# Lab2

## PROBLEM 1

This problem asks us to create a function that utilizes optim() to approximate a user defined function using a piecewise quadratic function.

We create a function called routine which takes 2 arguments:

- f: the function to be approximated and,
- points: User defined points at which the function will be interpolated

The algorithm is explained as follows:

1. Create a data.frame with user input (x) and corresponding f(x)
2. Define $\hat{f} = a_0 + a_1 x + a_2 x^2$ as a function that takes user points ($x_i$) and parameters ($a_i$)
3. Define the loss function: $\sum_{i=0}^{2} \left( f(x_i) - \hat{f}(x_i) \right)^2$. This function takes 2 inputs: user data ($x_i$) and parameters ($a_i$)
4. run optim() using the BFGS method to minimize the loss function and return those values of $a_i$ which will then help us create the piecewise quadratic function
5. We return only the parameters of the result of optim()

The code for this algorithm is as follows along with a test case for $f(x) = x^3$:

```r
routine = function(f, points){
  df = data.frame(x=points, y=sapply(points, f))
  f_hat = function(x, par) par[1] + par[2]*x + par[3]*x^2
  loss = function(data, par) with(data, sum((f_hat(x, par) - y)^2))
  a = optim(par = rep(0.5,3), fn = loss, data = df ,  method = "BFGS")
  return(a$par)
}

test <- routine(function(x) x^3, c(0.1, 0.55, 0.8))
test
```

```
## [1]  0.044 -0.575  1.450
```

Now we will build a generic function called inter_function(), on the interval [0,1] which will create a user specified number of equal-sized 'n' intervals and the target function will be approximated at each interval. The algorithm is explained as follows:

1. Initialize a data.frame which will store the calculated intervals ($x_{min}$, $x_{mid}$, $x_{max}$) & optimized parameters ($a_i$)
2. The interval is calculated as follows:

- $x_{min}$ = i/n , where i $\epsilon$ [0,n-1] & n is user specified
- $x_{max}$ = i/n , where i $\epsilon$ [1,n] & n is user specified
- $x_{mid}$ = ($x_{min}$ + $x_{max}$)/2

3. Initialize loop to call routine() for each interval and save resulting parameters ($a_i$)
4. Return the data.frame

The code for this function is as follows:

```r
inter_function <- function(f, n) {
  df <- data.frame(min = c(0:(n-1)),
                   mid = c(1:n),
                   max = c(1:n),
                   a0 = c(1:n),
                   a1 = c(1:n),
                   a2 = c(1:n),
                   stringsAsFactors=FALSE)

  df$min <- df$min/n
  df$max <- df$max/n
  df$mid <- (df$min+df$max)/2

  for(i in 1:nrow(df)) {
    ai_par <- routine(f, c(df[i,1] , df[i,2] , df[i,3]) )
                     df[i,4] <- ai_par[1]
                     df[i,5] <- ai_par[2]
                     df[i,6] <- ai_par[3]
  }
  return(df)
}
```

We now apply these function to approximate the following 2 functions

1. $f(x) = -x(1-x)$
2. $f(x) = -x\sin(10\pi x)$

First we check the plot of the actual value of the functions without interpolation

```r
f1 <- data.frame(x1 = seq(from = 0 , to = 1 , by = 0.01))

# true_f1 <- (-f1$x1*(1-f1$x1))
# plot(y=true_f1,
#      x=f1$x1,
#      xlab = "x",
#      ylab = "f(x)",
#      title("Actual f(x) = -x(1-x)"))
#
# true_f2 <- -f1$x1*sin(10*pi*f1$x1)
# plot(y=true_f2,
#      x=f1$x1,
#      xlab = "x",
#      ylab = "f(x)",
#      title("Actual f(x) = -x*sin(10*pi*x)"), t='l')
```

And now we use the approximation functions we have written to call both our functions that need to be approximated

```r
fn1 = function(x) -x*(1-x)
func1 <- inter_function(fn1, n = 100)

f1_hat <- c()
data1 <- c()
for( i in 1:100 ) {
  x= c(func1[i,1], func1[i,2] , func1[i,3])
  y = func1[i,4] + func1[i,5]*x+ func1[i,6]*x**2
  f1_hat = c(f1_hat, y)
  data1 = c(data1, x)
}

#plot(y=f1_hat , x=data1, title("Interpolated f(x), n=100", xlab = "x",
#                                    ylab = "f(x) = -x(1-x)"))
```

Looking at the plot for $f_1(x) = -x(1-x)$ we can see that our approximation seems to be pretty spot on and does not differ in any meaningful way from the true function.

