Created by: Marco Hafid - 22-620-546 Matus Kubla - 23-604-382

Importing the libraries

```
# Importing the required packages
library(numDeriv)
library(glmnet)
```

```
## Warning: package 'glmnet' was built under R version 4.2.3
```

```
## Loading required package: Matrix
```

```
## Warning: package 'Matrix' was built under R version 4.2.3
```

```
## Loaded glmnet 4.1-7
```

```
library(nlme)
library(Matrix)
```

```
# PROBLEM A

# Defining the Gradient Descent Function
gradient_descent <- function(f, start_value, max_iter, gamma) {
  current_value <- start_value
  history <- matrix(nrow = max_iter + 1, ncol = length(start_value))

  history[1,] <- start_value
  for (i in 1:max_iter) {
    grad <- grad(f, current_value)
    current_value <- current_value - gamma * grad
    history[i + 1,] <- current_value
  }

  list(minimizer = current_value, history = history)
}
```

Functions for plots

```
# Function that computes and plots the function values over the iterations
plot_optimization <- function(history, f) {
  function_values <- apply(history, 1, function(v) f(v))

  plot(function_values, type = 'l', main = "Function Optimization",
       xlab = "Iteration", ylab = "Function Value", col = "blue")
}

# Function for visualizing the iteration steps and optimization path
plot_contour_with_path <- function(f, range_x, range_y, history) {
  x_seq <- seq(range_x[1], range_x[2], length.out = 200)
  y_seq <- seq(range_y[1], range_y[2], length.out = 200)
  z <- outer(x_seq, y_seq, Vectorize(function(x, y) f(c(x, y))))
```

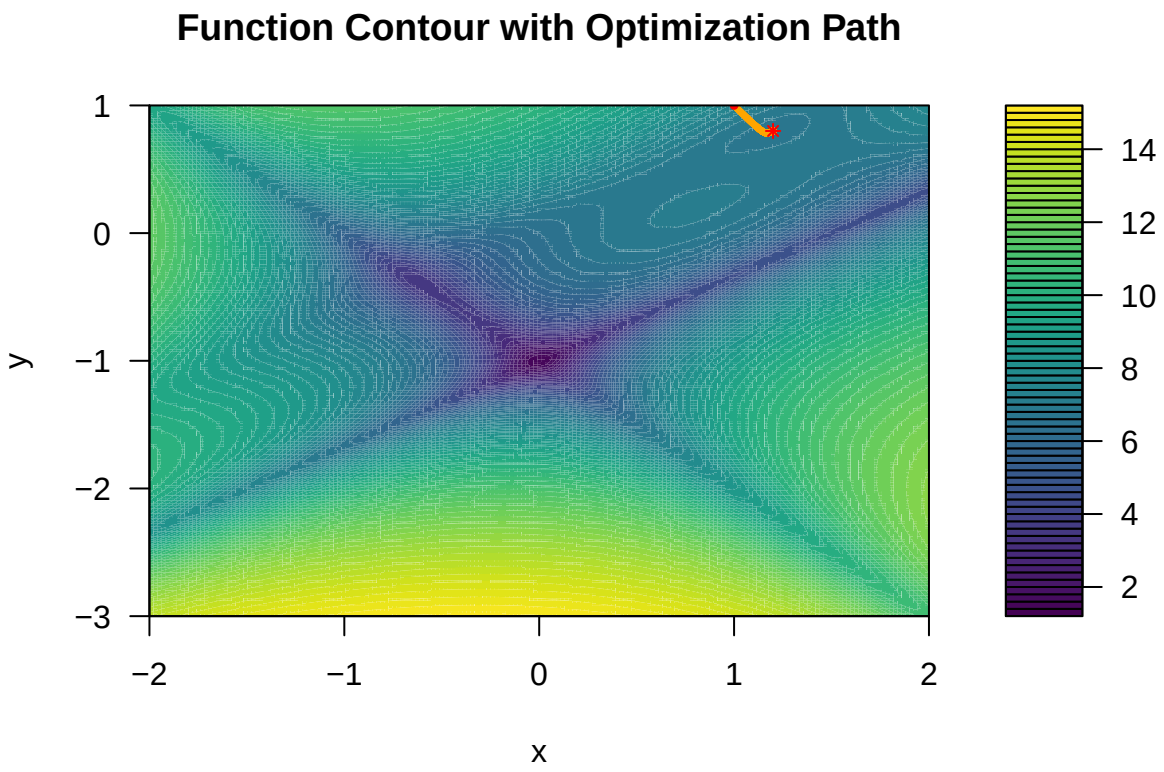
```
  filled.contour(x_seq, y_seq, log(z + 1), nlevels = 50,
                 color.palette = viridis::viridis,
                 xlab = "x", ylab = "y", main = "Function Contour with Optimization Path",
                 plot.axes = {
                   axis(1); axis(2)
                   points(history[,1], history[,2],
                          col = "orange", pch = 20, cex = 0.5)
                   lines(history[,1], history[,2],
                         col = "orange", type = "l")
                   points(history[1,1], history[1,2],
                          col = "red", pch = 20, cex = 0.7)
                   points(history[nrow(history),1], history[nrow(history),2], col = "red",
                          pch = 8, cex = 0.7)
                 })
}
```

