Laboration report in Computational Statistics

Laboration 2

732A90

Duc Tran
William Wiik

Division of Statistics and Machine Learning
Department of Computer Science
Linköping University

14 november 2023

Contents

1 Question 1: Optimisation of a two-dimensional function

Consider the function

$$g(x, y) = -x^2 - x^2y^2 - 2xy + 2x + 2$$

It is desired to determine the point (x, y), x, $y \in$ [-3, 3], where the function is maximized.

1.1 1 a)

Question

Derive the gradient and the Hessian matrix in dependence of x, y. Produce a contour plot of the function g

Solution

$$\frac{\partial g(x,y)}{\partial x} = -2x - 2xy^2 - 2y + 2$$
$$\frac{\partial g(x,y)}{\partial y} = -2yx^2 - 2x$$

$$\frac{\partial^2 g(x,y)}{\partial x^2} = -2 - 2y^2$$
$$\frac{\partial^2 g(x,y)}{\partial x \partial y} = -4xy - 2$$
$$\frac{\partial^2 g(x,y)}{\partial y^2} = -2x^2$$
$$\frac{\partial^2 g(x,y)}{\partial y \partial x} = -4xy - 2$$

$$H(x,y) = \begin{bmatrix} -2 - 2y^2 & -4xy - 2 \\ -4xy - 2 & -2x^2 \end{bmatrix} \tag{1}$$

```
gradient <- function(x, y) {

  x_derivate <- -2*x - 2*x*y^2 - 2*y + 2

  y_derivate <-  - 2*y*x^2 - 2*x

  matrix <- matrix(c(x_derivate,y_derivate),2,1)
  return(c(x_derivate,y_derivate))
}



hessian_mat <- function(x, y) {

  xx_derivate <- -2 -2*y^2
  xy_derivate <-  -4*x*y -2

  yy_derivate <- -2*x^2
  yx_derivate <- -4*y*x - 2

  matrix <- matrix(c(xx_derivate,xy_derivate,
                     yx_derivate,yy_derivate),2,2)
  return(matrix)
}



x1grid <- seq(-3,3,0.01)
x2grid <- seq(-3,3,0.01)

g <- function(x, y){
  -x^2 - x^2*y^2 - 2*x*y + 2*x + 2
}


z <- outer(x1grid,x2grid,g)
contour(x1grid,x2grid,z)
```