Similarly, for the second function:

```
fn2 = function(x) (-x*sin(10*pi*x))
func2 <- inter_function(fn2, n = 100)

f2_hat <- c()
data2 <- c()
for( i in 1:100 ) {
  x= c(func2[i,1], func2[i,2] , func2[i,3])
  y = func2[i,4] + func2[i,5]*x+ func2[i,6]*x**2
  f2_hat = c(f2_hat , y)
  data2 = c(data2 , x )
}

#plot(y=f2_hat , x=data2, title("Interpolated f(x), n=100 "), xlab = "x",
#                            ylab = "f(x) = -x*sin(10*pi*x)")
```

Continuing on with $f_2(x) - xsin(10\pi x)$ there does not seem to be any meaningful discrepancies between our approximation of the function and the actual function.

First thing we observe in our method of computing this is that, our approximated model has 3x as many data points, which is of course due to the fact that our approximation algorithm uses 3 points to approximate value at a given $x_i$. As a result, in actuality out approximation curve is not smooth rather it is as the name suggests, a "piece-wise" jointed approximation. To see this properly, we have attached the plots of the approximation at n=3 & n=10. And also to prove the point that the more number of intervals we use, the more smoother our curve will be, we also show an example with n=1000 intervals.

```
##n=3
f1_3 <- data.frame(x1 = seq(from = 0 , to = 1 , by = 1/3))
func5 <- inter_function(fn1, n = 3)

true_f5 <- (-f1_3$x1*(1-f1_3$x1))
# plot(y=true_f5,
#      x=f1_3$x1,
#      xlab = "x",
#      ylab = "f(x)",
#      t = 'l')




f5_hat <- c()
data5 <- c()
for( i in 1:3 ) {
  x= c(func5[i,1], func5[i,2] , func5[i,3])
  y = func5[i,4] + func5[i,5]*x+ func5[i,6]*x**2
  f5_hat = c(f5_hat, y)
```