Example 1: Goldstein-Price Function

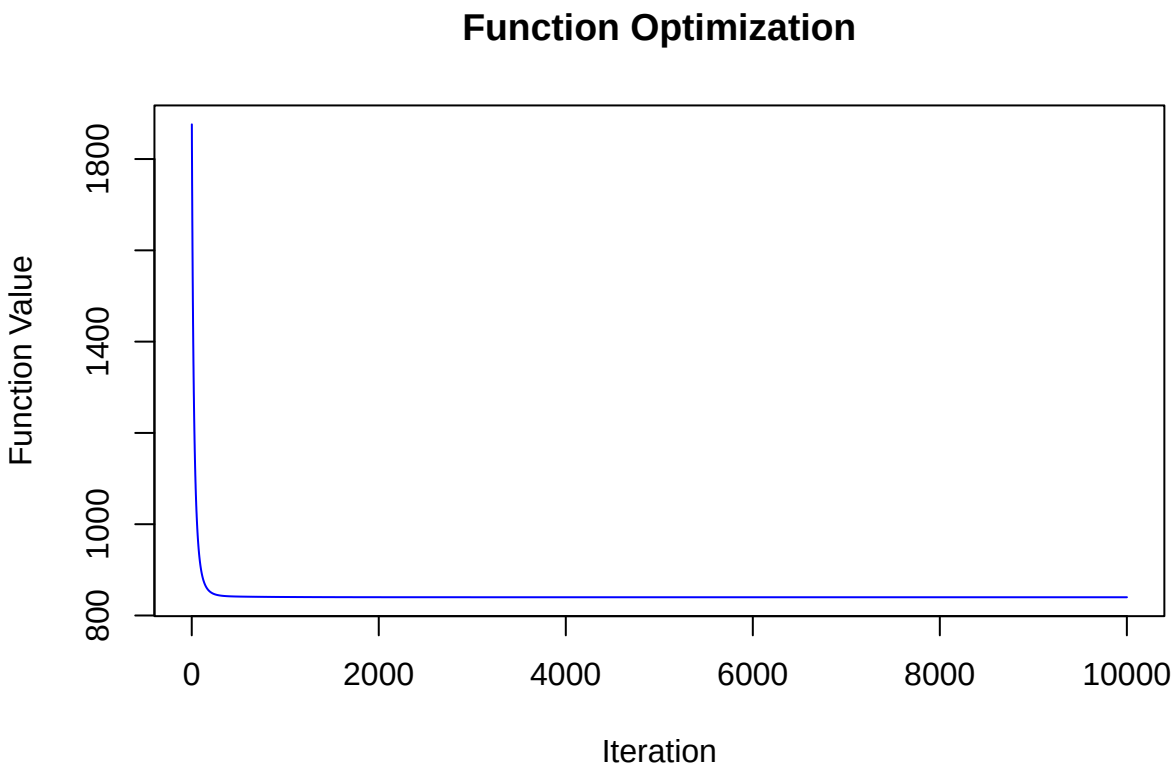
```
# Defining the Goldstein-Price Function
my_function_vec <- function(v) {
     x <- v[1]
     y <- v[2]
     (1 + (x + y + 1)^2 * (19 - 14*x + 3*x^2 - 14*y + 6*x*y + 3*y^2)) *
      (30 + (2*x - 3*y)^2 * (18 - 32*x + 12*x^2 + 48*y - 36*x*y + 27*y^2))
   }
```

1.1: Starting value (1,1), 10000 iterations, gamma of 0.0000005

```
# Perform gradient descent
result <- gradient_descent(my_function_vec,
                           start_value = c(1, 1), max_iter = 10000, gamma=0.0000005)

# Plot the contour with optimization path
# Define the range for x and y based on the function's landscape
plot_contour_with_path(my_function_vec, c(-2, 2), c(-3, 1), result$history)
```




```
plot_optimization(result$history, my_function_vec)
```




```
# Print out the optimized parameters
optimized_parameters <- result$minimizer
cat("Optimized parameters: x =",
    optimized_parameters[1], ", y =", optimized_parameters[2], "\n")
```

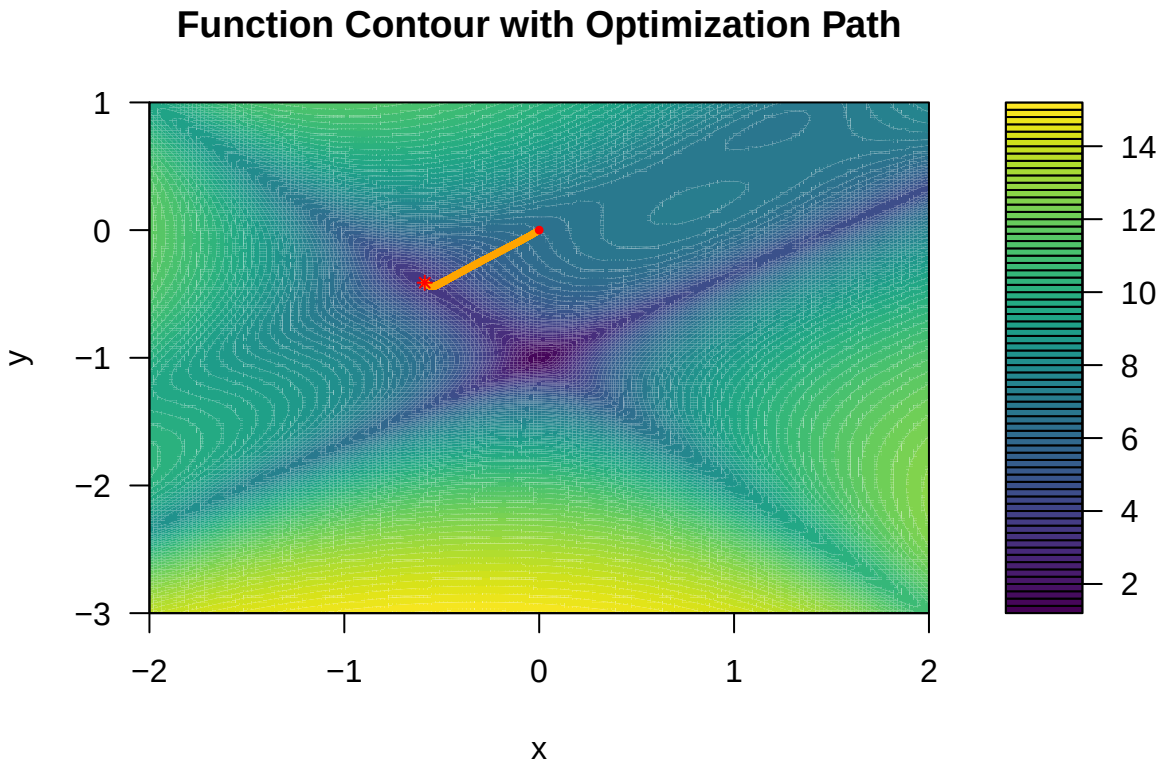
```
## Optimized parameters: x = 1.199998 , y = 0.7999988
```

```
# Evaluate the function at the optimized parameters
optimized_value <- my_function_vec(optimized_parameters)

# Print the minimized function value
cat("Function value at optimized parameters:",
    optimized_value, "\n")
```

