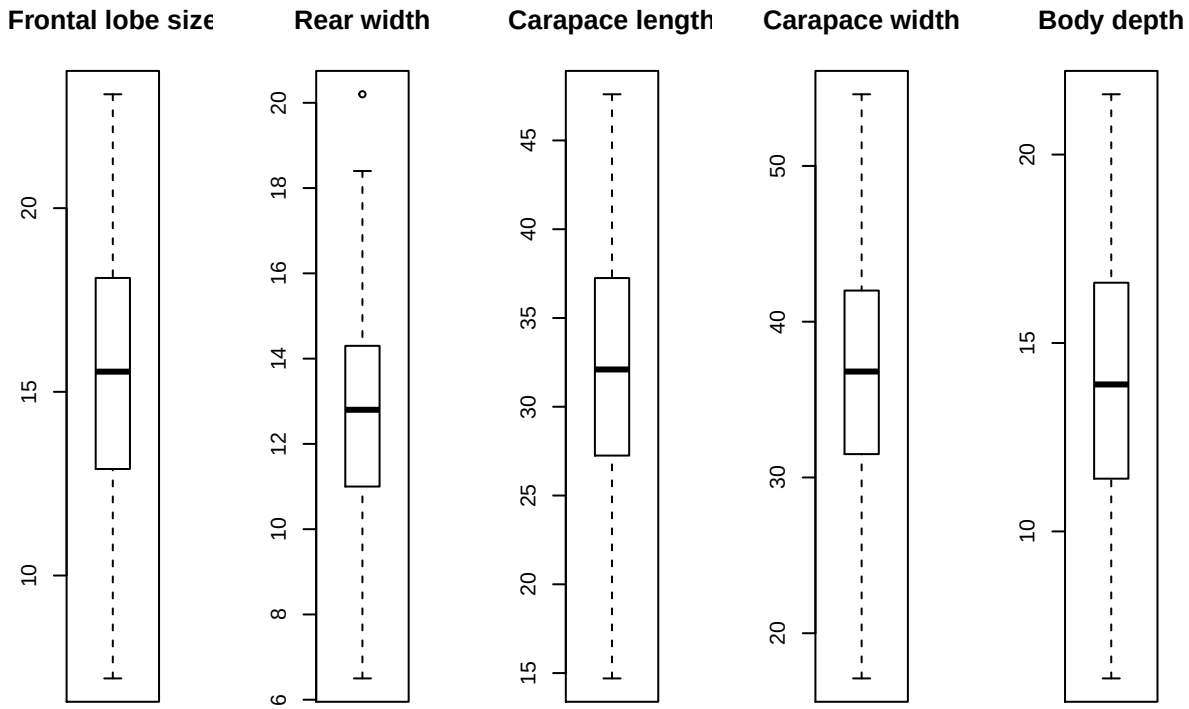
```
plot(Sepal.Width~Sepal.Length, data=iris,
     col=unclass(Species),
     pch=unclass(Species),
     main="Fisher's iris data")
legend("topleft", pch=1:3, col=1:3,
       legend=c("setosa", "versicolor", "virginica"))
```



Task 2

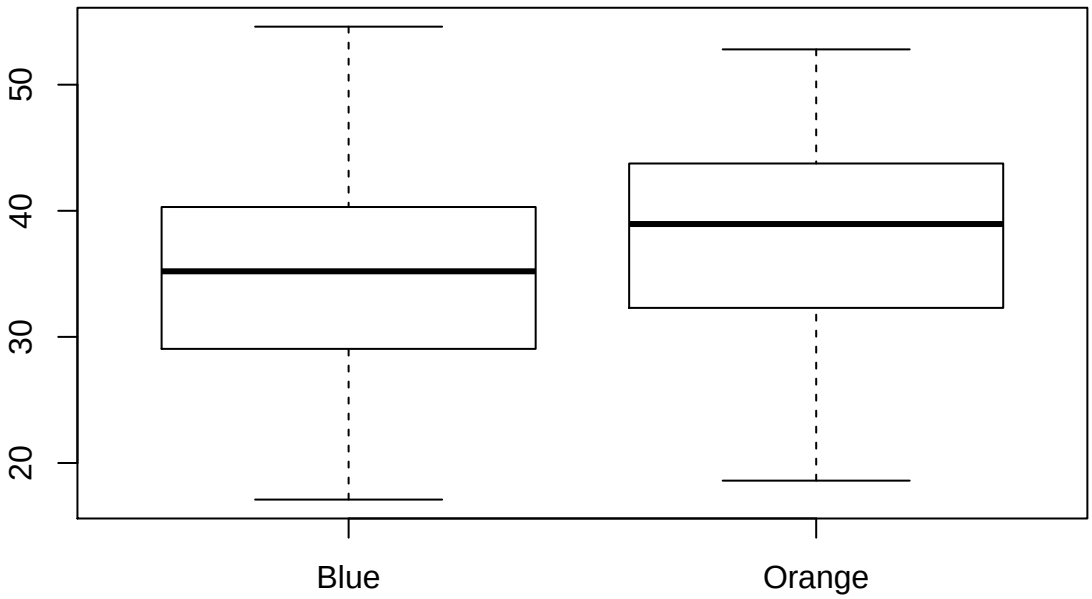
1.

```
crab <- read.csv("crab.csv")                    # Loading the dataset
par(mfrow=c(1,5))                               # Split plotting area
boxplot(crab$FL, main="Frontal lobe size")      # Create the boxplots
boxplot(crab$RW, main="Rear width")
boxplot(crab$CL, main="Carapace length")
boxplot(crab$CW, main="Carapace width")
boxplot(crab$BD, main="Body depth")
```



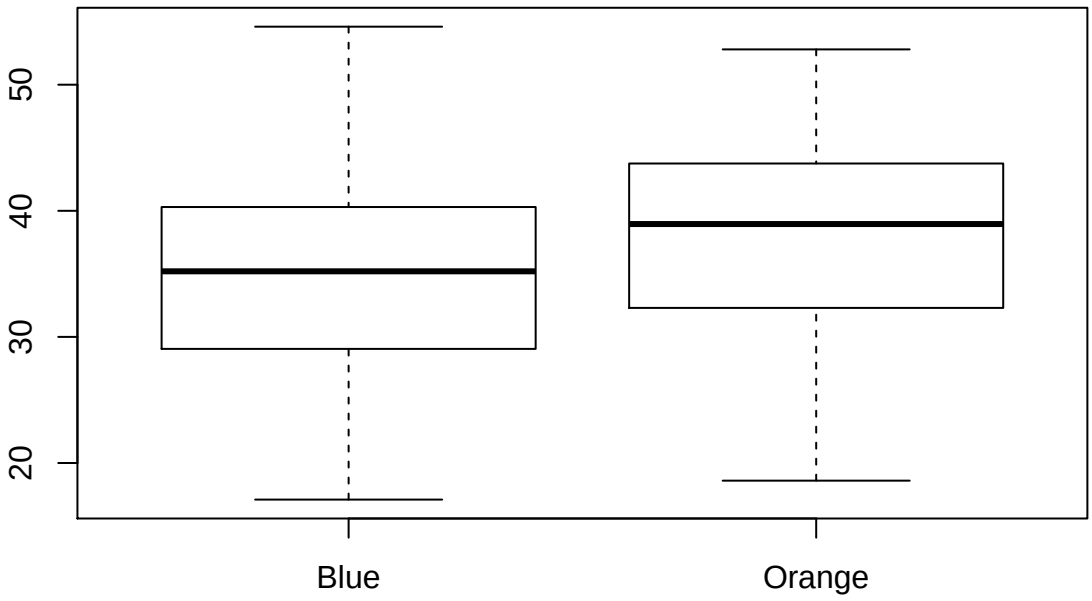
2.

```
par(mfrow=c(1,1))
CW.orange <- crab$CW[crab$sp=="O"]
CW.blue <- crab$CW[crab$sp=="B"]
boxplot(CW.blue, CW.orange, names=c("Blue", "Orange"))
```



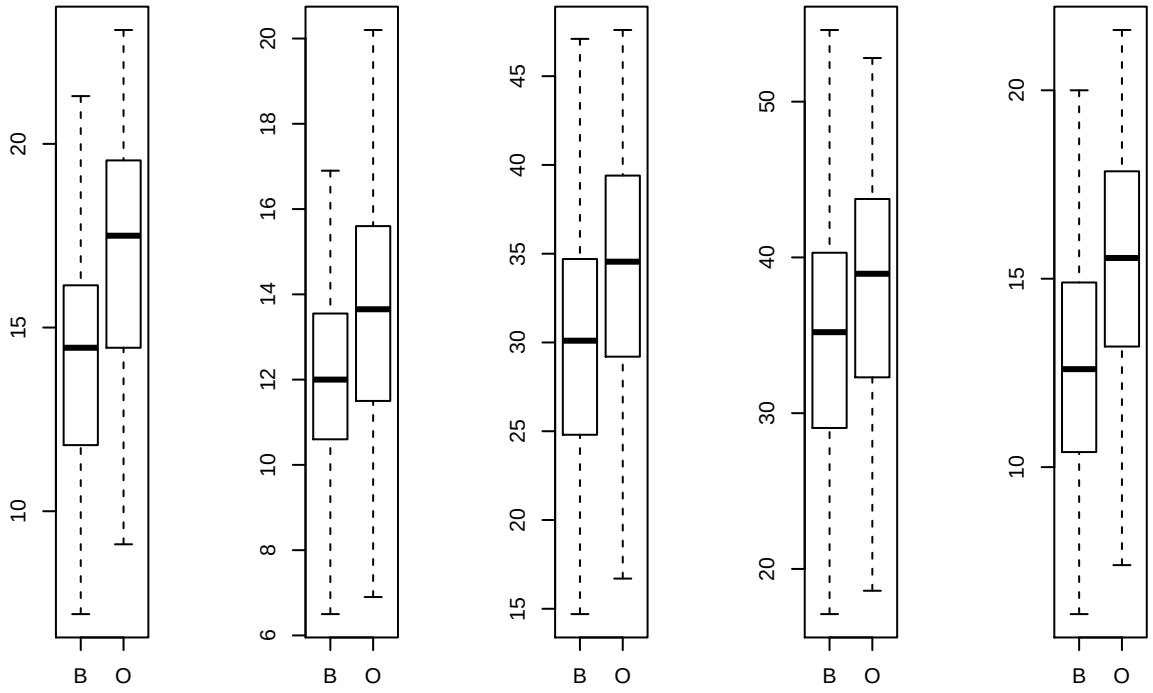
3.

```
boxplot(CW~sp,data=crab,names=c("Blue", "Orange"))
```



4.