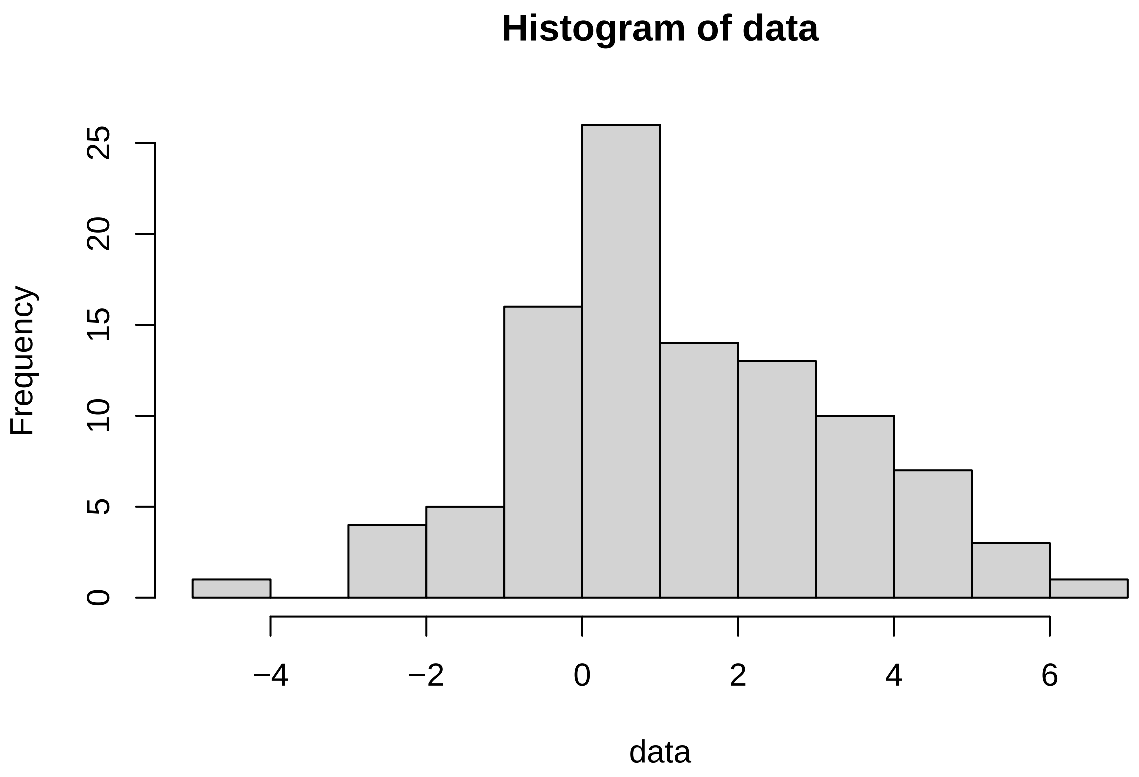
```r
  data5 = c(data5, x)
}

#plot(y=f5_hat , x=data5, title("Interpolated f(x), n=3", xlab = "x",
#                      ylab = "f(x) = -x(1-x)"), t= 'l')

##n=10
f1_2 <- data.frame(x1 = seq(from = 0 , to = 1 , by = 0.1))
func4 <- inter_function(fn1, n = 10)

true_f4 <- (-f1_2$x1*(1-f1_2$x1))
# plot(y=true_f4,
#      x=f1_2$x1,
#      xlab = "x",
#      ylab = "f(x)",
#      title("Actual f(x) = -x(1-x)"),
#      t = 'l')



f4_hat <- c()
data4 <- c()
for( i in 1:10 ) {
  x= c(func4[i,1], func4[i,2] , func4[i,3])
  y = func4[i,4] + func4[i,5]*x+ func4[i,6]*x**2
  f4_hat = c(f4_hat, y)
  data4 = c(data4, x)
}

#plot(y=f4_hat , x=data4, title("Interpolated f(x), n=10", xlab = "x",
#                                  ylab = "f(x) = -x(1-x)"), t= 'l')

##n=1000
f1_1 <- data.frame(x1 = seq(from = 0 , to = 1 , by = 0.001))

fn2 = function(x) (-x*sin(10*pi*x))
func3 <- inter_function(fn2, n = 1000)



true_f3 <- -f1_1$x1*sin(10*pi*f1_1$x1)
# plot(y=true_f3,
#      x=f1_1$x1,
#      xlab = "x",
#      ylab = "f(x)",
#      title("Actual f(x) = -x*sin(10*pi*x)"))


f3_hat <- c()
data3 <- c()
for( i in 1:1000 ) {
  x= c(func3[i,1], func3[i,2] , func3[i,3])
  y = func3[i,4] + func3[i,5]*x+ func3[i,6]*x**2
  f3_hat = c(f3_hat , y)
  data3 = c(data3 , x )
}

#plot(y=f3_hat , x=data3, title("Interpolated f(x), n=1000 "), xlab = "x",
#                              ylab = "f(x) = -x*sin(10*pi*x)", t='l' )
```

Thus higher number of intervals taken will result in a more significant approximation of the function.

## PROBLEM 2

### Log Likelihoood estimators $\mu$ & $\sigma$

The given data is plotted for observation.



As we can see that the data seems to be normally distributed with its mean lying somewhere close to 1, which confirms what is given to us in the problem statement about the data being distributed normally. Let us proceed further with solving this problem.

Given our data, we can write the likelihood as the joint density of the sample i.e.

$$L(\mu,\sigma^2) = \prod_{i=1}^{n} f(y_i|\mu,\sigma^2) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\Big(-\frac{y_i-\mu}{2\sigma^2}\Big) = \Big(\frac{1}{2\pi\sigma^2}\Big)^{n/2} \exp\Big(\frac{-1}{2\sigma^2}\sum_{i=1}^{n}(y_i-\mu)^2\Big)$$

Taking logs on both sides

$$ln[L(\mu,\sigma^2)] = -\frac{n}{2}ln\sigma^2 - \frac{n}{2}ln2\pi - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i-\mu)^2$$

For MLE of $\mu$ & $\sigma$, we take the partial derivatives w.r.t. $\mu$ & $\sigma$ and set them to zero.