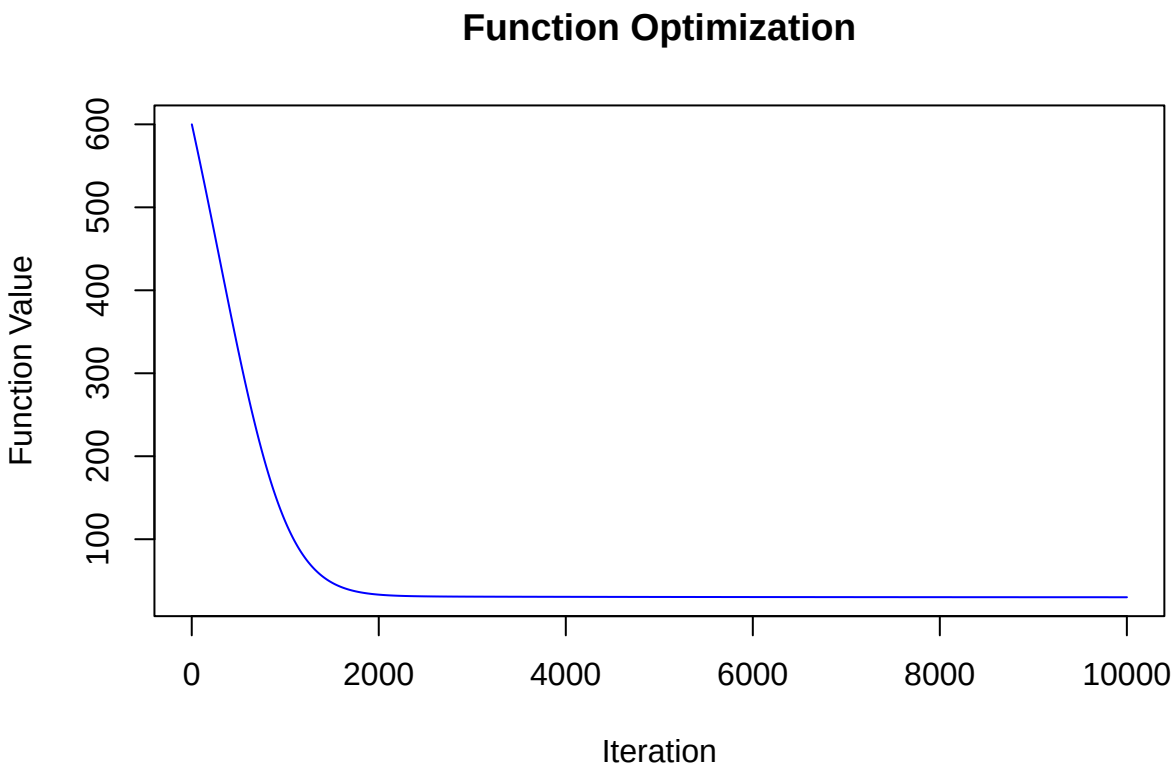
```
## Function value at optimized parameters: 840
```

1.2: Starting value (0,0), 10000 iterations, gamma of 0.0000005

```
# Perform gradient descent
result <- gradient_descent(my_function_vec,
                           start_value = c(0, 0), max_iter = 10000, gamma = 0.0000005)

# Plot the contour with optimization path
# Define the range for x and y based on the function's landscape
plot_contour_with_path(my_function_vec, c(-2, 2), c(-3, 1), result$history)
```




```
plot_optimization(result$history, my_function_vec)
```




```
# Print out the optimized parameters
optimized_parameters <- result$minimizer
cat("Optimized parameters: x =",
    optimized_parameters[1], ", y =", optimized_parameters[2], "\n")
```

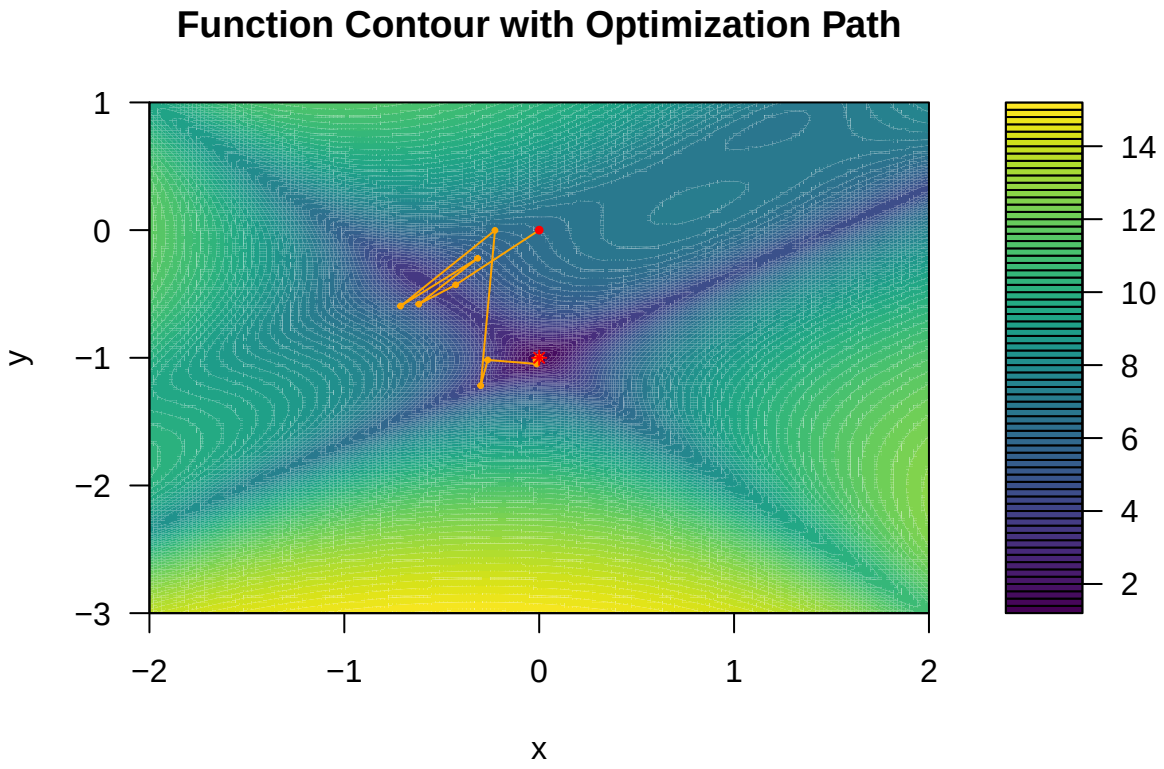
```
## Optimized parameters: x = -0.5873929 , y = -0.4120933
```

```
# Evaluate the function at the optimized parameters
optimized_value <- my_function_vec(optimized_parameters)

# Print the minimized function value
cat("Function value at optimized parameters:",
    optimized_value, "\n")
```