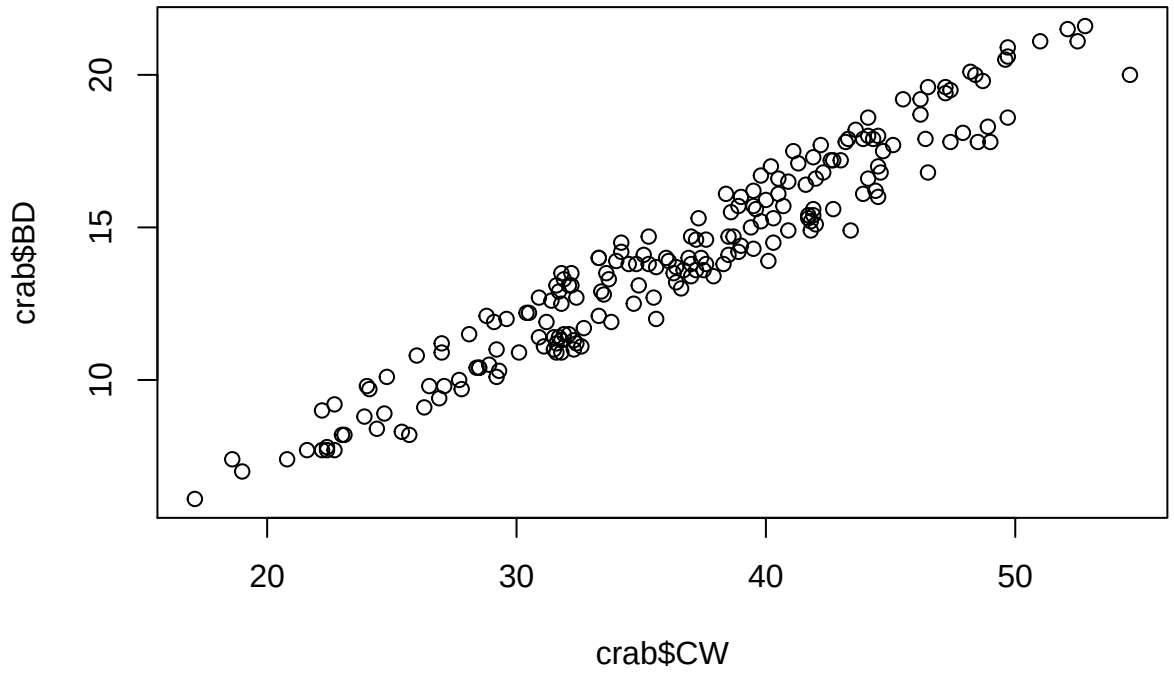
```
par(mfrow=c(1,5))
boxplot(FL~sp,data=crab)
boxplot(RW~sp,data=crab)
boxplot(CL~sp,data=crab)
boxplot(CW~sp,data=crab)
boxplot(BD~sp,data=crab)
```



Though the boxplots are different for the two species, there is a lot of overlap, so telling the two species apart will be difficult when we just look at a single variable.

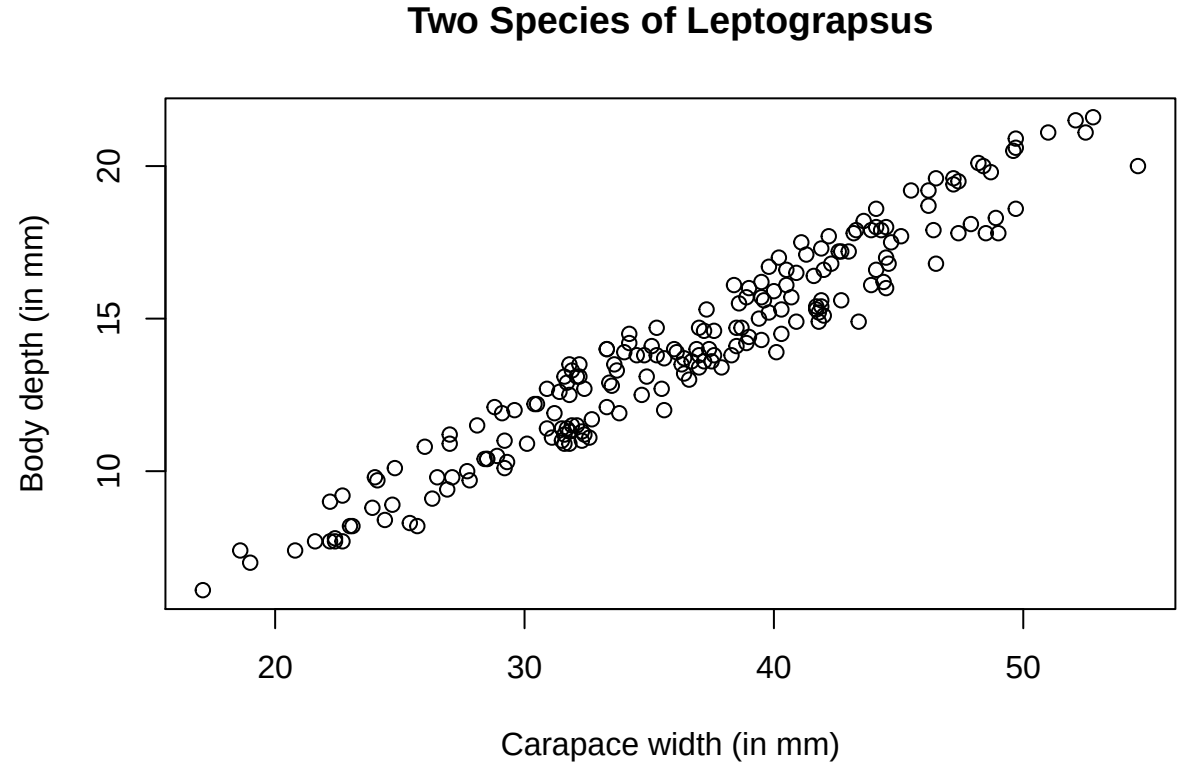
5.

```
plot(crab$CW, crab$BD)
```



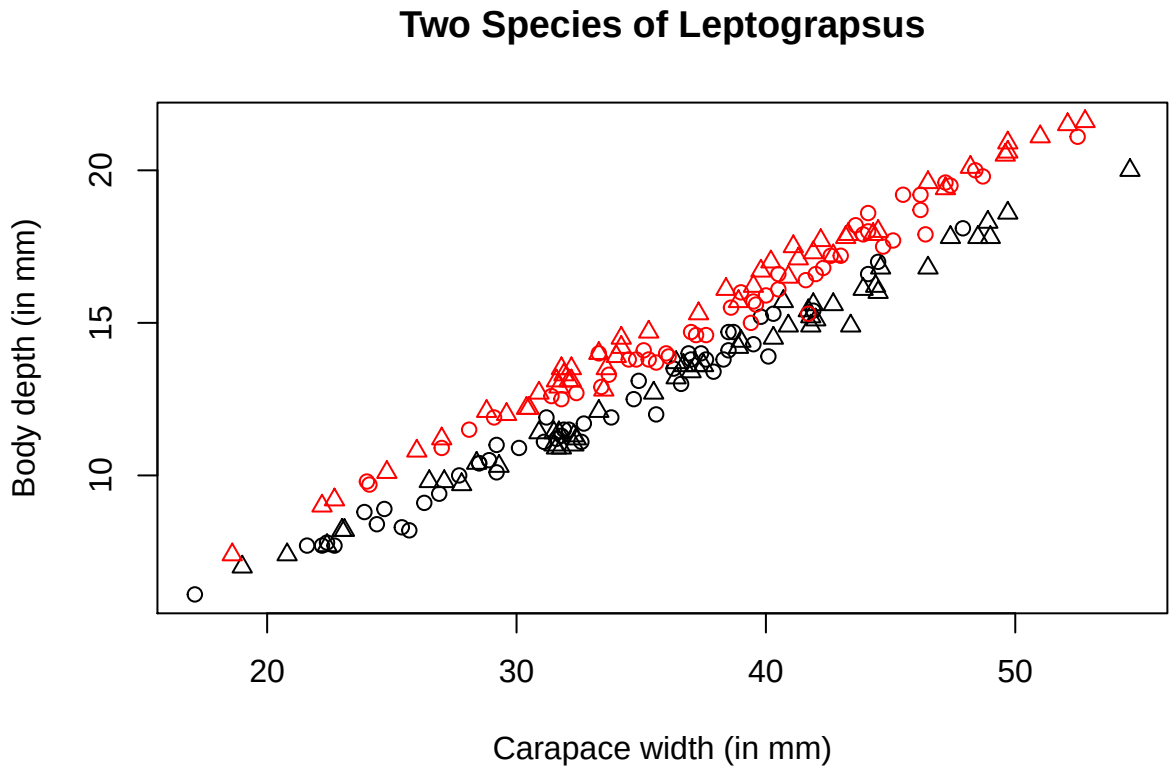
6.

```
plot(crab$CW, crab$BD, xlab="Carapace width (in mm)", ylab="Body depth (in mm)")
title("Two Species of Leptograpsus")
```



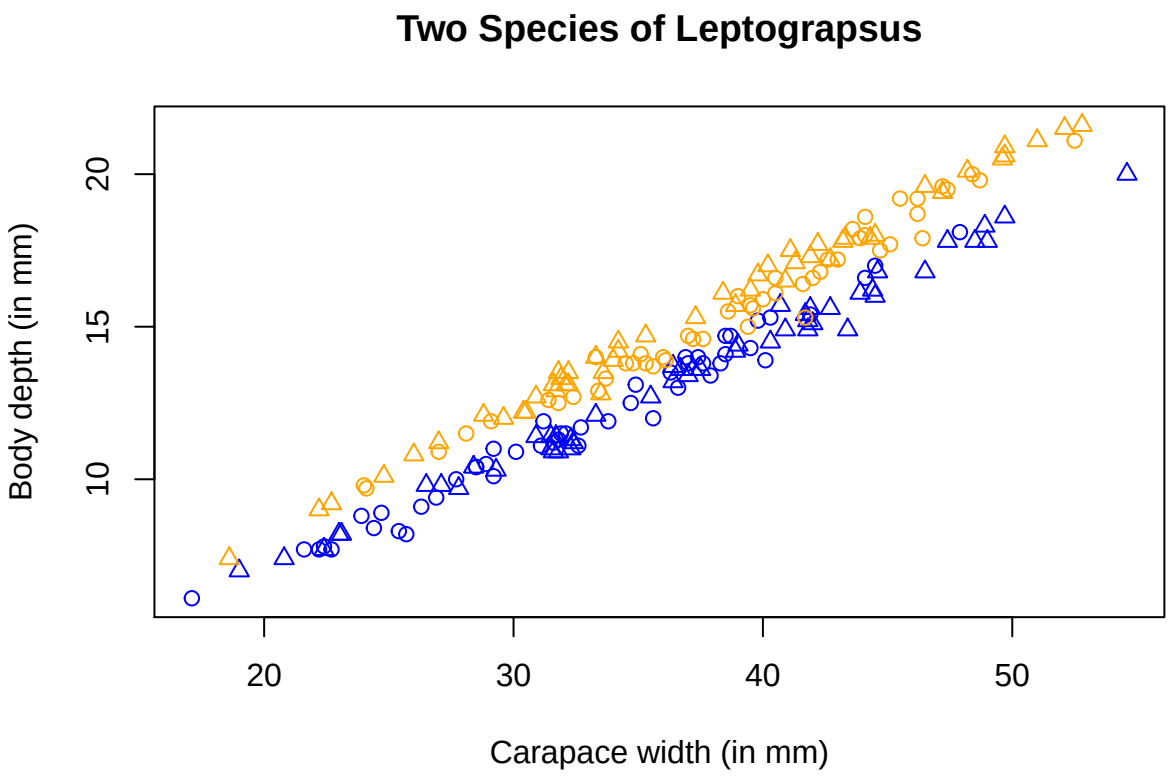
7.

```
plot(crab$CW, crab$BD, xlab="Carapace width (in mm)", ylab="Body depth (in mm)",
     col=unclass(crab$sp), pch=unclass(crab$sex))          # Scattterplot with species ...
# ... and gender
title("Two Species of Leptograpsus")
```