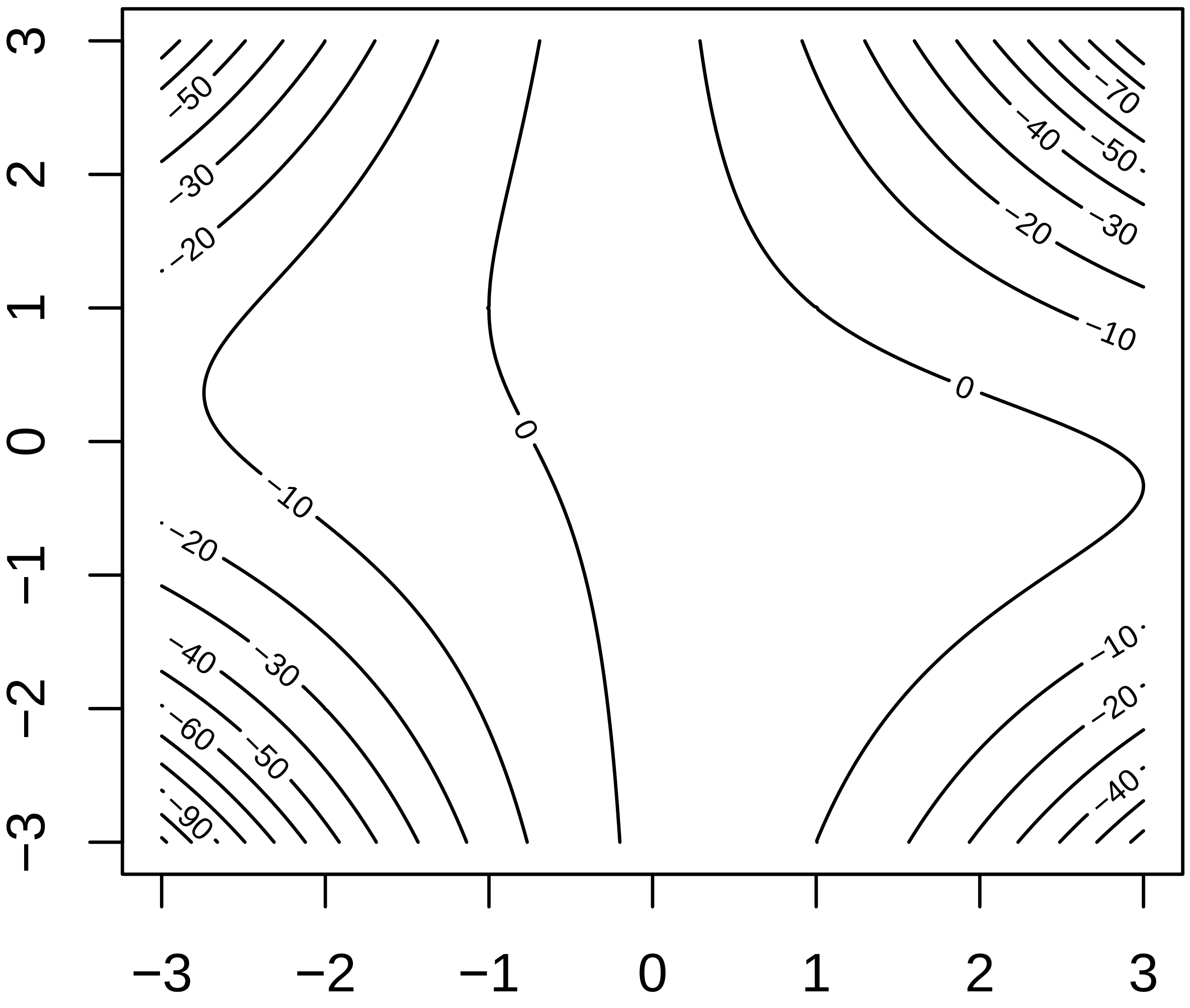

Figure 1: Contour plot of the function g

1.2 1 b)

Question

Write an own algorithm based on the Newton method in order to find a local maximum of g

Solution

Newton method:

$$x^{(t+1)} = x^{(t)} - (g''(x^{(t)}))^{-1} \cdot g'(x^{(t)})$$

```r
newton <- function(x, y, eps = 0.005){
  current_point <- c(x,y)
  converged <- FALSE
  iter <- 1
  while(!converged){
    x <- current_point[1]
    y <- current_point[2]
    next_point <- current_point - solve(hessian_mat(x,y)) %*% gradient(x,y)

    if(dist(next_point - current_point) / dist(next_point) < eps){
      return(next_point)
    } else {
      current_point <- next_point
    }

    iter <- iter + 1
    # Max number of iterations is set to 1000 to prevent endless loops.
    if(iter > 1000){
      stop("The function did not converge after 1000 iterations.")
    }
  }
}
```

1.3 1 c)

Question

Use different starting values: use the three points (x, y) = (2, 0), (-1, -2), (0, 1) and a fourth point of your choice. Describe what happens when you run your algorithm for each of those starting values. If your algorithm converges to points (x, y), compute the gradient and the Hessian matrix at these points and decide about local maximum, minimum, saddle point, or neither of it. Did you find a global maximum for $x, y \in$ [-3, 3]?

Solution

```r
library(stringr)
eigen_vec <- c()
gradient_vec <- c()

p1 <- newton(2,0)
p2 <- newton(-1,-2)
```

```r
p3 <- newton(0,1)
p4 <- newton(0,0)

eigen_vec[1] <- str_c(round(eigen(hessian_mat(p1[1], p1[2]))$values,2), collapse = ", ")
eigen_vec[2] <- str_c(round(eigen(hessian_mat(p2[1], p2[2]))$values,2), collapse = ", ")
eigen_vec[3] <- str_c(round(eigen(hessian_mat(p3[1], p3[2]))$values,2), collapse = ", ")
eigen_vec[4] <- str_c(round(eigen(hessian_mat(p4[1], p4[2]))$values,2), collapse = ", ")

gradient_vec[1] <- str_c(round(gradient(p1[1], p1[2]),2), collapse = ", ")
gradient_vec[2] <- str_c(round(gradient(p2[1], p2[2]),2), collapse = ", ")
gradient_vec[3] <- str_c(round(gradient(p3[1], p3[2]),2), collapse = ", ")
gradient_vec[4] <- str_c(round(gradient(p4[1], p4[2]),2), collapse = ", ")
```

