Computer Lab 2

Dinesh and Umamaheswarababu Maddela

2023-01-18

Question 1: Optimizing parameters

1.1 Write a function that uses optim() and finds values of (a0, a1, a2)

```
# Quadratic function
quad_function<- function(a0,a1,a2,x){
  return(a0+a1*x+a2*x^2)
}

funct_1<- function(x){
  return(-x*(1-x))
}

#Least square error function
least_square<- function(params,point,fun){
  a0<-params[1];a1<-params[2];a2<-params[3];
  x0<-point[1];x1<-point[2];x2<-point[3];
  k<-fun(x0);l<-fun(x1);m<-fun(x2);
  return((k-quad_function(a0,a1,a2,x0))^2+(l-quad_function(a0,a1,a2,x1))^2+(m-quad_function(a0,a
1,a2,x2))^2)
}

#to find the values of a0,a1,a2 using optim()
find_parameters<-function(params,point,fun){
  result<-optim(par = params, fn=least_square,point=point,fun_values=fun_values)
  return(result)
}
```

1.2 Now construct a function that approximates a function defined on the interval [0, 1]. Your function should take as a parameter the number of equal–sized intervals that [0, 1] is to be divided into and

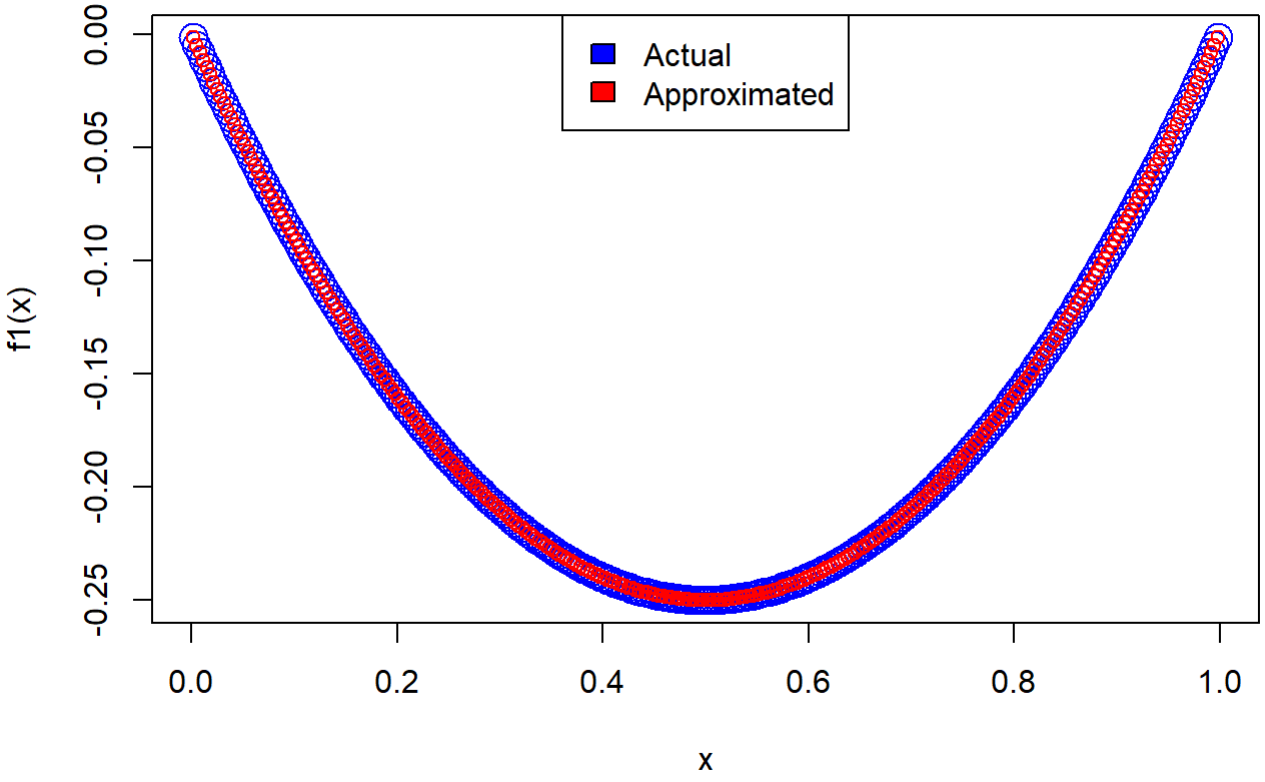
the function to approximate.

```
#Least square error function
least_square2<- function(params,point,fun_values){
  a0<-params[1];a1<-params[2];a2<-params[3];
  x0<-point[1];x1<-point[2];x2<-point[3];
  k<-fun_values[1];l<-fun_values[2];m<-fun_values[3];
  return((k-quad_function(a0,a1,a2,x0))^2+(l-quad_function(a0,a1,a2,x1))^2+(m-quad_function(a0,a
1,a2,x2))^2)
}

#to find the values of a0,a1,a2 using optim
find_parameters2<-function(params,point,fun_values){
  result<-optim(par = params, fn=least_square2,point=point,fun_values=fun_values)
  return(result$par)
}

#a function to split into equal sized intervals in [0,1]
equal_intervals<- function(n){   # n= no. of intervals required
  k=1/n
  parts<-list()
  for (i in 1:n){
    parts[i]<- list(c((i-1)*k,((i-1)*k + i*k)/2, i*k))
  }
  return(parts)
}

# a function to approximate the given function f(x)
approx_funct <- function(intervals,fun){
  params=c(0.1,0.2,0.3)
  splitted <- equal_intervals(intervals)
  par<-list()
  approximated <- NULL
  for (i in 1: intervals){
    x_values <- unlist(splitted[[i]])
    f_values <- sapply(x_values, fun)
    par[[i]]<- find_parameters2(params=params, point=x_values, fun_values=f_values)
    approximated[i]<-quad_function(a0<-par[[i]][1],a1<-par[[i]][2],a2<-par[[i]][3],x_values[2])
  }
  return(approximated)
}
```