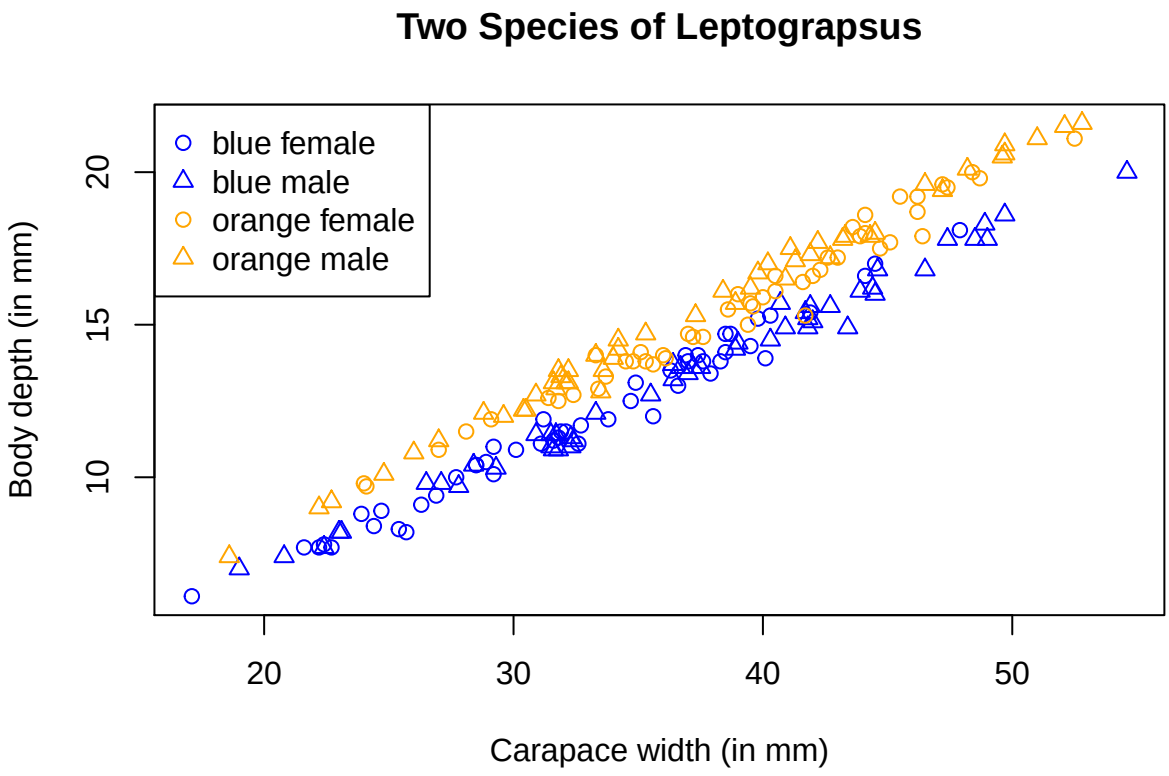
8.

```
my.cols <- c("blue", "orange")
plot(crab$CW, crab$BD, xlab="Carapace width (in mm)", ylab="Body depth (in mm)",
     col=my.cols[unclass(crab$sp)], pch=unclass(crab$sex))
title("Two Species of Leptograpsus")
```



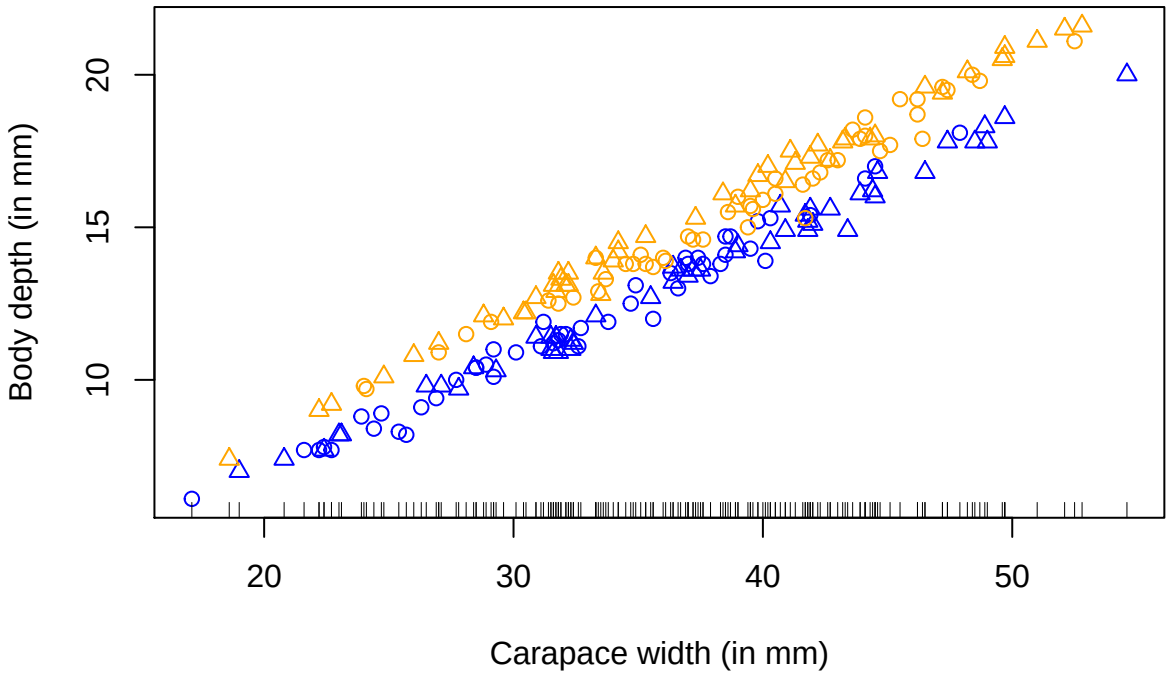
9.

```
plot(crab$CW, crab$BD, xlab="Carapace width (in mm)", ylab="Body depth (in mm)",
     col=my.cols[unclass(crab$sp)], pch=unclass(crab$sex))
legend("topleft",pch=c(1,2,1,2),col=c("blue","blue","orange","orange"),
       c("blue female", "blue male", "orange female", "orange male"))
title("Two Species of Leptograpsus")
```



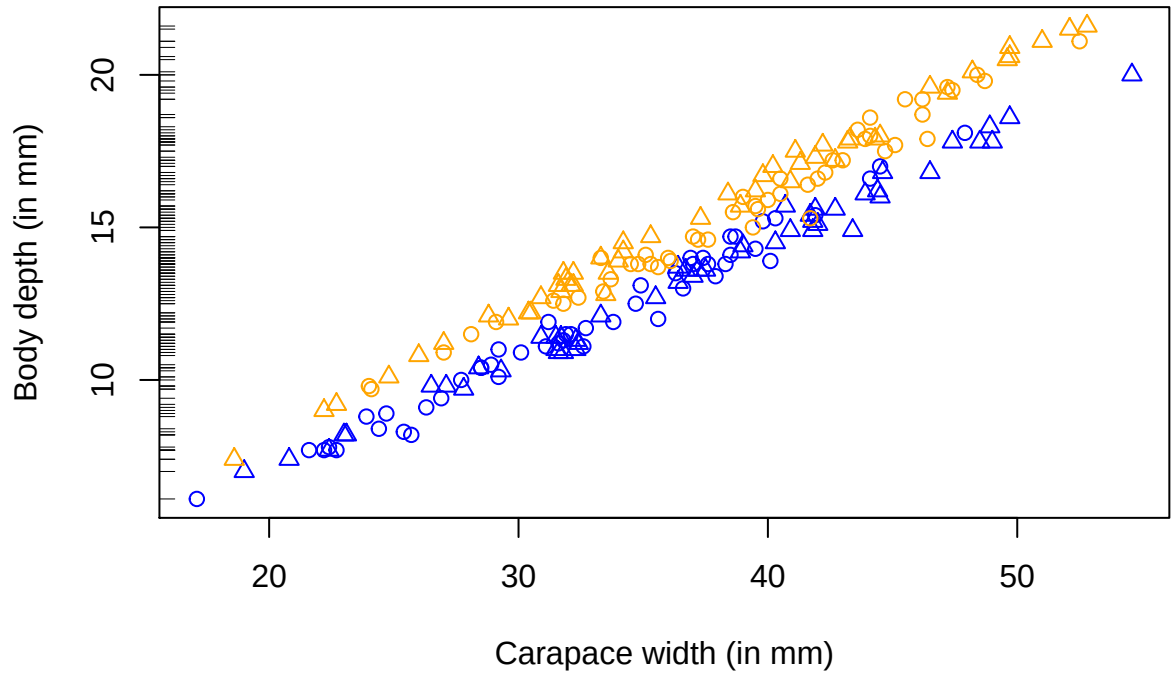
10.

```
plot(crab$CW, crab$BD, xlab="Carapace width (in mm)", ylab="Body depth (in mm)",
     col=my.cols[unclass(crab$sp)], pch=unclass(crab$sex))
rug(crab$CW)
```




```
plot(crab$CW, crab$BD, xlab="Carapace width (in mm)", ylab="Body depth (in mm)",
     col=my.cols[unclass(crab$sp)], pch=unclass(crab$sex))
```