Table 1: Information about the four poins

Points	Converges.to.point	Eigen.values	Gradient	Info
(2,0)	(1,-1)	-0.76, -5.24	0, 0	Local maximum
(-1,-2)	(0,1)	0.83, -4.83	0, 0	Saddle point
(0,1)	(0,1)	0.83, -4.83	0, 0	Saddle point
(0,0)	(0,1)	0.83, -4.83	0, 0	Saddle point

The eigen values at the converged points (1,-1) are both negative which means that it is negative definite which means it is a local maximum.

The eigen values at the converged points (0,1) are negative and positive which means that it is indefinite which means it is a saddle point.

To find the global maximum we need to find all the extreme points and for that we need to set $\frac{\partial g(x,y)}{\partial x} = 0$ and $\frac{\partial g(x,y)}{\partial y} = 0$.

This leads to a equation system.

$$\begin{cases} -2x - 2xy^2 - 2y + 2 = 0 & (1) \\ -2yx^2 - 2x = 0 & (2) \end{cases} \tag{2}$$

We can rewrite equation (2) to

$$y = -\frac{1}{x}$$

Now we put $y = -\frac{1}{x}$ in equation (1) which leads to

$$\begin{cases} -2x - 2x \cdot \frac{1}{x^2} - 2 \cdot -\frac{1}{x} + 2 = 0 \Rrightarrow -2x + 2 = 0 & (1) \\ y = -\frac{1}{x}, x \neq 0 & (2) \end{cases} \tag{3}$$

From equation (1) we can solve x

$$-2x + 2 = 0 \Rrightarrow x = 1$$

Now we can solve y from equation (2)

$$y = \frac{-1}{1} \Rrightarrow y = -1$$

The first extreme point is $(1, -1)$

Now we need to check when $x = 0$,

$$\begin{cases} -2 \cdot 0 - 2 \cdot 0 \cdot y^2 - 2y + 2 = 0 \Rrightarrow -2y + 2 = 0 \Rrightarrow y = 1 & (1) \\ -2 \cdot y \cdot 0^2 - 2 \cdot 0 = 0 \Rrightarrow 0 = 0 & (2) \end{cases} \tag{4}$$

The second extreme point is $(0, 1)$

Thus, we have two extreme point and need to check which point that is the extreme point. To do that we put the points in the equation below.

$$g(x, y) = -x^2 - x^2y^2 - 2xy + 2x + 2$$
$$g(1, -1) = -1^2 - 1^2 \cdot (-1)^2 - 2 \cdot 1 \cdot -1 + 2 \cdot 1 + 2 \Rrightarrow g(1, -1) = 4$$
$$g(0, 1) = -0^2 - 0^2 \cdot 1^2 - 2 \cdot 0 \cdot 1 + 2 \cdot 0 + 2 \Rrightarrow g(0, 1) = 2$$

The point $(1, -1)$ has the highest value and is therefor the global maximum.

1.4 1 d)

Question

What would be the advantages and disadvantages when you would run a steepest ascent algorithm instead of the Newton algorithm?

Solution

The disadvantage for steepest ascent algorithm is that it is slower than Newton algorithm because it has order $= 1$ instead of order $= 2$ that the Newton algorithm has. Steepest ascent does not use any information about curvature.

The advantage for using steepest ascent algorithm instead of the Newton algorithm is that you don't need to calculate Hessian $g''(x(t))$ in each iteration, which leads to less number of calculations to move one step.

2 Question 2:

The data in this task consists of 10 observations and is presented in table 2.

```
x <- c(0,0,0,0.1,0.1,0.3,0.3,0.9,0.9,0.9)
y <- c(0,0,1,0,1,1,1,0,1,1)

data_table <- data.frame(x=x, y=y)
kable(data_table,
      caption = "Data for Question 2")
```

Table 2: Data for Question 2

x	y
0.0	0
0.0	0
0.0	1
0.1	0
0.1	1
0.3	1
0.3	1
0.9	0
0.9	1
0.9	1

In table 2, variable x is the dose of drug for each subject and variable y indicates if an event occured (y=1).