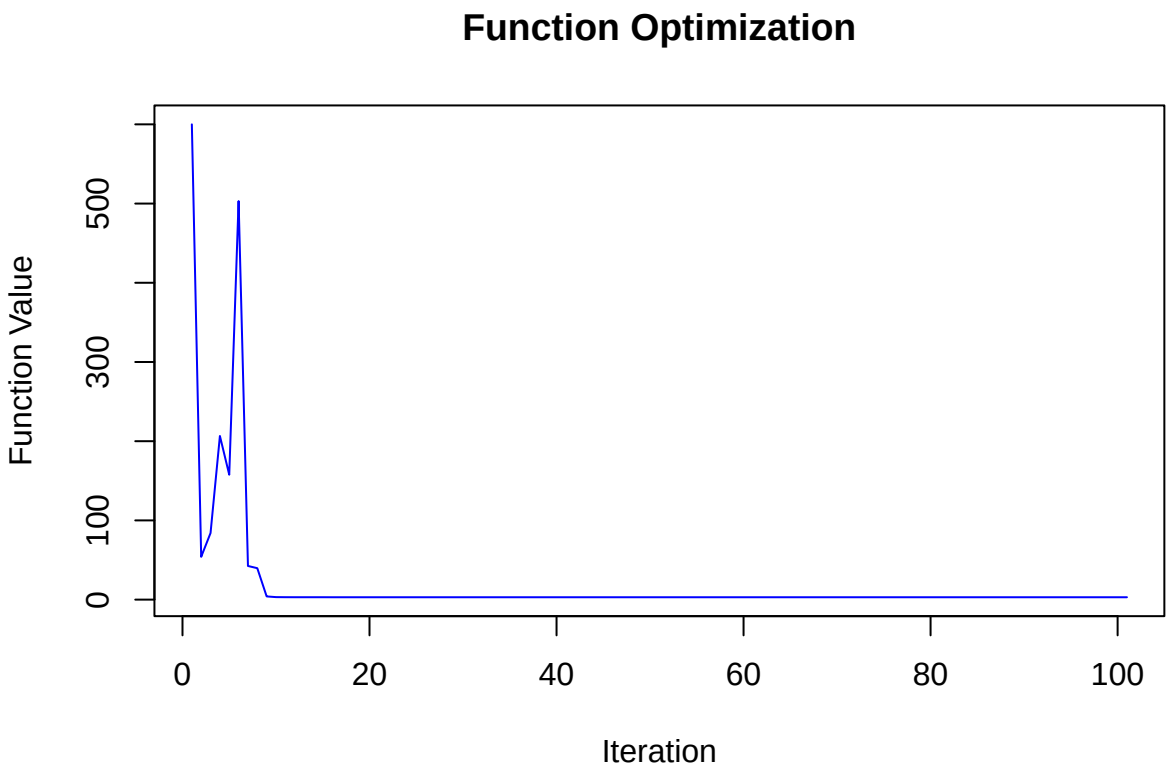
```
## Function value at optimized parameters: 30.06468
```

1.3: Starting value (0,0), 100 iterations, gamma of 0.000595

```
# Perform gradient descent
result <- gradient_descent(my_function_vec,
                           start_value = c(0, 0), max_iter = 100, gamma = 0.000595)

# Plot the contour with optimization path
# Define the range for x and y based on the function's landscape
plot_contour_with_path(my_function_vec, c(-2, 2), c(-3, 1), result$history)
```




```
plot_optimization(result$history, my_function_vec)
```




```
# Print out the optimized parameters
optimized_parameters <- result$minimizer
cat("Optimized parameters: x =",
    optimized_parameters[1], ", y =", optimized_parameters[2], "\n")
```

```
## Optimized parameters: x = -2.311692e-13 , y = -1
```

```
# Evaluate the function at the optimized parameters
optimized_value <- my_function_vec(optimized_parameters)

# Print the minimized function value
cat("Function value at optimized parameters:",
    optimized_value, "\n")
```

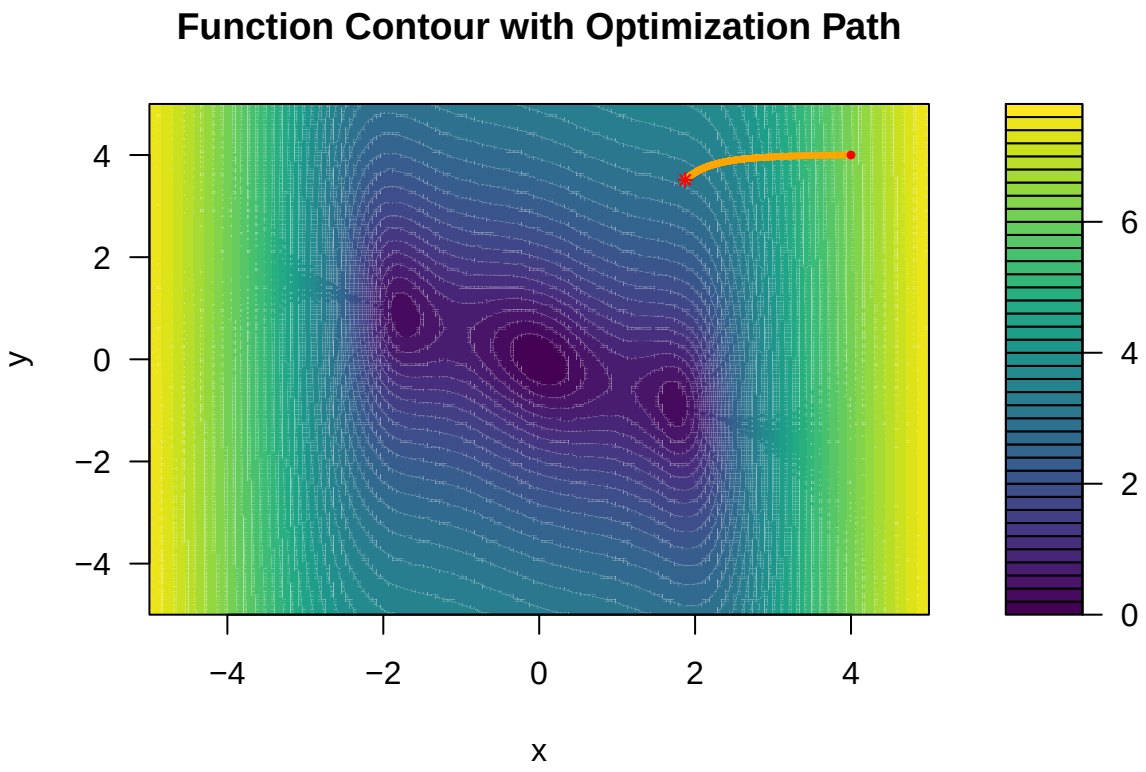
```
## Function value at optimized parameters: 3
```