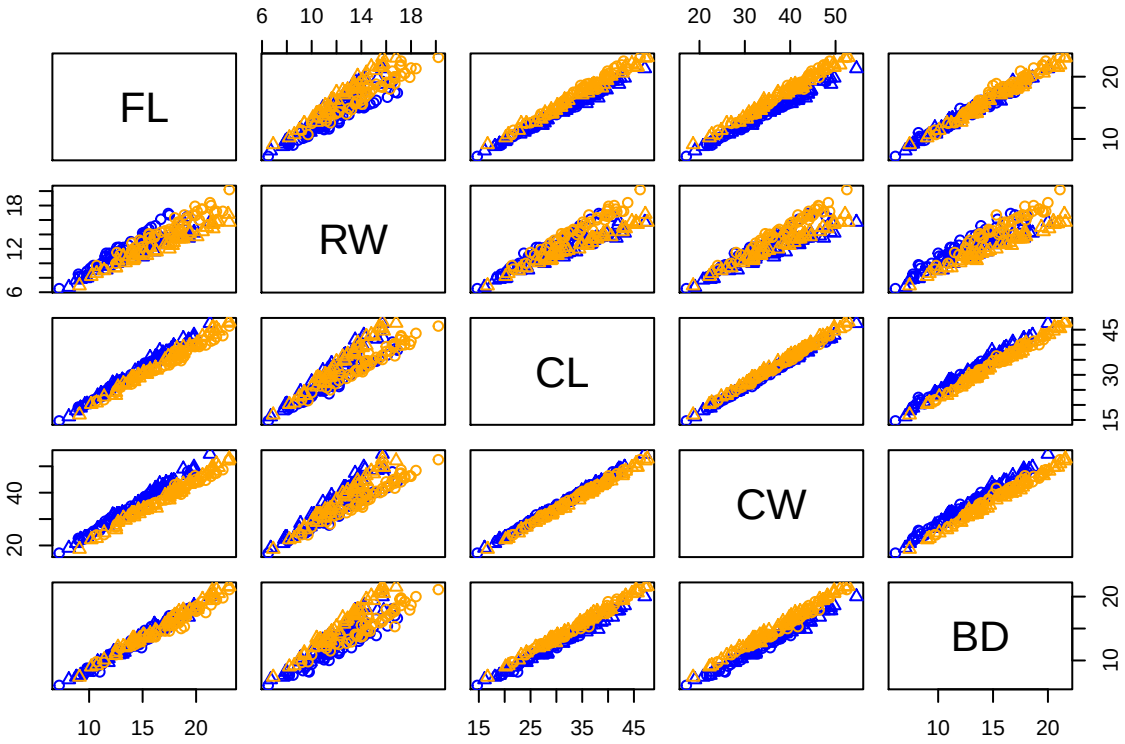
```
rug(crab$BD, side=2)
```



The function rug adds a rug representation of the marginal distribution of the data to the axes. Essentially every vertical line corresponds to one observation.

11.

```
pairs(crab[,4:8], col=my.cols[unclass(crab$sp)], pch=unclass(crab$sex))
```

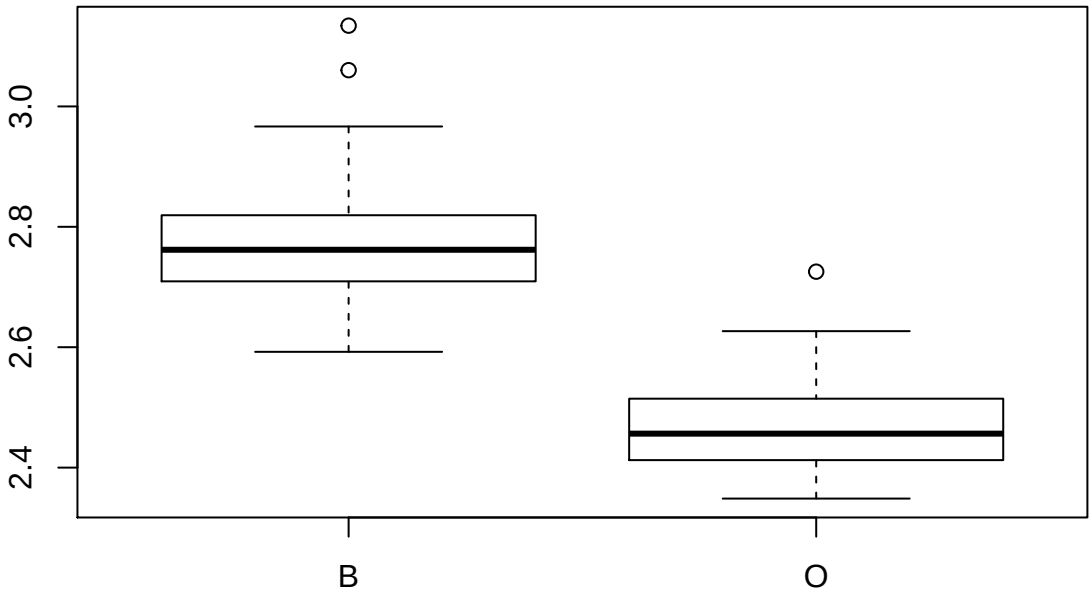


12. Most of the variability in the plots comes down to the overall size of the crab, which is probably mostly

related to age. The variables CW and BD allow for some separation between the species if they are used concurrently.

13.

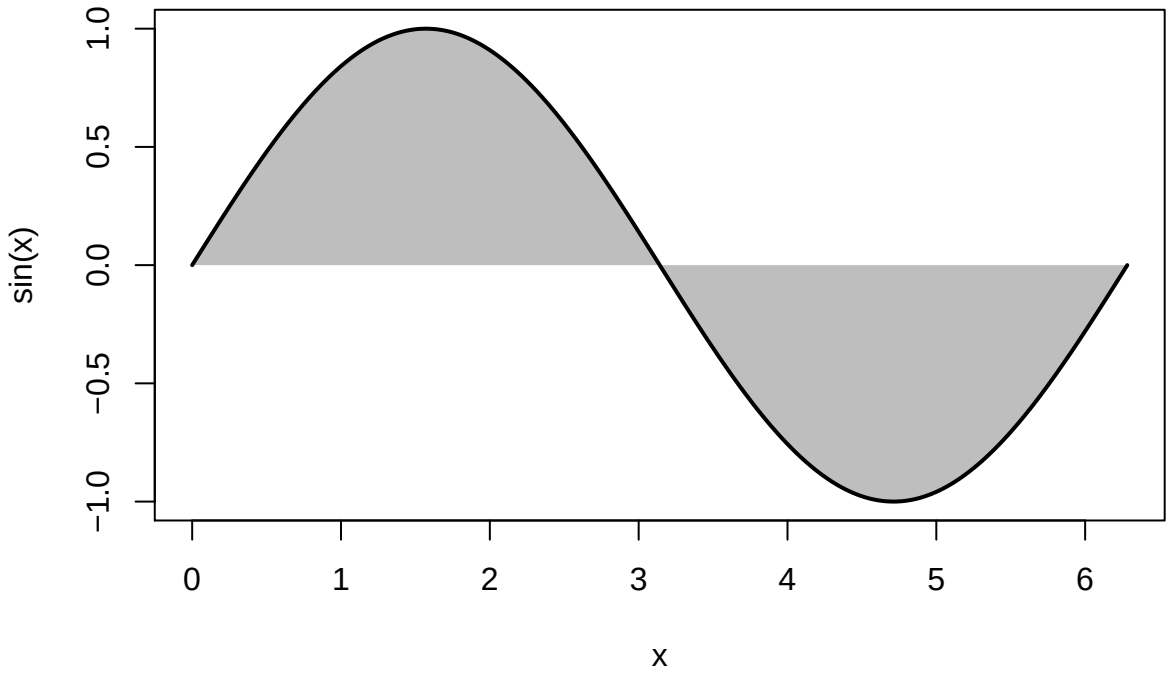
```
ratio <- crab$CW/crab$BD
boxplot(ratio ~ crab$sp)
```



The ratio seems to be very good at telling the two species apart.

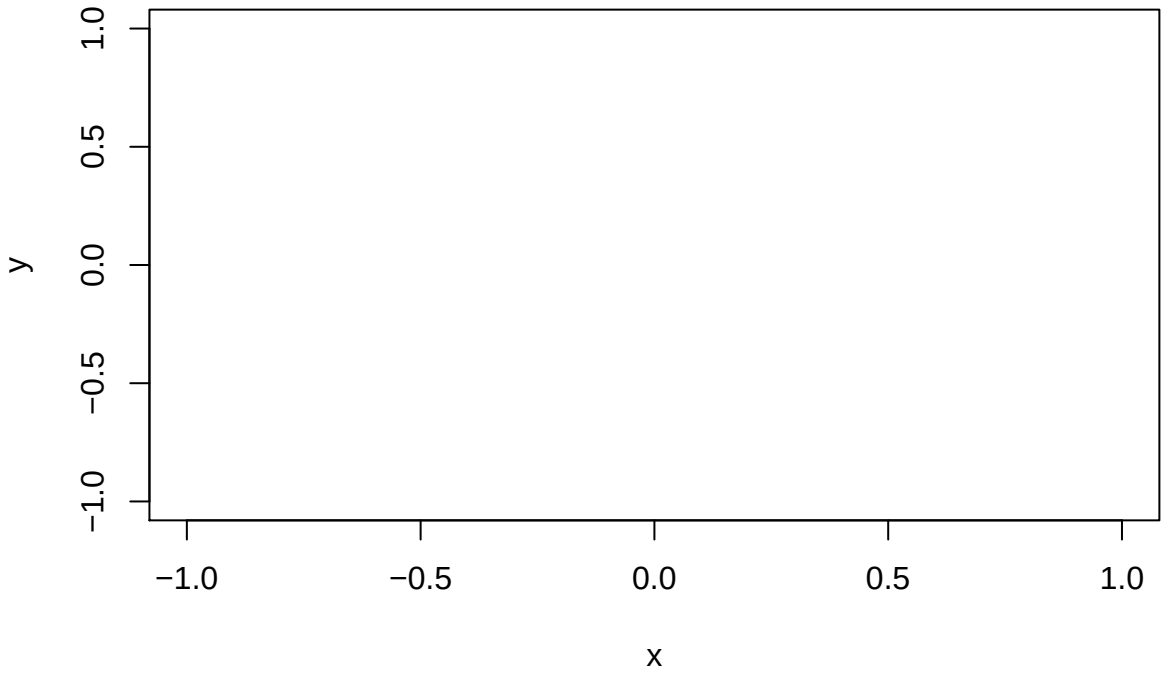
Task 3

```
x <- seq(0, 2*pi, length.out=250)
y <- sin(x)
plot(x, y, type="n", ylab="sin(x)")              # Set up the plotting region
polygon(x, y, col="grey", border=NA)             # Add the polygon
lines(x, y, lwd=2)
```



Task 4 1.

```
plot(NULL, xlim=c(-1,1), ylim=c(-1,1), xlab="x", ylab="y")   # Set up plotting region
points <- locator(n=15, type="p")
```