First, derivating w.r.t $\mu$, the first and second term become 0, and we derivate the 3rd term leading to:

$$\frac{\partial(ln[L(\mu,\sigma^2)])}{\partial\mu} = \frac{1}{\sigma^2}\Big(\sum_{i=1}^{n}(y_i) - n\mu\Big)$$

If we equate this to 0, and isolate $\mu$, we get:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} y_i$$

This we know to be the sample mean, we can summarize the result as follows:

$$\hat{\mu}_{MLE} = \bar{y}$$

Similarly for $\sigma$, now we derivate the log likelihood function w.r.t. $\sigma$, the second term becomes zero and we get:

$$\frac{\partial(ln[L(\mu,\sigma^2)])}{\partial\sigma} = -\frac{n}{2}.\frac{2}{\sigma} - \Big(\frac{\sum_{i=1}^{n}(y_i-\mu)^2}{2}\Big).\Big(-\frac{2}{\sigma^3}\Big)$$

Simplifying:

$$\frac{\partial(ln[L(\mu,\sigma^2)])}{\partial\sigma} = -\frac{n}{\sigma} + \frac{\sum_{i=1}^{n}(y_i-\mu)^2}{\sigma^3}$$

Setting partial derivative to 0 and solving for $\sigma$, we get

$$\sigma^2 = \frac{\sum_{i=1}^{n}(y_i-\mu)^2}{n}$$

And this expression we know is the sample variance, so we can summarize as:

$$\hat{\sigma}^2_{MLE} = s^2$$

In terms of sample values, the variance can be written as follows:

$$\hat{\sigma}^2 = \frac{\sum_{i=1}^{n}(y_i - \frac{1}{n}\sum_{i=1}^{n} y_i)^2}{n}$$

Now that we have simplified the functions for calculating the estimators, we can use the built in R functions for mean and variance for our given data:

```r
mu_data = mean(data)
sigma_sq_data = var(data)
#Note that var in R computed the unbiased estimator, with denominator n-1.
#So this will not give the exact value from your formula above.

cat("mu = ", mu_data, "\n")
```

```
## mu =  1.275528
```

```r
cat("sigma = ", sqrt(sigma_sq_data), "\n")
```

```
## sigma =  2.016082
```

## Writing the loglikelihood function

We now write the function for calculating the log-likelihood which will be then used to optimize with initial parameters, $\theta$

```r
loglikelihood <- function(theta, x) {
  mu = theta[1]
  sig = theta[2]
  n = length(x)
  lglklhood = -(n/2)*(log(2*pi*sig^2)) + (-1/(2*sig^2))*sum((x-mu)^2)
  return(-lglklhood)
}
```

We are passing the negative value of the log-likelihood as the optim() minimizes that are passed into it. Hence, in order to maximize log-likelihood, we return the negative value of the log-likelihood.

## Optimizing without specifying gradient

Next we pass the initial parameters $\mu = 0$ and $\sigma = 1$ and optimize as follows:

```r
optim_norm_BFGS_withoutgradient <- optim(par = c(0,1), loglikelihood, x = data, method = "BFGS")
optim_norm_BFGS_withoutgradient
```

```
## $par
## [1] 1.275528 2.005977
##
## $value
## [1] 211.5069
##
## $counts
## function gradient
##       37       15
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```r
optim_norm_CG_withoutgradient <- optim(par = c(0,1), loglikelihood, x = data, method = "CG")
optim_norm_CG_withoutgradient
```

```
## $par
## [1] 1.275528 2.005977
##
## $value
## [1] 211.5069
##
## $counts
## function gradient
##      297       45
##
## $convergence
## [1] 0
##
## $message
## NULL
```

The results show that when using the BFGS method, the function was called 37 times and the calculated gradient was 15. In the case of the CG method, the function was called 297 times with the gradient being 45. The results are pretty close to the true values that were calculated earlier in the paper.

## Optimizing with gradient

Now we'll attempt to optimize by providing a gradient function. For writing the gradient function we will use the understanding that we are optimizing the log-likelihood function. And we will obtain the optimized value of the log-likehood function at the MLE values for our parameters i.e. $\mu$ and $\sigma$. Thus, our gradient function will be written to return the following expressions:

$$\frac{\partial(ln[L(\mu,\sigma^2)])}{\partial\mu} = \frac{1}{\sigma^2}\Big(\sum_{i=1}^{n}(y_i) - n\mu\Big)$$

and,

$$\frac{\partial(ln[L(\mu,\sigma^2)])}{\partial\sigma} = -\frac{n}{\sigma} + \frac{\sum_{i=1}^{n}(y_i-\mu)^2}{\sigma^3}$$