Example 2: Three-Humped Camel function

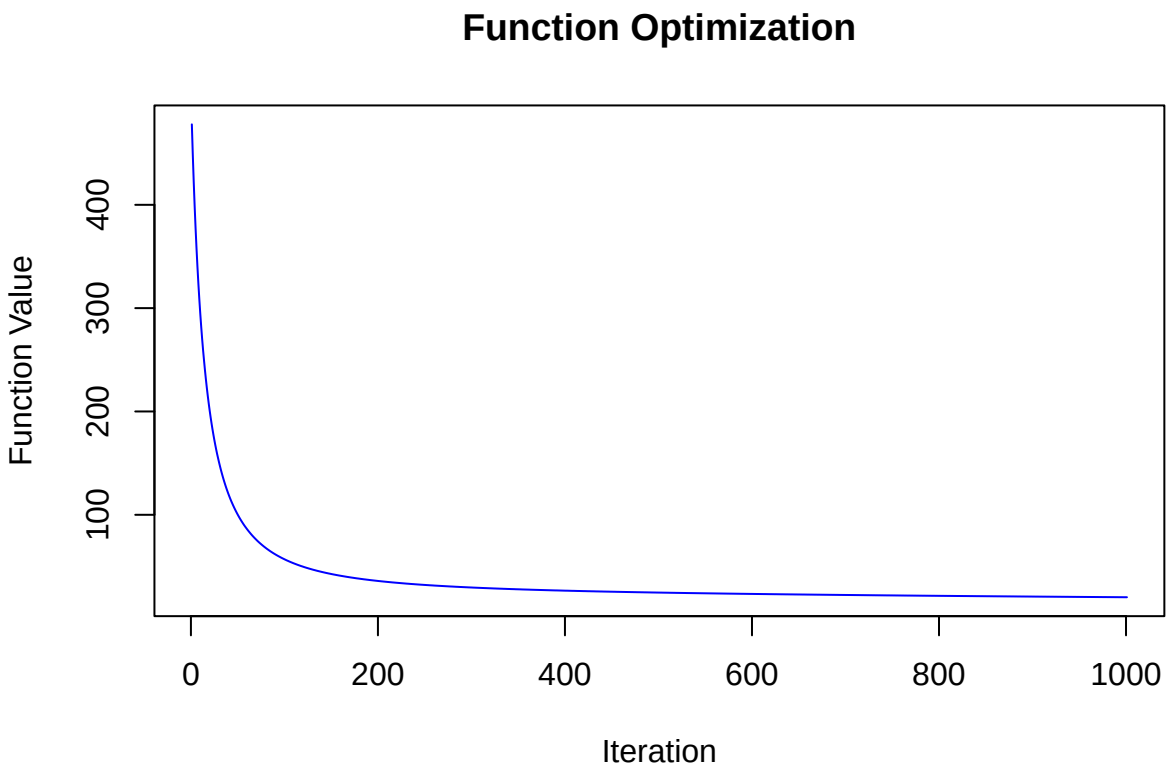
```
# Define the Three-Hump Camel function
three_hump_camel <- function(x) {
  return(2 * x[1]^2 - 1.05 * x[1]^4 + x[1]^6 / 6 + x[1] * x[2] + x[2]^2)
}
```

2.1 Starting value of (4,4), 1000 iterations and gamma of 0.00005

```r
# Perform gradient descent
result <- gradient_descent(three_hump_camel,
                           start_value = c(4, 4), max_iter = 1000, gamma = 0.00005)

# Plot the contour with optimization path
# Define the range for x and y based on the function's landscape
plot_contour_with_path(three_hump_camel, c(-5, 5), c(-5, 5), result$history)
```




```r
plot_optimization(result$history, three_hump_camel)
```




```
# Print out the optimized parameters
optimized_parameters <- result$minimizer
cat("Optimized parameters: x =",
    optimized_parameters[1], ", y =", optimized_parameters[2], "\n")
```

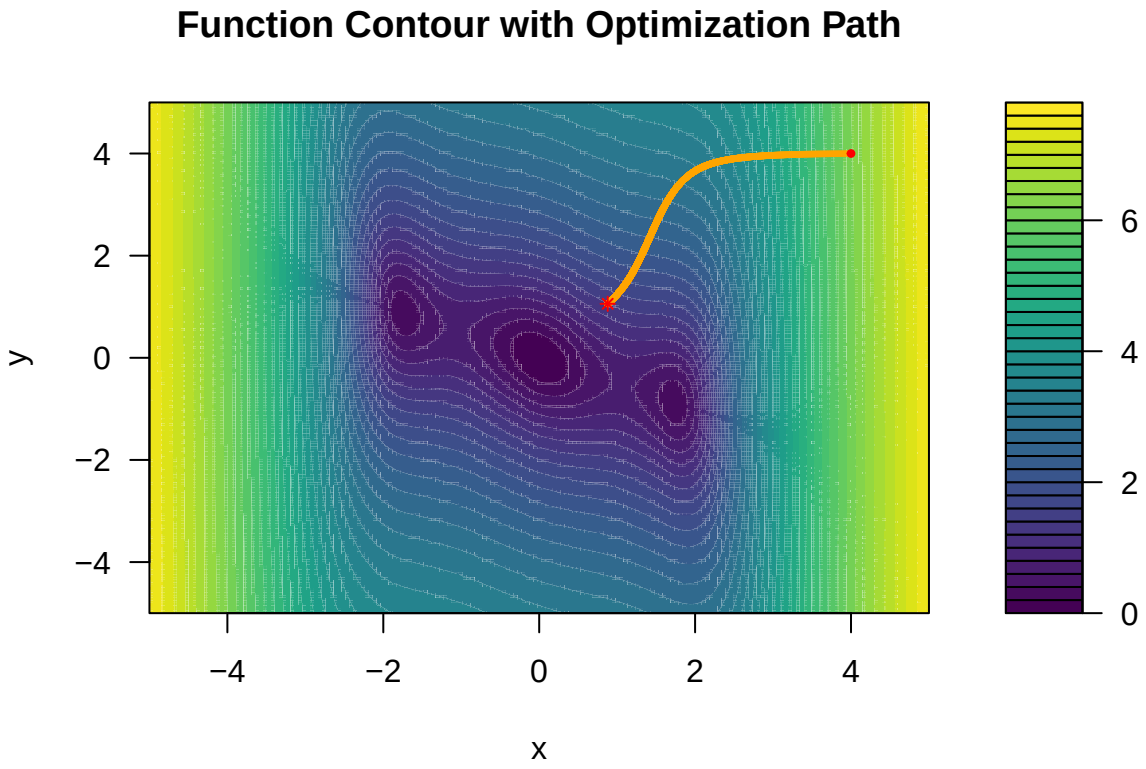
```
## Optimized parameters: x = 1.872738 , y = 3.512937
```

```
# Evaluate the function at the optimized parameters
optimized_value <- three_hump_camel(optimized_parameters)

# Print the minimized function value
cat("Function value at optimized parameters:",
    optimized_value, "\n")
```