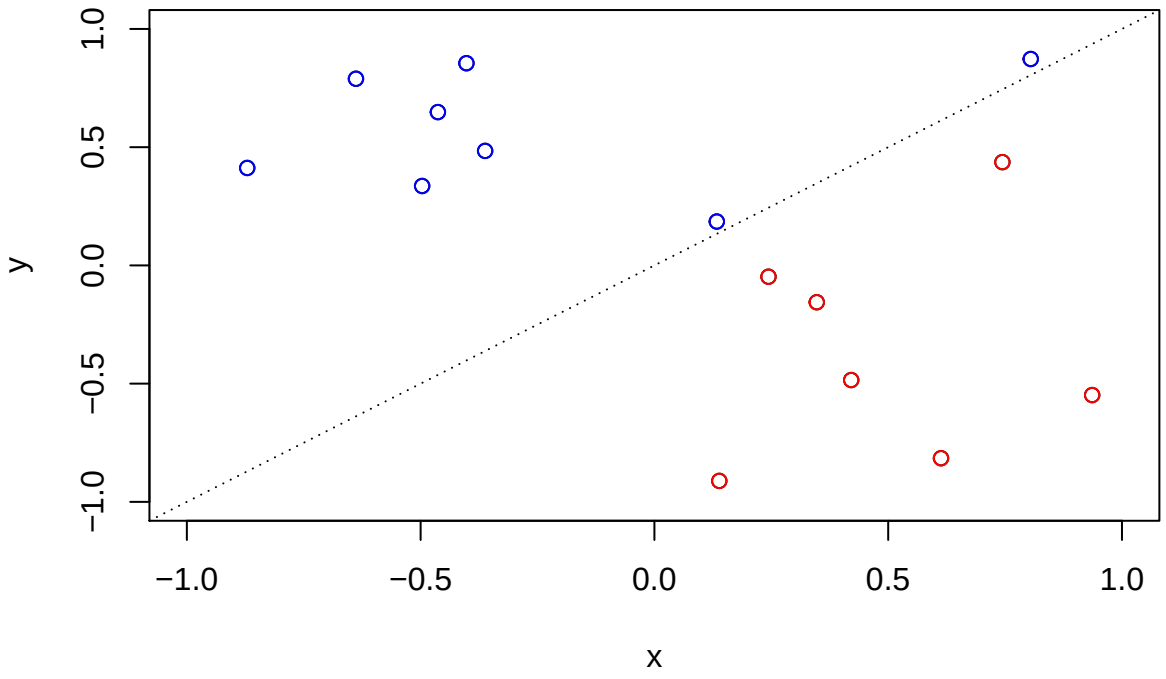



```
# Read in 15 points from mouse
```

2. and 3.

```
plot(points, xlim=c(-1,1), ylim=c(-1,1), xlab="x", ylab="y")
colours <- rep("blue", 15)                          # Create colour vector
colours[points$x>points$y] <- "red"                 # Set colour for points above the ...
                                                 # ... bisector to red.
```

```r
points(points, col=colours)                    # Draw points in correct color
# Part 3
abline(0, 1, lty=3)
```



Intro to R Programming: Lab 5 - Solutions

Task 1

We start with creating the vectors `x` and `y`:

```
x <- c(1,2,9)
y <- c(2,6,4)
```

The loop can be implemented as follows:

```
n <- length(x)                # Get length of x
z <- numeric(n)               # Create empty vector of same length as x
for (i in 1:n) {
  z[i] <- x[i] * y[i]         # Set z[i] to x[i] * y[i]
}
z
```

```
## [1]  2 12 36
```

This is equivalent to

```
z <- x*y
z
```

```
## [1]  2 12 36
```

Task 2

```
x <- c(1,2,9)                             # Create vector x
cumsum.x <- numeric(length(x))            # Create empty vector to hold result
cumsum.x[1] <- x[1]                       # Set first entry
for (i in 2:length(x))
  cumsum.x[i] <- cumsum.x[i-1]+x[i]       # Set remaining entries

cumsum.x                                  # Print result
```

```
## [1]  1  3 12
```

```
cumsum(x)                                 # Compare to built-in function
```

```
## [1]  1  3 12
```

Task 3

1.

```
x <- 1                                    # Set initial value (arbitrary)
for (i in 1:50) {                         # Repeat at most 50 times
  x <- 1 + 1/x                            # Update x
  }
x                                         # Print result
```

```
## [1] 1.618034
```

```
(1+sqrt(5)) / 2                           # Compare to desired answer
```

```
## [1] 1.618034
```

2.

```
x <- 1                                  # Set initial value (arbitrary)
for (i in 1:50) {                       # Repeat at most 50 times
  old.x <- x                            # Store old value
  x <- 1 + 1/x                          # Update x
if (abs(old.x-x)<10e-10)                # Check for convergence
    break
}
x
```

```
## [1] 1.618034
```

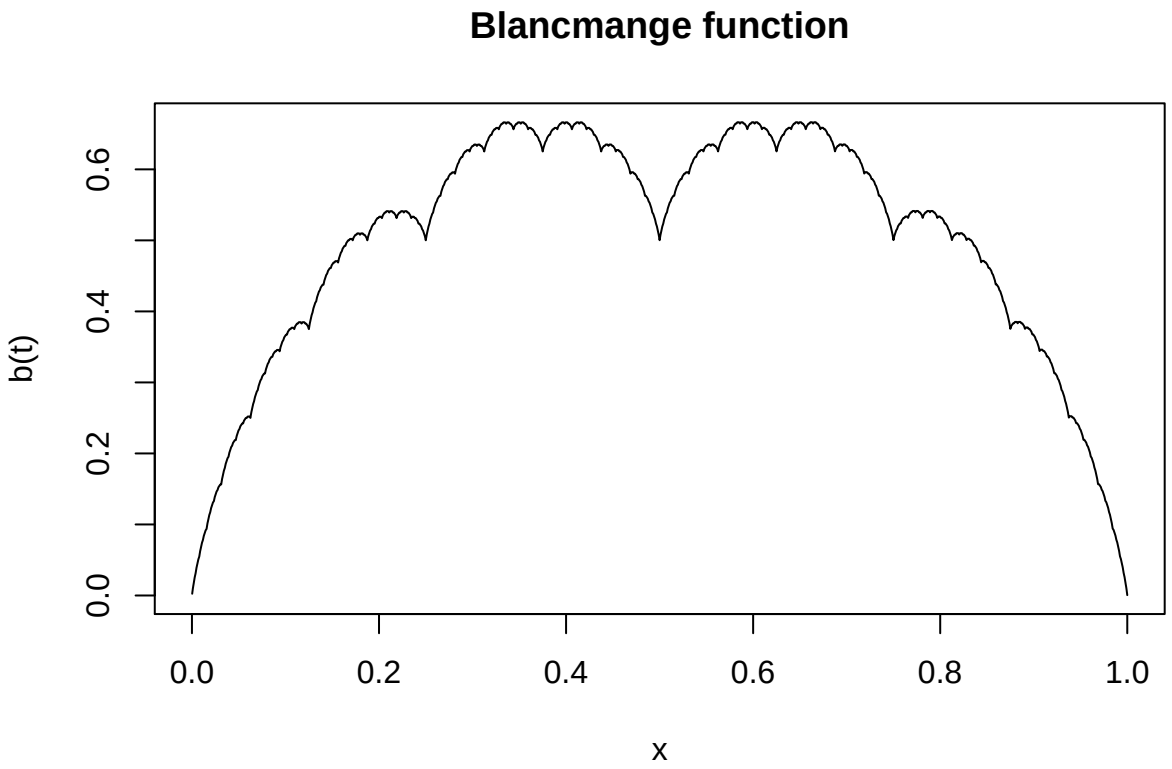
```
i
```

```
## [1] 23
```

Task 4

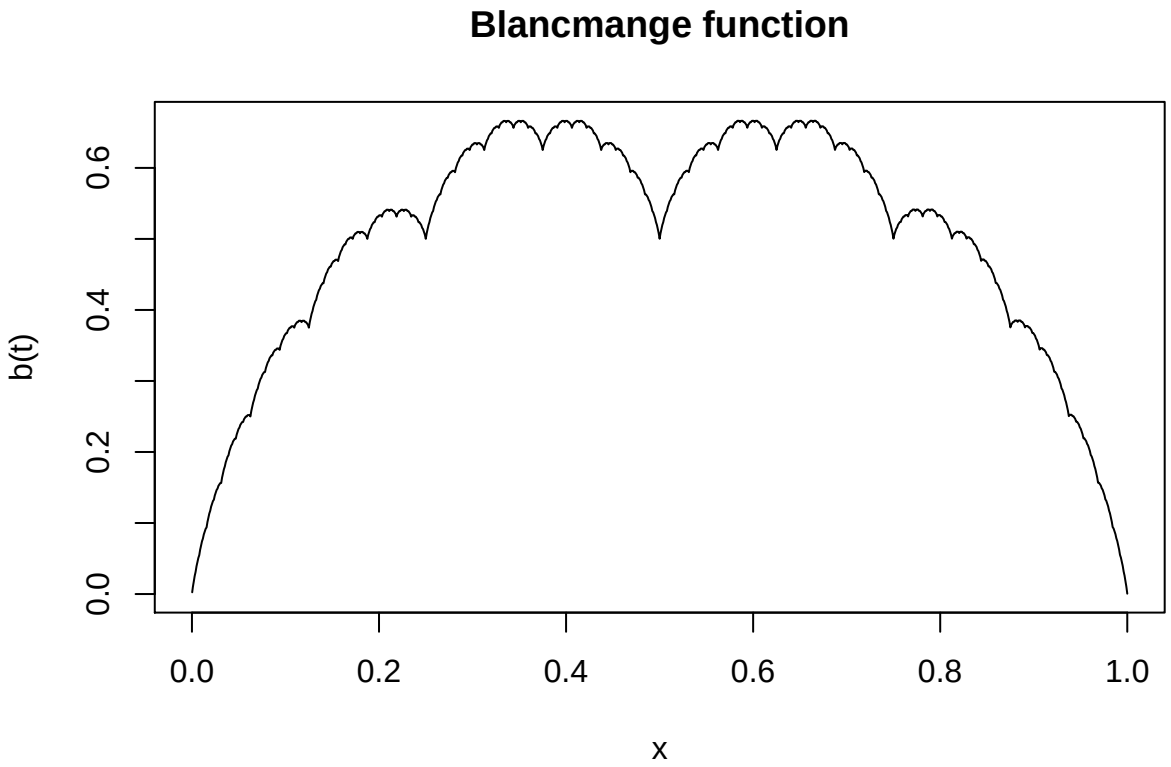
1.

```
b <- numeric(4096)                      # Initalise b to all 0's
t <- c(1:2048,2048:1)/4096              # Initial value of t
w <- 1/2                                # Set constant w
for (i in 1:10) {                       # Repeat 10 times ...
  b <- b + t                            # Update b
  t <- w * t[seq(2, 4096, by=2)]        # Update t (1st part)
  t <- c(t,t)                           # Repeat t twice
}
x <- 1:4096 / 4096                      # Set x coordinates for plot
plot(x, b, type="l", ylab="b(t)")       # Plot function
title("Blancmange function")            # Add title
```