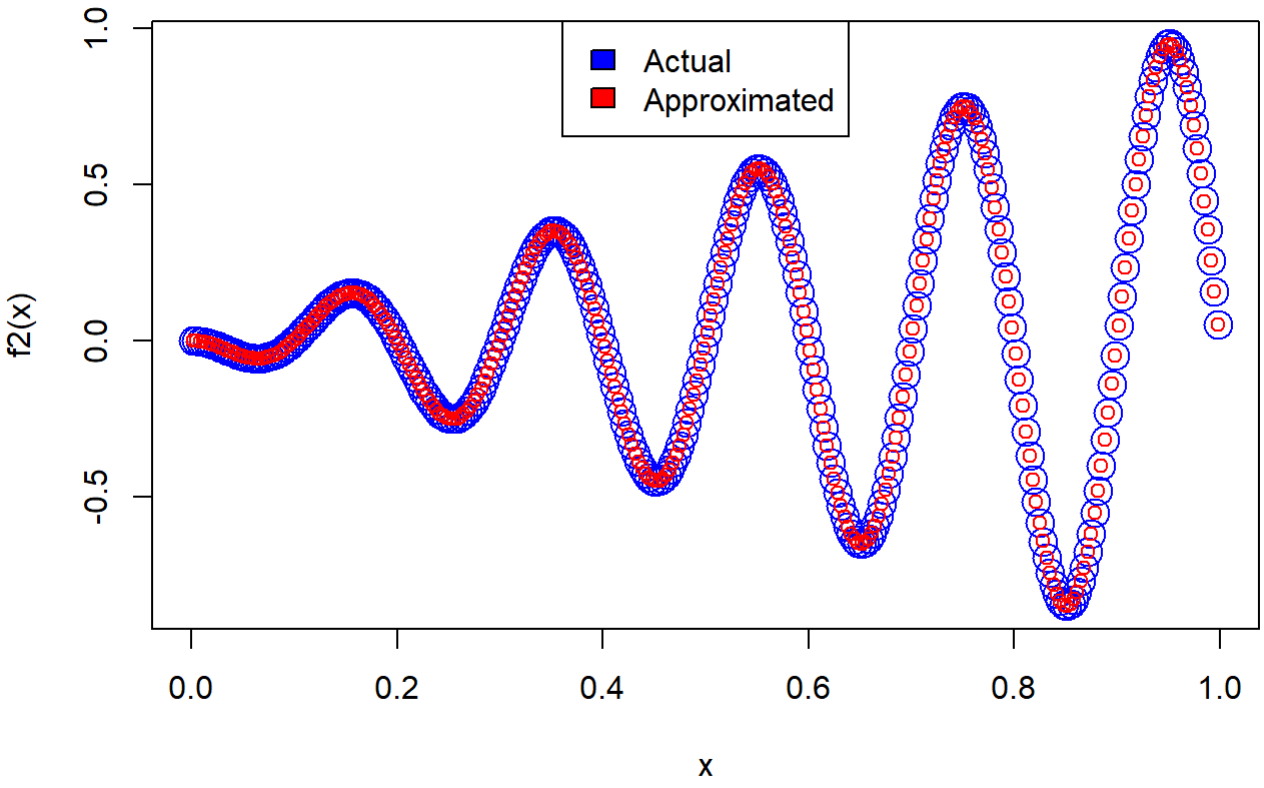
1.3 Apply your function from the previous item to f1(x) = −x(1−x) and f2(x) = −x sin(10πx).Plot f1(·), ˜f1(·) and f2(·), ˜f2(·). How did your piecewise–parabolic interpolater fare? Explain what you