So to reflect this, our code is as follows:

```
gradient <- function(par, x) {
  mu = par[1]
  sigma = par[2]
  n = length(x)
  mu_est = -(sum(x) - n*mu)/sigma^2
  sigma_est = n/sigma - sum((x - mu)^2)/sigma^3
  return(c(mu_est, sigma_est))
}
```

Note that we returned the -ve values of $\mu$ and $\sigma$. This is because we are minimizing -ve log likelihood, thus we accordingly need to pass the -ve values of estimators that will minimize the function. And thus, we modify our optim function with gradient for both methods as follows:

```
optim_norm_BFGS_withgradient <- optim(par = c(0,1),
                                      gr = gradient,
                                      loglikelihood,
                                      x = data,
                                      method = "BFGS")
optim_norm_BFGS_withgradient
```

```
## $par
## [1] 1.275528 2.005977
##
## $value
## [1] 211.5069
##
## $counts
## function gradient
##       39       15
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
optim_norm_CG_withgradient <- optim(par = c(0,1),
                                    gr = gradient,
                                    loglikelihood,
                                    x = data,
                                    method = "CG")
optim_norm_CG_withgradient
```

```
## $par
## [1] 1.275528 2.005976
##
## $value
## [1] 211.5069
##
## $counts
## function gradient
##       53       17
##
## $convergence
## [1] 0
##
## $message
## NULL
```

The results are still accurate. All 4 of our attempts lead to convergence, with the value of the function also resulting in the same output for all 4 calls. The bigger difference actually lies in the number of times the function is getting called. If you compare the counts of the 4 calls, we see that there are significantly lesser number of calls to the function being optimized i.e. the loglikelihood() in the CG method, where the number of loglikelihood() calls have been reduced by ~80%! This obviously comes at the cost of accuracy. As we can see that even though the optimized value of mean and the resulting function values are equal to the true value, there is a slight difference in accuracy of the sigma values. And this delta was slightly higher in the CG method with gradient.

### Recommended configuration?

Therefore on the matter of desired settings, because this data set is much smaller, the speed of the optimization doesn't get impacted much, but with larger multivariate datasets, an increased number of function calls can create a processing bottleneck without any significant improvement in accuracy of our results. Hence the recommendation should be that if it is convenient to analytically derive the gradient of the function to be optimized, then we can save on processing speed and get results faster without much loss in precision.And even when the gradient function is not easy, to solve there are packages available that can be used to build a gradient function.

## Log-likelihood vs Likelihood?

The choice between maximizing likelihood vs log(likelihood) is largely led by the following 2 reasons:

1. Analytical convenience: Log() is a monotonically increasing function hence in order to maximize likelihood, we just have to maximize loglikehood. Also, the distributive properties of logarithms help simplify most expressions from multiplication to summation, especially as in the case of Gaussian, we dont have to calculate the exponential terms by taking logarithms. From a computational perspective, summation is not as computationally expensive as multiplication.

2. Avoiding underflow: In statistics and ML we will be working with large datasets comprising of small numbers. If we keep working with likelihood functions, the resulting product terms can very quickly lead to extremely small numbers and loss of floating point precision thus resulting in underflow. Hence, it is recommended that we use log-likelihood for our computations.

## Appendix - CODE

```r
load("data.Rdata")
#hist(data)

loglikelihood <- function(theta, x) {
  mu = theta[1]
  sig = theta[2]
  n = length(x)
  lglklhood = -(n/2)*(log(2*pi*sig^2)) + (-1/(2*sig^2))*sum((x-mu)^2)
  return(-lglklhood)
}

optim_norm <- optim(par = c(0,1), loglikelihood, x = data, method = "BFGS")
optim_norm

optim_norm1 <- optim(par = c(0,1), loglikelihood, x = data, method = "CG")
optim_norm1


mean(data)
sd(data)



gradient <- function(par, x) {
  mu = par[1]
  sigma = par[2]
  n = length(x)
  mu_est = -(sum(x) - n*mu)/sigma^2
  sigma_est = n/sigma - sum((x - mu)^2)/sigma^3
  return(c(mu_est, sigma_est))
}


optim_norm2 <- optim(par = c(0,1), gr = gradient, loglikelihood, x = data, method = "BFGS")
optim_norm2

optim_norm3 <- optim(par = c(0,1), gr = gradient, loglikelihood, x = data, method = "CG")
optim_norm3
```