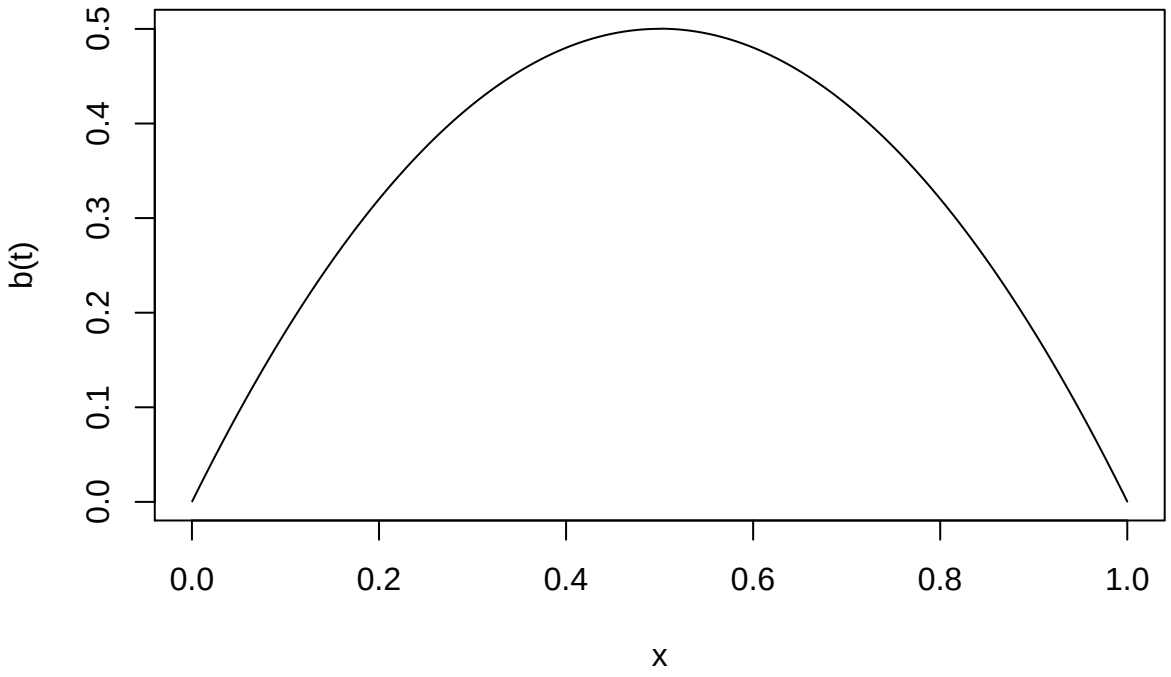
We can exploit the recycling rules in R and use

```r
b <- numeric(4096)                          # Initalise b to all 0's
t <- c(1:2048,2048:1)/4096                  # Initial value of t
w <- 1/2                                    # Set constant w
for (i in 1:10) {                           # Repeat 10 times ...
  b <- b + t                                # Update b
  t <- w * t[seq(2, length(t), by=2)]       # Update t
}
x <- 1:4096 / 4096                          # Set x coordinates for plot
plot(x, b, type="l", ylab="b(t)")           # Plot function
title("Blancmange function")                # Add title
```



2. When setting $w = \frac{1}{4}$ we obtain a perfectly well-behaved parabola.

```r
b <- numeric(4096)                          # Initalise b to all 0's
t <- c(1:2048,2048:1)/4096                  # Initial value of t
w <- 1/4                                    # Set constant w
for (i in 1:10) {                           # Repeat 10 times ...
  b <- b + t                                # Update b
  t <- w * t[seq(2, length(t), by=2)]       # Update t
}
x <- 1:4096 / 4096                          # Set x coordinates for plot
plot(x, b, type="l", ylab="b(t)")           # Plot function
```



Task 5

1.

```
n <- 10                                    # Simulate points
coords <- matrix(rnorm(2*n), ncol=2)
plot(coords, pch=16, xlab="x", ylab="y")   # Draw the points
for (i in 1:n)                             # For all pairs of points ...
  for (j in 1:n)
    lines(coords[c(i,j),], col="blue")     # Connect i-th and j-th point
```

The `for` loop is not optimal. It connects each pair of points twice ($i \to j$ and $j \to i$), and it also connects each point with itself, which is not necessary either. Thus a better solution would be

```
plot(coords, pch=16, xlab="x", ylab="y")
for (i in 1:(n-1))
  for (j in (i+1):n)
    lines(coords[c(i,j),], col="blue")
```

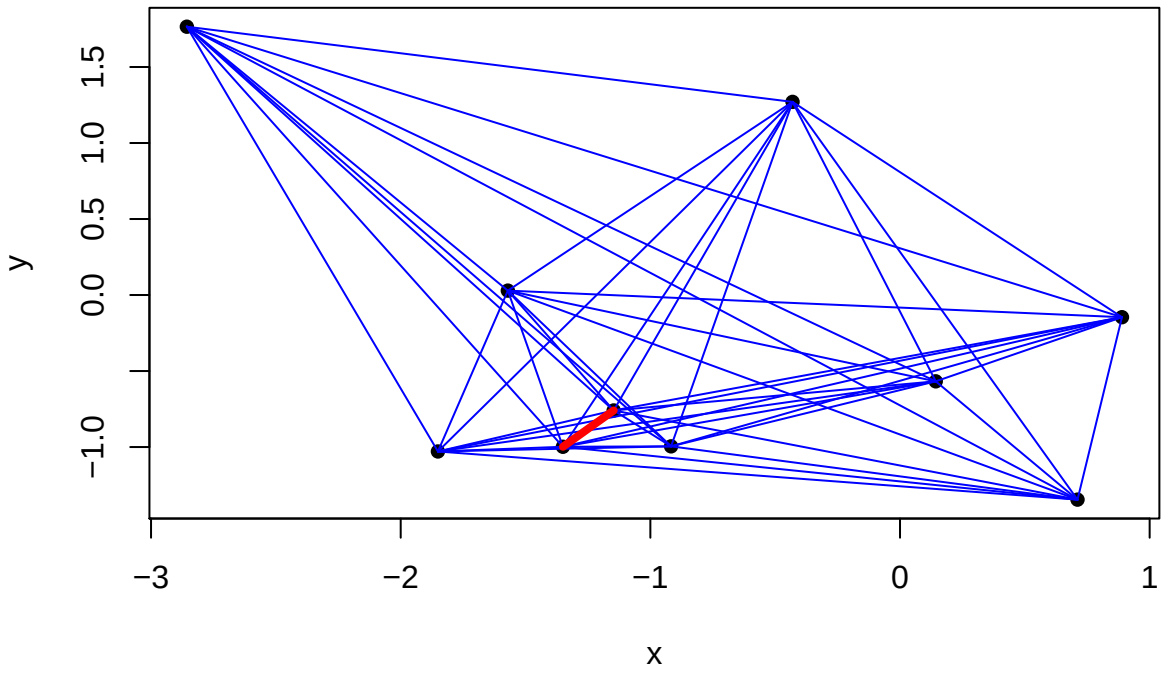
This setup makes sure that $i < j$, thus each pair of points is only connected once.

2. Full solution to parts 1. and 2.

```
n <- 10                                    # Simulate points
coords <- matrix(rnorm(2*n), ncol=2)
plot(coords, pch=16, xlab="x", ylab="y")
for (i in 1:(n-1))
  for (j in (i+1):n)
    lines(coords[c(i,j),], col="blue")

closest.pair <- c(NA,NA)                   # Initialise closest pair
closest.distance <- Inf                    # Initialise closest distance
for (i in 1:(n-1))                         # Go through all pairs of points
  for (j in (i+1):n){
    dist <- sum((coords[i,]-coords[j,])^2) # Compute (squared) distance
    if (dist<closest.distance) {           # If we find a pair which is closer ...
```

```
      closest.pair <- c(i,j)                # ... store it ...
      closest.distance <- dist              # ... along with the distance
    }
  }
lines(coords[closest.pair,], col="red", lwd=4)
```




```
                                        # Connect two closest points
```

Intro to R Programming: Lab 7 - Solutions

Task 1

1. In positional form, the position of the arguments determines how R matches the arguments:

```
dbinom(3, 10, 0.5)
```

   In named form, the names of the arguments determine how R matches the arguments:

```
dbinom(size=10, prob=0.5, x=3)
```

2. `dbinom(size=4, 1, 0.5)` is equivalent to `dbinom(size=4, x=1, prob=0.5)`, thus it evaluates the p.m.f. of the $\mathsf{Bi}(4, 0.5)$ distribution at $x = 1$.

Task 2

The function finds the minimum value of the vector supplied as argument.

Task 3

```
# Compute least squares regression estimate
# Argument: x (vector or matrix of covariates), y (response)
# Output: Least-squares estimate of regression coefficients
least.squares <- function(x, y) {
  X <- cbind(1, x)
  if (length(y)!=nrow(x))
    stop("Number of observations in x and y do not match.")
  if (nrow(x)<ncol(x))
    stop("More parameters than observations. No unique solution exists.")
  XtX <- t(X)%*%X
  Xty <- t(X)%*%y
  solve(XtX, Xty)
}
```

Task 4

```
A <- rbind(c(1, 8, 5),
           c(4, 3, 6))
# The code computes the sums of the columns of the matrix $A$.
apply(A,2,sum)
```

```
## [1]  5 11 11
```

```
# The code computes the sums of the rows of the matrix $A$.
apply(A,1,sum)
```

```
## [1] 14 13
```

```
# The code subtracts the vector $(1,3,5)$ from each row of the matrix $A$,
# i.e. it subtract $1$ from the first column, $3$ from the second column, etc.
sweep(A,2,c(1,3,5),"-")
```

```
##      [,1] [,2] [,3]
## [1,]    0    5    0
## [2,]    3    0    1
```

```r
# The code divides each column of the matrix by the vector $(8,6)$,
#i.e. it divides the first row by $8$ and the second row by $6$.
sweep(A,1,c(8,6),"/")
```

```
##           [,1] [,2]  [,3]
## [1,] 0.1250000  1.0 0.625
## [2,] 0.6666667  0.5 1.000
```

Task 5

1.

```r
binomial.coefficient <- function(n,k) {
  factorial(n) / (factorial(n-k)*factorial(k))
}
binomial.coefficient(6, 3)
```

```
## [1] 20
```