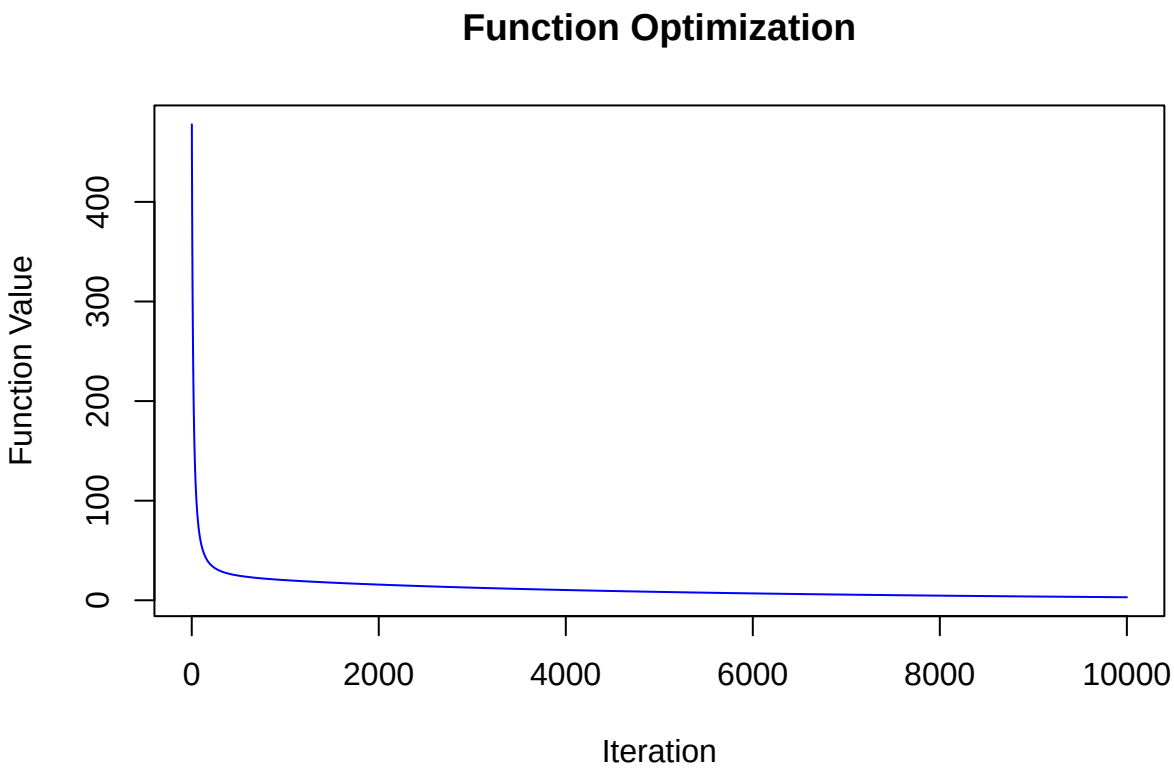
```
## Function value at optimized parameters: 20.20844
```

2.2 Starting value of (4,4), 10000 iterations and gamma of 0.00005

```
# Perform gradient descent
result <- gradient_descent(three_hump_camel,
                           start_value = c(4, 4), max_iter = 10000, gamma = 0.00005)

# Plot the contour with optimization path
# Define the range for x and y based on the function's landscape
plot_contour_with_path(three_hump_camel, c(-5, 5), c(-5, 5), result$history)
```




```
plot_optimization(result$history, three_hump_camel)
```




```
# Print out the optimized parameters
optimized_parameters <- result$minimizer
cat("Optimized parameters: x =",
    optimized_parameters[1], ", y =", optimized_parameters[2], "\n")
```

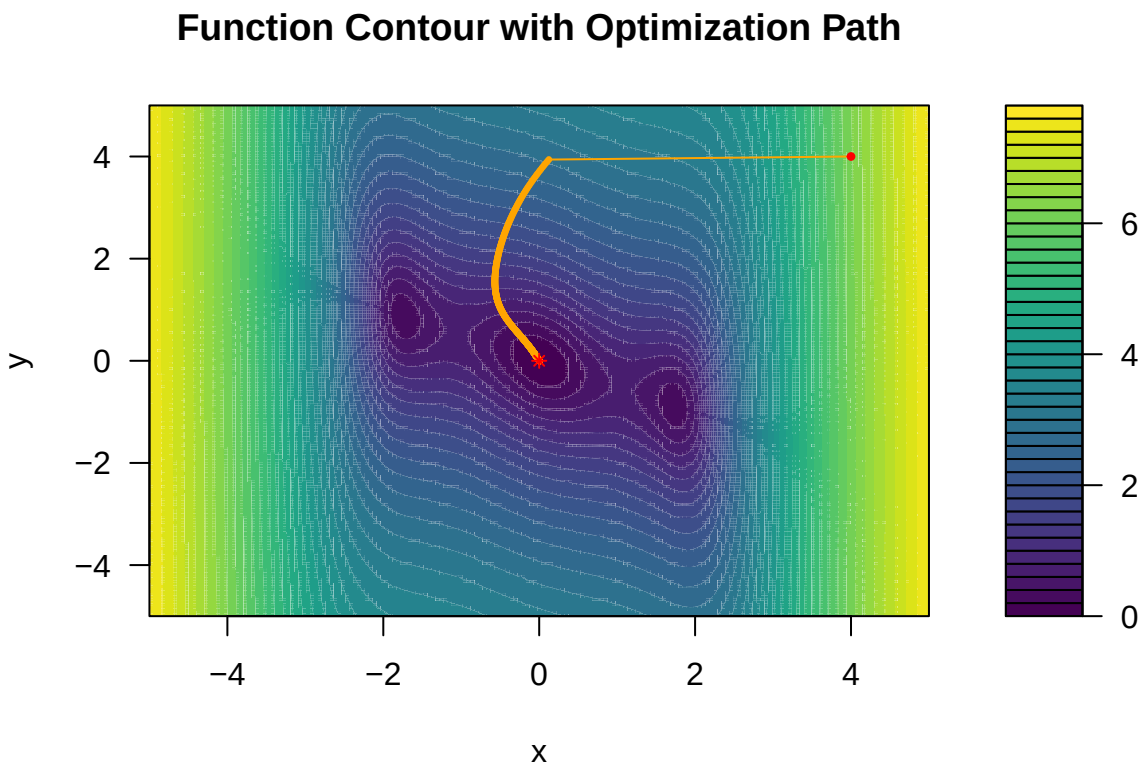
```
## Optimized parameters: x = 0.8788452 , y = 1.057409
```

```
# Evaluate the function at the optimized parameters
optimized_value <- three_hump_camel(optimized_parameters)

# Print the minimized function value
cat("Function value at optimized parameters:",
    optimized_value, "\n")
```