A logistic regression

$$p(x) = P(Y = 1|x) = \frac{1}{1 + exp(-\beta_0 - \beta_1 x)}$$

is fit to the data where β_0 and β_1 is estimated with the log likelihood function

$$g(\boldsymbol{b}) = \sum_{i=1}^{n} [y_i \cdot log\{(1 + exp(-\beta_0 - \beta_1 x_i))^{-1}\} + (1 - y_i) \cdot log\{1 - (1 + exp(-\beta_0 - \beta_1 x_i))^{-1}\}]$$

where $\boldsymbol{b} = (\beta_0, \beta_1)^T$.

The gradient for the log likelihood function is

$$\boldsymbol{g'}(\boldsymbol{b}) = \sum_{i=1}^{n} \left\{ y_i - \frac{1}{1 + exp(-\beta_0 - \beta_1 x_i)} \right\} \begin{pmatrix} 1 \\ x_i \end{pmatrix}$$

2.1 2a

Question: Write a function for an ML-estimator for (β_0, β_1) using the steepest ascent method with a step-size reducing line search (back-tracking). For this, you can use and modify the code for the steepest ascent example from the lecture. The function should count the number of function and gradient evaluations.

Answer: The code used to solve this task is as follows.

```r
# Value of Log-likelihood function at point b0,b1
g <- function(point){
  b0 <- point[1]
  b1 <- point[2]
  value <- 0
  for(i in 1:length(x)){
    value <- value +
      sum(y[i] * log( solve(1+exp(-b0-b1*x[i])) ) + (1-y[i]) * log(1 -solve(1+exp(-b0-b1*x[i]))) )
  }
  return(value)
}

# Gradient of the log-likelihood function at point b0,b1
gradient <- function(point){
  b0 <- point[1]
  b1 <- point[2]
  left_exp <- (y - (1/(1+exp(-b0-b1*x))))
  b0 <- sum(left_exp*1)
  b1 <- sum(left_exp*x)
  estimate <- c(b0,b1)
  return(estimate)
}


steepest_asc <- function(starting_point, eps=1e-8, alpha0=1){
  current_point <- starting_point
  converged <- FALSE
  iter <- 1
  g_iter <- 0
  gradient_iter <- 0

  while(!converged){
    alpha <- alpha0
    next_point <- current_point + alpha*gradient(current_point)
    gradient_iter <- gradient_iter + 1

    # Checks if next point is not an improvement and modifies alpha
    while(g(next_point) < g(current_point)){
      # We evaluate g twice to be inside the while loop.
      g_iter <- g_iter + 2
      alpha <- alpha/2
      next_point <- current_point + alpha*gradient(current_point)
      gradient_iter <- gradient_iter + 1
      if(alpha < 0.005){
        result <- list("Stopping point" = current_point,
                       "Function calls" = g_iter,
                       "Gradient evaluations" = gradient_iter)
        return(result)
      }
    }

    # We evaluate g twice to exit the while loop.
    g_iter <- g_iter + 2

    # Stopping condition
    if(all(abs(next_point - current_point) < 0.00001)){
      result <- list("Stopping point" = current_point,
                     "Function calls" = g_iter,
                     "Gradient evaluations" = gradient_iter)
      return(result)
    } else {
      # New point accepted
      iter <- iter+1
      current_point <- next_point
    }

    # Max number of iterations is set to 1000 to prevent endless loops.
    if(iter > 1000){
      stop("The function did not converge after 1000 iterations.")
    }
  }
  return(current_point, g_iter, gradient_iter)
}
```

2.2 2b

Question: Compute the ML-estimator with the function from a. for the data (x_i, y_i) above. Use a stopping criterion such that you can trust five digits of both parameter estimates for β_0 and β_1. Use the starting value (β_0, β_1) = (-0.2, 1). The exact way to use backtracking can be varied. Try two variants and compare number of function and gradient evaluation done until convergence.