2.

```r
binary.entropy <-  function(p) {
  -p*log(p)-(1-p)*log(1-p)
}
```

3.

```r
approx.lbincoef <- function(n, k) {
  n*binary.entropy(k/n)
}
```

4.

```r
log(binomial.coefficient(9000, 4000))
```

```
## Warning in factorial(n): value out of range in 'gammafn'
```

```
## Warning in factorial(n - k): value out of range in 'gammafn'
```

```
## Warning in factorial(k): value out of range in 'gammafn'
```

```
## [1] NaN
```

```r
                                  # fails (numerically unstable)
approx.lbincoef(9000,4000)        # gives an approximation
```

```
## [1] 6182.654
```

Task 6

1. We can either make use of an `if ...else ...` statement:

```r
# Compute the Box-Cox transform (using if ... else ... statement)
# Arguments: y (vector of data), lambda (parameter of transform)
# Returns: Box-Cox transform of y
box.cox <- function(y, lambda=0) {
  if (lambda==0) {
    result <- log(y)
  } else {
    result <- (y^lambda-1) / lambda
  }
```

```
  result
}
```

or use the function `ifelse`. When using the function `ifelse` we have to be careful: the length of the first argument determines the length of its result. Thus we repeat `lambda` such that it has the same length as `y`.

```
# Compute the Box-Cox transform (using ifelse function)
# Arguments: y (vector of data), lambda (parameter of transform)
# Returns: Box-Cox transform of y
box.cox <- function(y, lambda=0) {
  ifelse(rep(lambda, length(y))==0, log(y), (y^lambda-1) / lambda)
}
```

Note that we want to use the same `lambda` for the entire vector `y`, i.e. `lambda` is a scalar.

2.

```
library(MASS)
data(mammals)
box.cox(mammals$brain, lambda=0)
```

```
##  [1]  3.7954892  2.7408400  2.0918641  6.0473722  4.7833164  4.7449321
##  [7]  4.5870062  1.7047481  4.0604430  1.8562980  1.3862944  1.7404662
## [13]  1.8870696 -1.9661129  0.0000000  2.3795461  2.5095993  1.8405496
## [19]  8.4344635 -1.2039728  6.0378709  6.4846352  1.2527630  4.7449321
## [25]  3.2425924  1.6094379  2.8622009  6.5220928  6.0063532  5.7838252
## [31]  2.5095993  7.1853870  8.6503245  1.3609766  5.1873858  4.0253517
## [37]  2.8332133  0.0000000 -0.9162907 -1.3862944  2.5257286  6.1944054
## [43]  2.4932055  5.1647860  5.0562458  6.0867747  5.1901752  0.8754687
## [49]  4.3944492  3.0445224  3.6686767  0.6418539  0.1823216  1.0986123
## [55] -1.1086626  5.1929569  3.2188758  5.1298987  0.9555114  2.4336134
## [61]  0.9162907  3.9199912
```

```
box.cox(mammals$brain, lambda=0.1)
```

```
##  [1]  4.6162513  3.1532529  2.3267476  8.3077105  6.1338045  6.0719948
##  [7]  5.8201701  1.8586778  5.0086904  2.0397646  1.4869835  1.9011104
## [13]  2.0768701 -1.7849010  0.0000000  2.6865161  2.8525858  2.0208189
## [19] 13.2436377 -1.1343185  8.2903241  9.1259991  1.3346158  6.0719948
## [25]  3.8300578  1.7461894  3.3138545  9.1977747  8.2327679  7.8315188
## [31]  2.8525858 10.5143326 13.7508316  1.4579378  6.7990725  4.9561152
## [37]  3.2753167  0.0000000 -0.8755646 -1.2944944  2.8733329  8.5788834
## [43]  2.8315328  6.7611497  6.5802076  8.3799899  6.8037590  0.9149343
## [49]  5.5184557  3.5588211  4.4320693  0.6629006  0.1839938  1.1612317
## [55] -1.0494156  6.8084339  3.7972966  6.7027765  1.0026509  2.7552943
## [61]  0.9595823  4.7993641
```

Task 7

```
midrange <- function(x) {
  (min(x)+max(x))/2
}

n.sim <- 1e4
n <- 100

data <- matrix(rnorm(n.sim*n), ncol=n)
means <- apply(data, 1, mean)
```

```
var(means)
```

```
## [1] 0.009935067
```

```
medians <- apply(data, 1, median)
var(medians)
```

```
## [1] 0.01535831
```

```
midranges <- apply(data, 1, midrange)
var(midranges)
```

```
## [1] 0.09393282
```

Another more compact solution, involving even more `apply`'s is

```
n.sim <- 1e4
n <- 100
data <- matrix(rnorm(n.sim*n), ncol=n)
stats <- apply(data, 1, function(x) c(mean=mean(x), median=median(x),
                                      midrange=(min(x)+max(x))/2))
apply(stats, 1, var)
```

```
##       mean     median   midrange
## 0.01006624 0.01554431 0.09297406
```

Intro to R Programming: Lab 8 - Solutions

Task 1

1.

```
cv <- function(x)
  sd(x)/mean(x)
```

2.

```
apply(iris[,1:4],2,cv)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##    0.1417113    0.1425642    0.4697441    0.6355511
```

Task 2

1.

```
# Compute the direct asymptotic confidence interval for a proportion
# Arguments: theta (observed proportion), n (sample size), alpha (1-significance level)
# Returns: vector of length 2 containing the lower and upper bound of the CI
ci.proportion.direct <- function(theta, n, alpha=0.05) {
  # Compute standard deviation of x/n
  sd.theta <- sqrt(theta*(1-theta)/n)
  # Compute confidence interval
  ci <- theta + c(-1,1) * qnorm(1-alpha/2) * sd.theta
  # Label the two values in ci
  names(ci) <- c("lower", "upper")
  ci
}
```

We can compute the 95% confidence interval for an observed proportion of $\theta = 0.7$ and a sample size of $n = 50$ as follows:

```
ci.proportion.direct(n=5, theta=0.7)
```

```
##     lower     upper
## 0.2983269 1.1016731
```

2.

We just add another parameter `x` and give `theta` the default value $\frac{x}{n}$. Thus if the user calls the function using `x` as argument (e.g. in `ci.proportion.direct(n=20, x=10)`), `theta` is computed automatically. If the user calls the function using `theta` as argument (e.g. in `ci.proportion.direct(n=20, theta=0.5)`), it does not matter if `x` is not given, as it is not used in the function except as a default value for `theta`.

```
# Compute the direct asymptotic confidence interval for a proportion
# (more flexible version)
# Arguments: x (observed number of "successes") or theta (observed proportion),
# n (sample size), alpha (1-significance level)
# Returns: vector of length 2 containing the lower and upper bound of the CI
ci.proportion.direct <- function(x, n, theta=x/n, alpha=0.05) {
  # Compute standard deviation of x/n
  sd.theta <- sqrt(theta*(1-theta)/n)
  # Compute confidence interval
  ci <- theta + c(-1,1) * qnorm(1-alpha/2) * sd.theta
```

```
  # Label the two values in ci
  names(ci) <- c("lower", "upper")
  ci
}
```

3.

We just add one line of code restricting the confidence interval to $[0, 1]$.

```
# Compute the direct asymptotic confidence interval for a proportion
# (more flexible version, not outside [0,1])
# Arguments: x (observed number of "successes") or theta (observed proportion),
#            n (sample size), alpha (1-significance level)
# Returns: vector of length 2 containing the lower and upper bound of the CI
ci.proportion.direct <- function(x, theta=x/n, n, alpha=0.05) {
    # Compute standard deviation of x/n
  sd.theta <- sqrt(theta*(1-theta)/n)
  # Compute confidence interval
  ci <- theta + c(-1,1) * qnorm(1-alpha/2) * sd.theta
  # Label the two values in ci
  names(ci) <- c("lower", "upper")
  # Restrict CI to [0,1]
  ci <- c(max(ci[1],0), min(ci[2],1))
  ci
}
```

Task 3

```
#' Determine the day of the week using Zeller's congruence formula
#' @param day day of the month
#' @param month month as an integer
#' @param year year as a four-digit integer
#' @return day of the week
zeller <- function(day, month, year) {
  q <- day
  m <- (month+9) %% 12 + 1
  if (month<=2)
    year <- year - 1
  k <- year %% 100
  j <- floor(year/100)
  h <- (floor(2.6*m-0.2)+q+k+floor(k/4)+floor(j/4)-2*j+6) %% 7 + 1
  c("Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday")[h]
}

# we can use
zeller(27,11,2017)
```

```
## [1] "Monday"
```

Task 4

1.

```
lcg <- function(n, a=2^16+3, M=2^31, c=0, z0=1) {
  x <- numeric(n)
  z <- z0
  for (i in 1:n) {
    z <- (a*z + c) %% M
```

```
    x[i] <- z/M
  }
  x
}
```

2.

```
x <- lcg(300000)
```

3.

```
X <- matrix(x, ncol=3, byrow=TRUE)
```

4.

```
library(rgl)
plot3d(X)
```

Rotating the 3-dimensional scatterplot reveals that the numbers generated lie on only 15 hyperplanes in the 3-dimensional unit cube, so they are far from being uniformly spread across the 3-dimensional unit cube. According to an IBM salesperson at the time "each number is random individually, but we don't guarantee that more than one of them is random" — whatever this means.

The default values of the parameters a, M and c given in part (a) and used by RANDU are thus extremely poor. Although other choices of a, M and c give better RNG, one can prove that whatever the choice of a, M and c, the pseudo-random numbers generated by a linear congruential generator lie on a finite, and often very small number of parallel hyperplanes. The number of hyperplanes depends on the choice of a, M and c. Thus no one should be using linear congruential generators any more.