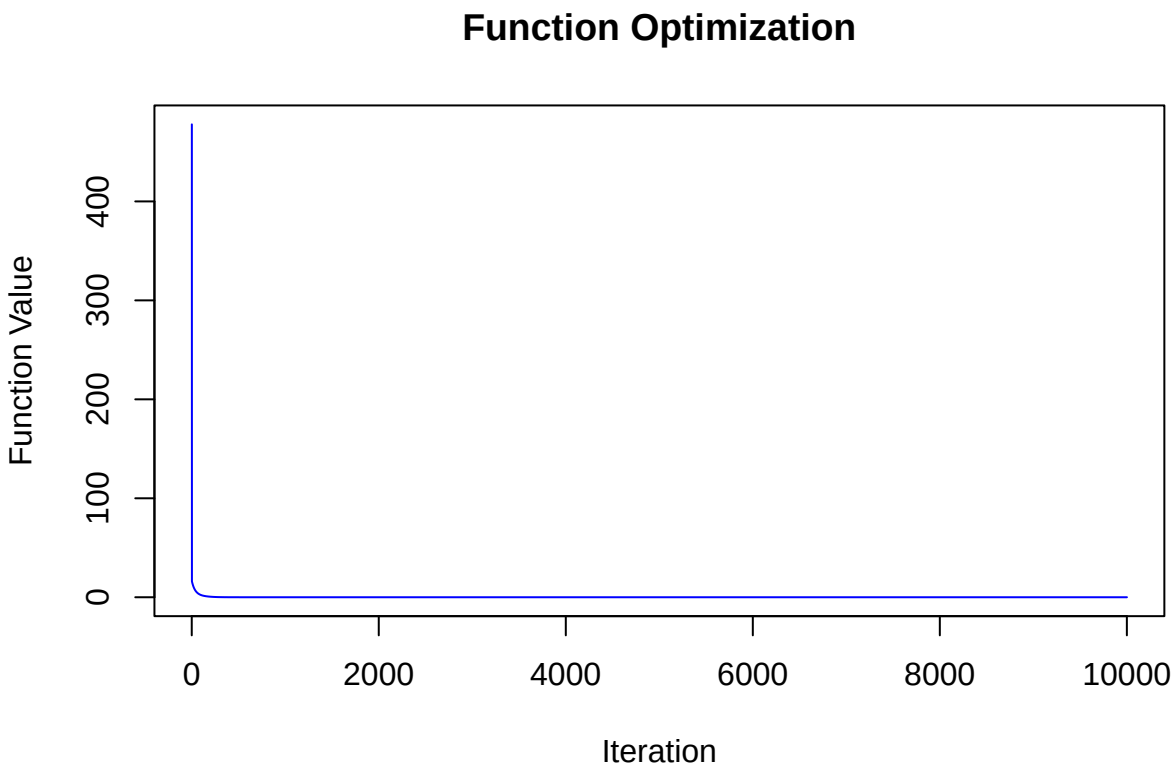
```
## Function value at optimized parameters: 3.042564
```

2.3 Starting value of (4,4), 10000 iterations and gamma of 0.005

```
# Perform gradient descent
result <- gradient_descent(three_hump_camel,
                           start_value = c(4, 4), max_iter = 10000, gamma = 0.005)

# Plot the contour with optimization path
# Define the range for x and y based on the function's landscape
plot_contour_with_path(three_hump_camel, c(-5, 5), c(-5, 5), result$history)
```




```
plot_optimization(result$history, three_hump_camel)
```




```
# Print out the optimized parameters
optimized_parameters <- result$minimizer
cat("Optimized parameters: x =",
    optimized_parameters[1], ", y =", optimized_parameters[2], "\n")
```

```
## Optimized parameters: x = -6.77583e-21 , y = 8.708783e-21
```

```
# Evaluate the function at the optimized parameters
optimized_value <- three_hump_camel(optimized_parameters)

# Print the minimized function value
cat("Function value at optimized parameters:",
    optimized_value, "\n")
```

```
## Function value at optimized parameters: 1.086574e-40
```

We tested our gradient descent algorithm on the Goldman-Price and the Three-Hump Camel functions, analysing convergence performance with varying parameter values. In the case of the more complex (several local minima) Goldman-Price function we first tested the algorithm with a small learning rate (gamma) and a starting position fairly distanced from the known (0,-1) global minimum. This led to weak performance as the algorithm converged to the local minimum in (1.2,0.8) far from the optimum point, as reflected in the very high function value of 840. To address this we first moved the starting point closer to the global minimum to (0,0). While this led to a considerably lower function value of around 30 (which was to be expected due to moving closer to the global minimum) the algorithm nevertheless still got trapped in another local

minimum at (-0.6, -0.4). To solve this, we increased the learning rate which in turn allowed the algorithm to jump over the local minimum (as clearly seen in the contour graph) and eventually converge to the global minimum at (0,-1), minimising the function value at 3. The higher learning rate allowed the algorithm to converge considerably faster, only needing around 100 iterations compared to more than a 1000 in the previous case. The number of iterations could also be limited by adding a threshold value for the minimum marginal improvement.