Answer: Computing the ML-estimator with our function and data with the starting values (β_0, β_1) = (-0.2, 1) in our function gives us the result as follows

```
starting_point <- c(-0.2,1)
steepest_asc(starting_point)
```

```
## $`Stopping point`
## [1] -0.009343497  1.262798358
##
## $`Function calls`
## [1] 108
##
## $`Gradient evaluations`
## [1] 54
```

The function called the likelihood function 108 times, gradient function 54 times, and converged to the values: $\beta_0 \approx -0.00934$ and $\beta_1 \approx 1.2628$

Another starting value with (β_0, β_1) = (-1, 2) in our function gives us the result as follows

```r
starting_point <- c(-1,2)
steepest_asc(starting_point)
```

```
## $`Stopping point`
## [1] -0.009364355  1.262861673
##
## $`Function calls`
## [1] 122
##
## $`Gradient evaluations`
## [1] 61
```

The function called the likelihood function 122 times, gradient function 61 times, and converged to the values: $\beta_0 \approx -0.00936$ and $\beta_1 \approx 1.2629$. The starting values (-1, 2) is further away than (-0.2, 1) to the converged values. By starting further away from the converged values, the amount of iterations needed to converge increases.

Another variant is to change the step-length α, this is done by changing the standard value from 1 to 4.

```r
starting_point <- c(-0.2,1)
steepest_asc(starting_point, alpha0 = 4)
```

```
## $`Stopping point`
## [1] -0.009356339  1.262802774
##
## $`Function calls`
## [1] 222
##
## $`Gradient evaluations`
## [1] 111
```

From the output we converged to the same point, but the function used 222 function calls and 111 gradient calls. This can be interpreted that the step-length is too large since we start close to the convergent point so that the next point with $\alpha = 4$ is not an improvement.

2.3 2c

Question: Use now the function optim with both the BFGS and the Nelder-Mead algorithm. Do you obtain the same results compared with b.? Is there any difference in the precision of the result? Compare the number of function and gradient evaluations which are given in the standard output of optim.

Answer: The function optim minimizes the function that we put in. Therefore we have changed the function g to -g and the sign of the gradient was also changed. This is done with the code as follows

```r
# -g
h <- function(point){
  b0 <- point[1]
  b1 <- point[2]
  value <- 0
  for(i in 1:length(x)){
    value <- value -
      sum(y[i] * log( solve(1+exp(-b0-b1*x[i])) ) + (1-y[i]) * log(1 - solve(1+exp(-b0-b1*x[i]))) )
  }
  return(value)
}

# gradient for -g
gradient2 <- function(point){
  b0 <- point[1]
  b1 <- point[2]
  left_exp <- -(y - (1/(1+exp(-b0-b1*x))))
  b0 <- sum(left_exp*1)
  b1 <- sum(left_exp*x)
  estimate <- c(b0,b1)
  return(estimate)
}
```

The results for BFGS with the optim function is as follows

```
starting_point <- c(-0.2, 1)
optim(starting_point, fn=h, gr=gradient2, method="BFGS")
```

```
## $par
## [1] -0.009356126  1.262812832
##
## $value
## [1] 6.484279
##
## $counts
## function gradient
##       12        8
##
## $convergence
## [1] 0
##
## $message
## NULL
```

The results for Nelder-Mead with the optim function is as follows

```
optim(starting_point, fn=h, method="Nelder-Mead") # Does not use the gradient
```

```
## $par
## [1] -0.009423433  1.262738266
##
## $value
## [1] 6.484279
##
## $counts
## function gradient
##       47       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

In table 3, the results for BFGS, Nelder-Mead, and our function are summarized.

Table 3: Results from our function, BFGS and Nelder-Mead.

	Beta 0	Beta 1	Function calls	Gradient calls
Our function	-0.00934	1.2628	108	54
BFGS	-0.00936	1.2628	12	8
Nelder-Mead	-0.00942	1.2627	47	NA

From table 3, we get similar estimations for β_0 and β_1 between BFGS, Nelder-Mead, and our function. The precision differ the most with the function Nelder-Mead compared to BFGS and our function. Our implemented function used the most functions and gradient calls out of all methods. BFGS method used the least amount of function calls. The Nelder-Mead method did not use the gradient and this could explain why the precision of the algorithm differed from the other two methods.

2.4 2d

Question: Use the function glm in R to obtain an ML-solution and compare it with your results before.

Answer:

```
glm(y~x, family= binomial(link = "logit"))$coefficients
```

```
##  (Intercept)            x
## -0.009359853  1.262823430
```

The estimates from the glm function are $\beta_0 \approx -0.00936$ and $\beta_1 \approx 1.2628$. These estimates are similar to the estimates in table Y, the estimates from glm function are closest to the BFGS method.

3 Statement of Contribution

We both worked on the tasks individually and helped each other when needed. We later compared and discussed our solutions before dividing the task of writing the laboration report.

3.1 Question 1

Text written by William.

3.2 Question 2

Text written by Duc.

4 Appendix

The code used in this laboration report are summarised in the code as follows:

```r
library(knitr)
knitr::opts_chunk$set(
  echo = TRUE,
  fig.width = 4.5,
  fig.height = 4.5)


gradient <- function(x, y) {

  x_derivate <- -2*x - 2*x*y^2 - 2*y + 2

  y_derivate <-  - 2*y*x^2 - 2*x

  matrix <- matrix(c(x_derivate,y_derivate),2,1)
  return(c(x_derivate,y_derivate))
}



hessian_mat <- function(x, y) {

  xx_derivate <- -2 -2*y^2
  xy_derivate <-  -4*x*y -2

  yy_derivate <- -2*x^2
  yx_derivate <- -4*y*x - 2

  matrix <- matrix(c(xx_derivate,xy_derivate,
                     yx_derivate,yy_derivate),2,2)
  return(matrix)
}
```

```r
x1grid <- seq(-3,3,0.01)
x2grid <- seq(-3,3,0.01)

g <- function(x, y){
  -x^2 - x^2*y^2 - 2*x*y + 2*x + 2
}


z <- outer(x1grid,x2grid,g)
contour(x1grid,x2grid,z)


newton <- function(x, y, eps = 0.005){
  current_point <- c(x,y)
  converged <- FALSE
  iter <- 1
  while(!converged){
    x <- current_point[1]
    y <- current_point[2]
    next_point <- current_point - solve(hessian_mat(x,y)) %*% gradient(x,y)

    if(dist(next_point - current_point) / dist(next_point) < eps){
      return(next_point)
    } else {
      current_point <- next_point
    }

    iter <- iter + 1
    # Max number of iterations is set to 1000 to prevent endless loops.
    if(iter > 1000){
      stop("The function did not converge after 1000 iterations.")
    }
  }
}


library(stringr)
eigen_vec <- c()
gradient_vec <- c()

p1 <- newton(2,0)
p2 <- newton(-1,-2)
p3 <- newton(0,1)
p4 <- newton(0,0)

eigen_vec[1] <- str_c(round(eigen(hessian_mat(p1[1], p1[2]))$values,2), collapse = ", ")
eigen_vec[2] <- str_c(round(eigen(hessian_mat(p2[1], p2[2]))$values,2), collapse = ", ")
eigen_vec[3] <- str_c(round(eigen(hessian_mat(p3[1], p3[2]))$values,2), collapse = ", ")
```

```
eigen_vec[4] <- str_c(round(eigen(hessian_mat(p4[1], p4[2]))$values,2), collapse = ", ")

gradient_vec[1] <- str_c(round(gradient(p1[1], p1[2]),2), collapse = ", ")
gradient_vec[2] <- str_c(round(gradient(p2[1], p2[2]),2), collapse = ", ")
gradient_vec[3] <- str_c(round(gradient(p3[1], p3[2]),2), collapse = ", ")
gradient_vec[4] <- str_c(round(gradient(p4[1], p4[2]),2), collapse = ", ")





converges <- c()
converges[1] <- paste0("(", str_c(round(p1,2),collapse = ","),")")
converges[2] <- paste0("(", str_c(round(p2,2),collapse = ","),")")
converges[3] <- paste0("(", str_c(round(p3,2),collapse = ","),")")
converges[4] <- paste0("(", str_c(round(p4,2),collapse = ","),")")

df_1c <- data.frame("Points" = c("(2,0)","(-1,-2)","(0,1)", "(0,0)"),
                    "Converges to point" = converges,
                    "Eigen values" = eigen_vec,
                    "Gradient" = gradient_vec,
                    "Info" = c("Local maximum", "Saddle point", "Saddle point", "Saddle point"))

knitr::kable(df_1c, digits = 2, caption = "Information about the four poins")


x <- c(0,0,0,0.1,0.1,0.3,0.3,0.9,0.9,0.9)
y <- c(0,0,1,0,1,1,1,0,1,1)

data_table <- data.frame(x=x, y=y)
kable(data_table,
      caption = "Data for Question 2")
# Value of Log-likelihood function at point b0,b1
g <- function(point){
  b0 <- point[1]
  b1 <- point[2]
  value <- 0
  for(i in 1:length(x)){
    value <- value +
      sum(y[i] * log( solve(1+exp(-b0-b1*x[i])) ) + (1-y[i]) * log(1 -solve(1+exp(-b0-b1*x[i]))) )
  }
  return(value)
}