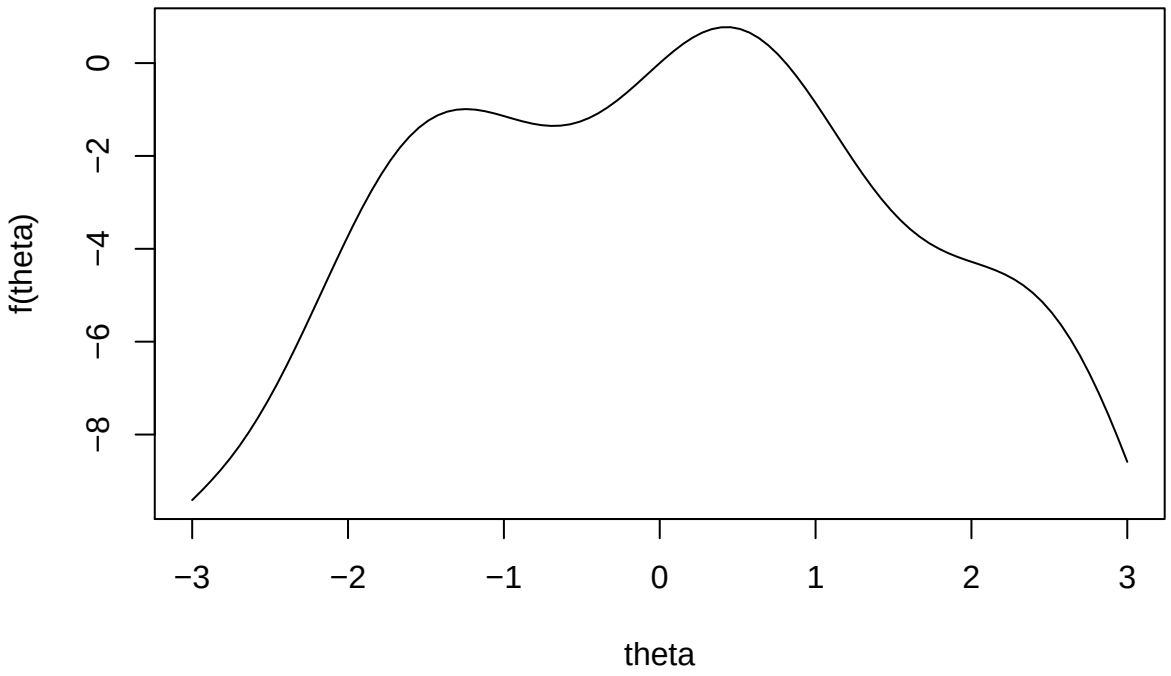
Intro to R Programming: Lab 8 - Solutions

Task 1

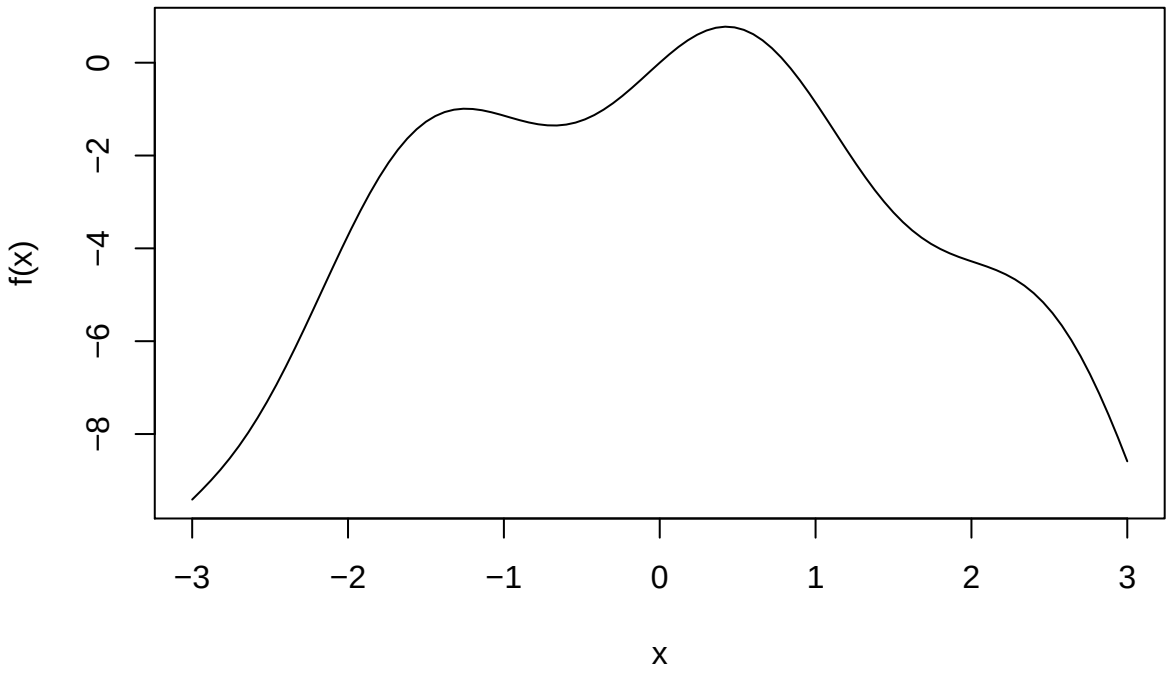
All parts

```
f <- function(theta) sin(3*theta) - theta^2
f.d <- function(theta) 3*cos(3*theta) - 2*theta
f.dd <- function(theta) -9*sin(3*theta) - 2

theta <- seq(-3, 3, length.out=100)
plot(theta, f(theta), type="l")
```




```
curve(f, from=-3, to=3)           # Alternative using curve
```




```
optimize(f, lower=-3, upper=3, maximum=TRUE)
```

```
## $maximum
## [1] 0.4273252
##
## $objective
## [1] 0.7759736
```

```
theta <- 2
for (h in 1:50) {
  theta.old <- theta
  theta <- theta - f.d(theta) / f.dd(theta)
  if (abs(theta-theta.old)<1e-10)
    break
}
```

Depending on the initial value of `theta` we obtain different results with Newton's method.

For $\theta_0 = -4$ Newton's method converges to the local maximum at $\theta = -1.2446$. For $\theta_0 = 1.5$ Newton's method converges to the local minimum at $\theta = -0.6806$. For $\theta_0 = 2$ Newton's method converges to the global maximum at $\theta = 0.4273$

Task 2

I will use the brain weights from the `mammals` data from `MASS`.

```
x <- MASS::mammals$brain
```

1. The minimiser

```
f <- function(theta, x) {
  sum((theta-x)^2)
}
optimize(f, range(x), x=x)
```

```
## $minimum
## [1] 283.1342
```

```
##
## $objective
## [1] 52790554
```

turns out to be the mean of `x`:

```
mean(x)
```

```
## [1] 283.1342
```

2. The minimiser

```
f <- function(m, x) {
  sum(abs(m-x))
}
optimize(f, range(x), x=x)
```

```
## $minimum
## [1] 17.3734
##
## $objective
## [1] 17202.48
```

turns out the be the median of `x`.

```
median(x)
```

```
## [1] 17.25
```

Task 3

All parts

```
f <- function(z) {
  pnorm(z) - 0.95
}
uniroot(f, c(-10,10))
```

```
## $root
## [1] 1.644854
##
## $f.root
## [1] 8.678571e-08
##
## $iter
## [1] 9
##
## $init.it
## [1] NA
##
## $estim.prec
## [1] 0.0001105832
```

```
qnorm(0.95)                 # Check against qnorm to confirm
```

```
## [1] 1.644854
```

```
z <- 0                      # Newton's method for root finding ...
for (h in 1:100) {
  z.old <- z
  z <- z - (pnorm(z)-0.95)/dnorm(z)
```

```
  if (abs(z-z.old)<1e-10)
    break
}
z
```

```
## [1] 1.644854
```

Task 4

1. We start with defining the loglikelihood function as given in the question. To be able to use the function in part 2 we have to take in the parameters as a single argument (a vector of length 5).

```
gmm.loglik <- function(par, x) {
  p <- par[1]
  mu.1 <- par[2]
  sigma.1 <- par[3]
  mu.2 <- par[4]
  sigma.2 <- par[5]
  sum(log(p * dnorm(x,mu.1,sigma.1) + (1-p) * dnorm(x,mu.2,sigma.2)))
}
```

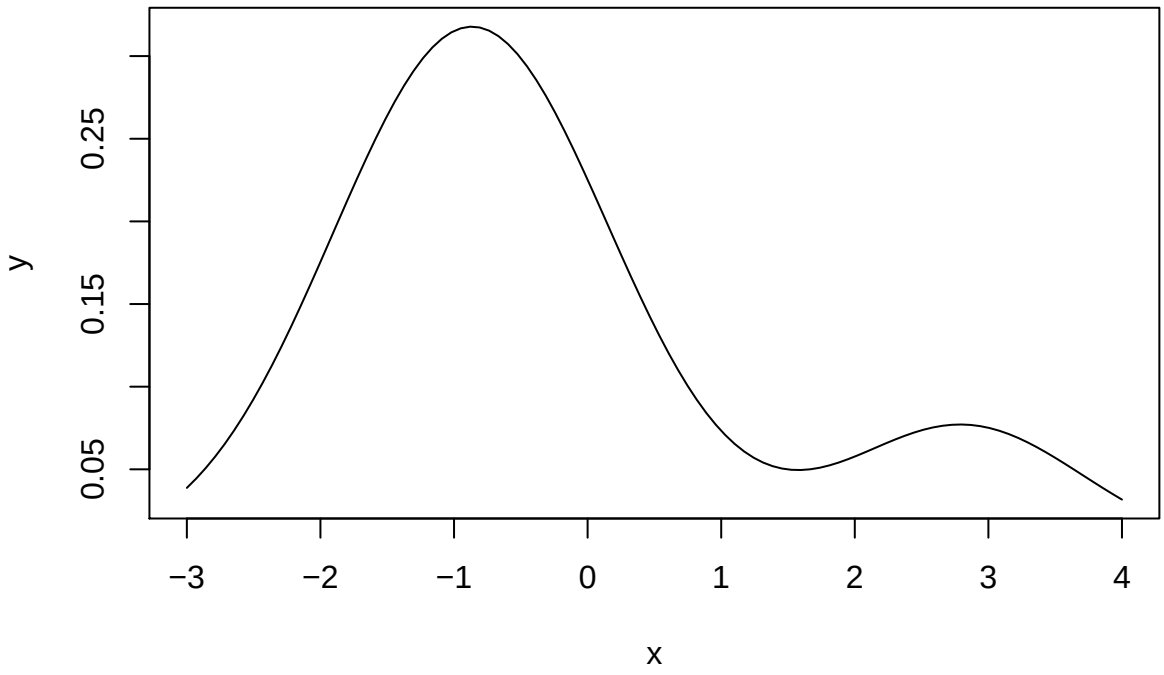
2. We can now use `optim` to maximise this function.

```
mixturedata <- readRDS("mixturedata.RDS")
opt.par <- optim(fn=gmm.loglik, par=c(0.5,-1,1,1,1), x=mixturedata,
                 control=list(fnscale=-1, maxit=1000))
opt.par
```

```
## $par
## [1]  0.8285764 -0.8664510  1.0818417  2.8128841  0.7995587
##
## $value
## [1] -184.6631
##
## $counts
## function gradient
##      708       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

3.

```
x <- seq(-3,4,length.out=100)
y <- opt.par$par[1]*dnorm(x,opt.par$par[2],sqrt(opt.par$par[3]))+
  (1-opt.par$par[1])*dnorm(x,opt.par$par[4],sqrt(opt.par$par[5]))
plot(x,y,type="l")
```


y
0.05
0.15
0.25
−3
−2
−1
0
1
2
3
4
x