observe. Take the number of subintervals to be at least 100.

```
#function f1(x)
funct_1 <- function(x) {
  return(-x * (1 - x))
}

#function f2(x)
funct_2 <- function(x) {
  return(-x * sin(10 * pi * x))
}

# a function to return the actual results for the given intervals
actual<- function(intervals,fun){
  interval<-equal_intervals(intervals)
  result<-NULL
  for (i in 1:intervals){
    result[i]<-fun(interval[[i]][2])
  }
  return(result)
}
# actual and approximated values of function f1(x)
f1_actual<-actual(intervals = 300,funct_1)
f1_approximated<-approx_funct(intervals = 300, fun = funct_1)

# actual and approximated values of function f2(x)
f2_actual<-actual(intervals = 300,funct_2)
f2_approximated<-approx_funct(intervals = 300, fun = funct_2)

inter<-equal_intervals(300)
x_axis<-NULL
for (i in 1:300){
  x_axis[i]<-inter[[i]][2]
}

#plotting f1(x) actual and approximated
plot(x_axis,f1_actual,xlab = "x", ylab = "f1(x)", col="blue",cex=2)
points(x_axis,f1_approximated, col="red",cex=1)
legend("top",c("Actual","Approximated"), fill=c("blue","red"))
```




```
#plotting f2(x) actual and approximated
plot(x_axis,f2_actual,xlab = "x", ylab = "f2(x)", col="blue",cex=2)
points(x_axis,f2_approximated, col="red",cex=1)
legend("top",c("Actual","Approximated"), fill=c("blue","red"))
```



From the above plots we can say that the piecewise–parabolic interpolater could approximate the given functions fairly. The minimization of squared error in optim() is best suited to find the parameters a0, a1 and a2. To improve the approximation we can take more number of intervals in [0,1] and use high degree polynomial functions.

Question 2: Maximizing likelihood

The file data.RData contains a sample from normal distribution with some parameters μ, σ. For this question read ?optim in detail.

2.1 Load the data to R environment.

```
load("data.RData")
data_new <- data
```

2.2 Write down the log-likelihood function for 100 observations and derive maximum likelihood estimators for μ, σ analytically by setting partial derivatives to zero. Use the derived formulae to

obtain parameter estimates for the loaded data.

The probability density function of normal distribution is:

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

When observations(n=100),

$$L(\mu, \sigma^2|y) = \frac{1}{\sqrt{(2\pi\sigma^2)^100}} * \exp^{-}(\frac{1}{2\sigma^2}) \sum_{i=1}^{100}(y_i - \mu)^2$$

Log-Likelihood expression,

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

The maximum likelihood estimator for μ is obtained by partially differentiating with respect to the parameter μ and setting to zero:

$$\frac{\partial(ln(L(\mu, \sigma^2)))}{\partial\mu} = -\frac{1}{\sigma^2}\sum_{i=1}^{n}(y_i - \mu) \mid \mu = 0$$

After solving the equation,

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

The maximum likelihood estimator for σ is obtained by partially differentiating with respect to the parameter σ and setting to zero:

$$\frac{\partial L}{\partial\sigma} = -(\frac{n}{2})(\frac{1}{\sigma^2}) + \frac{1}{2\sigma^4}\sum_{i=1}^{n}(y_i - \mu)^2 \mid \sigma = 0$$

After solving the equation,

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

```
Likelihood<-function(y){
  n<-length(y)
  mu<- (1/n) * sum(y)
  sigma<-sqrt( (1/n)* sum((y-mu)^2) )
  return(data.frame(mean=mu,sigma=sigma))
}
Likelihood(data_new)
```

mean <dbl>	sigma <dbl>
1.275528	2.005976

1 row

2.3 Optimize the minus log-likelihood function with initial parameters μ = 0, σ = 1. Try both Conjugate Gradient method (described in the presentation handout) and BFGS (discussed in the lecture) algorithm with gradient specified and without. Why it is a bad idea to maximize likelihood rather than maximizing log-likelihood?