# Gradient of the log-likelihood function at point b0,b1
gradient <- function(point){
  b0 <- point[1]
  b1 <- point[2]
  left_exp <- (y - (1/(1+exp(-b0-b1*x))))
  b0 <- sum(left_exp*1)
  b1 <- sum(left_exp*x)
  estimate <- c(b0,b1)
  return(estimate)
}


steepest_asc <- function(starting_point, eps=1e-8, alpha0=1){
  current_point <- starting_point
  converged <- FALSE
  iter <- 1
  g_iter <- 0
  gradient_iter <- 0

  while(!converged){
    alpha <- alpha0
    next_point <- current_point + alpha*gradient(current_point)
    gradient_iter <- gradient_iter + 1

    # Checks if next point is not an improvement and modifies alpha
    while(g(next_point) < g(current_point)){
      # We evaluate g twice to be inside the while loop.
      g_iter <- g_iter + 2
      alpha <- alpha/2
      next_point <- current_point + alpha*gradient(current_point)
      gradient_iter <- gradient_iter + 1
      if(alpha < 0.005){
        result <- list("Stopping point" = current_point,
                       "Function calls" = g_iter,
                       "Gradient evaluations" = gradient_iter)
        return(result)
      }
    }

    # We evaluate g twice to exit the while loop.
    g_iter <- g_iter + 2

    # Stopping condition
    if(all(abs(next_point - current_point) < 0.00001)){
      result <- list("Stopping point" = current_point,
                     "Function calls" = g_iter,
                     "Gradient evaluations" = gradient_iter)
      return(result)
    } else {
      # New point accepted
      iter <- iter+1
      current_point <- next_point
    }

    # Max number of iterations is set to 1000 to prevent endless loops.
    if(iter > 1000){
      stop("The function did not converge after 1000 iterations.")
    }
  }
  return(current_point, g_iter, gradient_iter)
}
starting_point <- c(-0.2,1)
steepest_asc(starting_point)
starting_point <- c(-1,2)
steepest_asc(starting_point)
starting_point <- c(-0.2,1)
steepest_asc(starting_point, alpha0 = 4)

# -g
h <- function(point){
  b0 <- point[1]
  b1 <- point[2]
  value <- 0
  for(i in 1:length(x)){
    value <- value -
      sum(y[i] * log( solve(1+exp(-b0-b1*x[i])) ) + (1-y[i]) * log(1 - solve(1+exp(-b0-b1*x[i]))) )
  }
  return(value)
}

# gradient for -g
gradient2 <- function(point){
  b0 <- point[1]
  b1 <- point[2]
  left_exp <- -(y - (1/(1+exp(-b0-b1*x))))
  b0 <- sum(left_exp*1)
  b1 <- sum(left_exp*x)
  estimate <- c(b0,b1)
  return(estimate)
}
starting_point <- c(-0.2, 1)
optim(starting_point, fn=h, gr=gradient2, method="BFGS")
optim(starting_point, fn=h, method="Nelder-Mead") # Does not use the gradient


beta0 <- c(-0.00934, -0.00936, -0.00942)
beta1 <- c(1.2628, 1.2628, 1.2627)
fun_calls <- c(108, 12, 47)
gradient_calls <- c(54, 8, NA)
table_data <- data.frame(beta0, beta1, fun_calls, gradient_calls)
rownames(table_data) <- c("Our function", "BFGS", "Nelder-Mead")
colnames(table_data) <- c("Beta 0", "Beta 1", "Function calls", "Gradient calls")
kable(table_data, caption = "Results from our function, BFGS and Nelder-Mead.")
glm(y~x, family= binomial(link = "logit"))$coefficients
```