In the second and simpler example on the Three-Humped Camel function, we started far away (4,4) from the global minimum at (0,0), with a lower initial number of iterations and a relatively small learning rate. The algorithm clearly failed to reach the global minimum as seen on the contour graph. To test if the algorithm was converging too slowly (rather than getting stuck at a local minimum), we increased the number of iterations to 10000. This proved to considerably improve performance, landing significantly closer to the global minimum -confirming that the algorithm was in fact merely slowly converging and not stuck. Because of this, we increased the learning rate to address this issue, which led to the global minimum being reached, and that relatively quickly. With 10000 iterations the minimum value and optimal points of 0 and (0,0) were almost perfectly matched.

```
# PROBLEM B

# Stochastic gradient descent to approximate the ridge estimator
# Optional addition of tolerance parameter determining threshold for
# minimum difference for the algorithm to continue iterating
stochastic_grad <- function(y, X, initial_values, max_iterations, lambda, tol = 1e-4) {
  n <- nrow(X)
  p <- ncol(X)
  a <- initial_values
  # Randomly sampling indexes and computing gradients
  for (m in 1:max_iterations) {
    i <- sample(1:n, 1)
    gi_gradient <- compute_gradient(X[i, ], y[i], a, lambda)
    gamma_m <- 1 / m
    a_new <- a - gamma_m * gi_gradient
    # Check if the threshold was reached
    if (sum((a_new - a)^2) < tol) {
      message("Threshold reached in ", m, " iterations.")
      break
    }

    a <- a_new
  }

  return(a)
}

# Define function to compute the gradient
compute_gradient <- function(xi, yi, a, lambda) {
  residual <- yi - sum(a * xi)
  gradient <- -2 * xi * residual
  gradient[2:length(gradient)] <-
    gradient[2:length(gradient)] + 2 * lambda * a[2:length(a)]
  return(gradient)
}

# Simulate/use data of choice
set.seed(123)
```

```r
n <- 1000
p2 <- 20
X <- matrix(rnorm(n * p2), n, p2)
beta_true <- runif(20, min = -3, max = 3)
y <- X %*% beta_true + rnorm(n)

initial_values <- rep(0, p2)
max_iterations <- 10000
lambda <- 0.1

# Calling the stochastic gradient function to estimate the parameters
sgd_result <- stochastic_grad(y, X, initial_values, max_iterations, lambda)
```

```
## Threshold reached in 234 iterations.
```

```r
print("Custom Ridge Estimator:")
```

```
## [1] "Custom Ridge Estimator:"
```

```r
print(sgd_result)
```

```
##  [1] -2.0564552  1.4208058  0.9419958  1.1125212 -2.3903050 -2.2533960
##  [7] -4.5543633  1.1940661 -0.7199086 -0.7946098 -3.5604074 -1.2538860
## [13] -1.4104107  1.1161669  1.7076634  1.5879321 -1.9281409 -4.1638569
## [19] -0.1514946  1.7004603
```

```r
# Using glmnet for Ridge Regression:
x_matrix <- Matrix(X, sparse = TRUE)

# Fit Ridge Regression model with glmnet (For ridge regression, alpha = 0)
ridge_glmnet_model <- glmnet(x_matrix, y, alpha = 0, lambda = lambda)
print("glmnet Ridge Estimator:")
```

```
## [1] "glmnet Ridge Estimator:"
```

```r
# Extracting coefficients (excluding intercept by indexing from -1)
ridge_glmnet_coef <- as.vector(coef(ridge_glmnet_model, s = lambda)[-1])
print(ridge_glmnet_coef)
```

```
##  [1] -1.75538481  2.79296441 -0.53278869 -0.14950127 -2.83361459 -0.80583717
##  [7] -2.65328729 -0.24732971 -2.12005193 -0.12686908 -2.65581858 -1.57291488
## [13] -0.10034528  0.55561813  2.54499019  0.09247588 -0.93504248 -2.13238980
## [19]  1.67435340  0.92015088
```

```r
# Stochastic gradient descent to approximate the ridge estimator
# Optional addition of tolerance parameter determining threshold for
# minimum difference for the algorithm to continue iterating
stochastic_grad <- function(y, X, initial_values, max_iterations, lambda, tol = 1e-8) {
  n <- nrow(X)
  p <- ncol(X)
```

```
  a <- initial_values
  # Randomly sampling indexes and computing gradients
  for (m in 1:max_iterations) {
    i <- sample(1:n, 1)
    gi_gradient <- compute_gradient(X[i, ], y[i], a, lambda)
    gamma_m <- 1 / m
    a_new <- a - gamma_m * gi_gradient
    # Check if the threshold was reached
    if (sum((a_new - a)^2) < tol) {
      message("Threshold reached in ", m, " iterations.")
      break
    }

    a <- a_new
  }

  return(a)
}

# Define function to compute the gradient
compute_gradient <- function(xi, yi, a, lambda) {
  residual <- yi - sum(a * xi)
  gradient <- -2 * xi * residual
  gradient[2:length(gradient)] <-
    gradient[2:length(gradient)] + 2 * lambda * a[2:length(a)]
  return(gradient)
}


# Simulate/use data of choice
set.seed(123)
n <- 1000
p2 <- 20
X <- matrix(rnorm(n * p2), n, p2)
beta_true <- runif(20, min = -3, max = 3)
y <- X %*% beta_true + rnorm(n)

initial_values <- rep(0, p2)
max_iterations <- 10000
lambda <- 0.1

# Calling the stochastic gradient function to estimate the parameters
sgd_result <- stochastic_grad(y, X, initial_values, max_iterations, lambda)
print("Custom Ridge Estimator:")
```

```
## [1] "Custom Ridge Estimator:"
```

```
print(sgd_result)
```

```
##  [1] -1.72203079  2.54611040 -0.49280710 -0.14467162 -2.61071462 -0.73071899
##  [7] -2.45166463 -0.20883869 -1.98527049 -0.14408253 -2.41852967 -1.41961032
## [13] -0.14098064  0.51624362  2.34365054  0.08651787 -0.84608023 -1.93349008
## [19]  1.50981213  0.85123291
```

```r
# Using glmnet for Ridge Regression:
x_matrix <- Matrix(X, sparse = TRUE)

# Fit Ridge Regression model with glmnet (For ridge regression, alpha = 0)
ridge_glmnet_model <- glmnet(x_matrix, y, alpha = 0, lambda = lambda)
print("glmnet Ridge Estimator:")
```

```
## [1] "glmnet Ridge Estimator:"
```

```r
# Extracting coefficients (excluding intercept by indexing from -1)
ridge_glmnet_coef <- as.vector(coef(ridge_glmnet_model, s = lambda)[-1])
print(ridge_glmnet_coef)
```

```
##  [1] -1.75538481  2.79296441 -0.53278869 -0.14950127 -2.83361459 -0.80583717
##  [7] -2.65328729 -0.24732971 -2.12005193 -0.12686908 -2.65581858 -1.57291488
## [13] -0.10034528  0.55561813  2.54499019  0.09247588 -0.93504248 -2.13238980
## [19]  1.67435340  0.92015088
```

We randomly sampled a dataset and tested the sgd model with different threshold values. In the first case, a higher threshold value was used and led to a faster termination of the algorithm. This came at a cost of the model's accuracy as it failed to converge towards the 'true' value of the Ridge estimator. However, in the second case, a decrease in the threshold led to a higher number of iterations and considerably better convergence to the 'true' values of the Ridge estimator. This makes sense as a lower threshold implies a higher required precision. This can naturally be replicated with different/larger datasets. The low number of parameters was chosen for readability purposes.