We optimize the minus log-likelihood function using the above log-likelihood function, which is shown to be:

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

```
Log_likelihood<-function(parvec,y){
  mu<-parvec[1]
  sigma<-parvec[2]
  n<-length(y)
  ll<- - (n/2)*log(2*pi) - (n/2)*log(sigma^2) - (1/(2*sigma^2))*sum((y-mu)^2)
  return(- ll)
}
```

We perform multi-dimensional optimization here because we have both mu and $sigma$. This is accomplished by employing the Gradient method. The gradient forms of mu and $sigma$ are,

Gradient mean(μ)

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

Gradient sigma(σ)

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

```
# Gradient function
gradient <- function(parvec,y){
  n <- length(y)
  mu <- parvec[1]
  sigma <- parvec[2]
  grad_mu <- - (1/sigma^2)* sum(y-mu)
  grad_sigma <- (n/sigma) - (1/sigma^3) * sum((y-mu)^2)
  return(c(grad_mu, grad_sigma))
}

# Optimize with gradient
CG_with_grad <- optim(c(0,1), fn = Log_likelihood, gr=gradient, y=data_new, method = "CG")
BFGS_with_grad <- optim(c(0,1), fn = Log_likelihood, gr=gradient, y=data_new, method = "BFGS")
normal_with_grad <- optim(c(0,1), fn = Log_likelihood, gr=gradient, y=data_new)

# Optimize without gradient
CG_without_grad <- optim(par=c(0,1),fn=Log_likelihood,y=data_new,method="CG")
BFGS_without_grad <- optim(par=c(0,1),fn=Log_likelihood,y=data_new,method="BFGS")
normal_without_grad <- optim(par=c(0,1),fn=Log_likelihood,y=data_new)
```

Using the log-likelihood function instead of the likelihood function makes it easier to find the maximum likelihood estimate (MLE) because the exponent in the likelihood function can make it difficult to take the derivative and find the MLE. The log-likelihood function eliminates the exponent, making it simpler to find the MLE. Therefore, it is more efficient to maximize the log-likelihood rather than the likelihood.

Maximizing the log-likelihood function is more efficient than maximizing the likelihood function. The logarithm is a monotonically increasing function, so maximizing the log of a function is equivalent to maximizing the function itself. Additionally, the logarithm simplifies the mathematical analysis and improves numerical stability. When we take the log of a function, it makes the calculations of partial derivatives easier, and also prevents underflow of the numerical precision of the computer, which happens when we work with the product of a large number of small probabilities.

Logp(x) is typically optimized more effectively than p(x) using gradient methods because the gradient of logp(x) is generally more well-scaled. That is, it has a size that consistently and helpfully reflects the objective function's geometry,making it simpler to determine the right step size and reach the optimal solution in fewer steps.

2.4 Did the algorithms converge in all cases? What were the optimal values of parameters and how many function and gradient evaluations were required for algorithms to converge? Which settings would you recommend?

Yes, the algorithms converge in all cases. It is clear that the algorithms converged in all cases. This was noticed because their respective convergence values were zero.

The optimal values of parameter are :

$$\mu = 1.275528$$

and

$$\sigma = 2.005976$$

Tables compare the results, function and gradient evaluations required for algorithms to converge:

ACTUAL VALUES

	values
mean	1.275528
sd	2.016082

With Gradient

	mean	sd	function	gradient
CG	1.275528	2.005977	53	17
BFGS	1.275527	2.005977	38	15
Default	1.275563	2.006113	65	NA

Without Gradient

	mean	sd	function	gradient
CG	1.275528	2.005977	180	33
BFGS	1.275527	2.005977	37	15
Default	1.275563	2.006113	65	NA

The BFGS method without gradient has the lowest number of function and gradient evaluations. It is commonly faster and more robust than CG, and is suitable for large scale problems. It reaches the optimal solution in fewer steps compared to CG, and is less likely to get stuck and need slight adjustments to achieve substantial descent in each iteration.

The CG method, which utilizes gradient, use fewer number of functions and gradient evaluations than the method without gradient. The distinction between using (with) gradient and not using it(without) in BFGS method are not so obvious. So, using gradient in the method is better. Therefore, I will recommend the BFGS method with specified gradient.