Created by:

Importing the libraries

```
# Installing 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
```

Problem A Gradient Descent Function

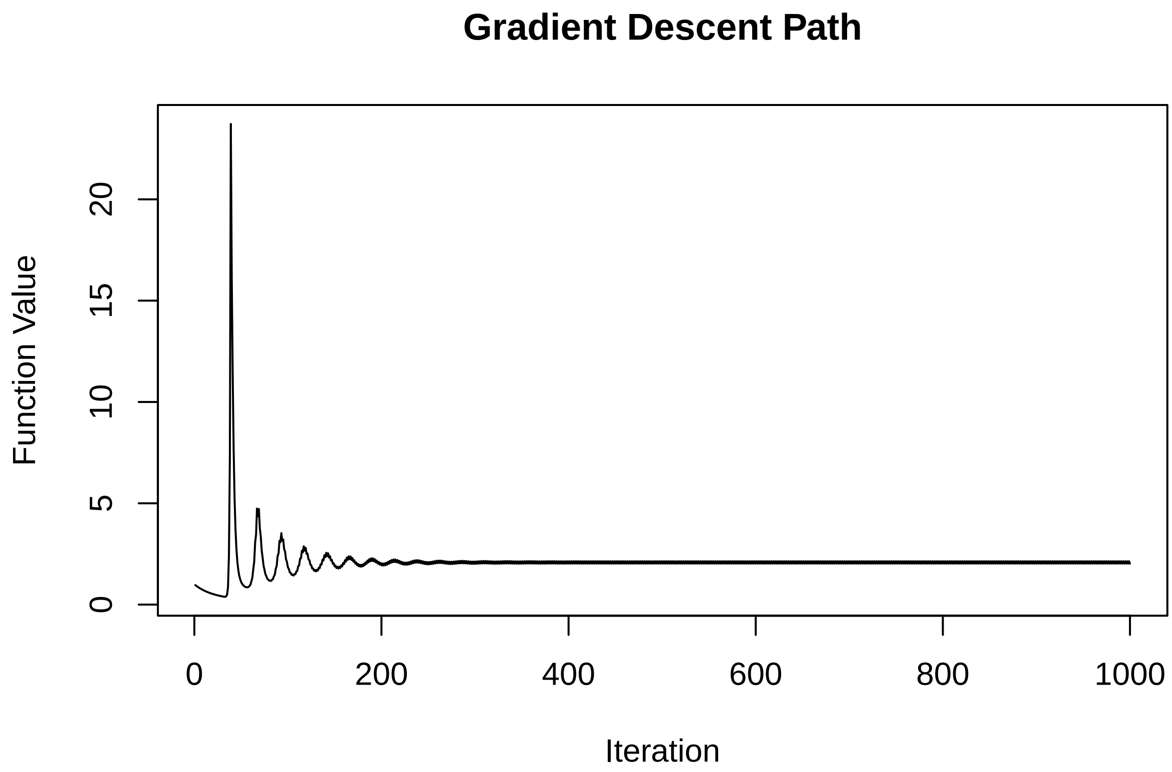
```
# Define the Gradient Descent Function
# x = current_value
# learning_rate = gamma
gradient_descent <- function(f, start_value, max_iterations, learning_rate) {
  x <- start_value
  path <- numeric(max_iterations)

  for (i in 1:max_iterations) {
    gradient <- grad(f, x)
    x <- x - learning_rate * gradient
    path[i] <- f(x)
  }

  return(list(minimizer = x, values_path = path))
}
```

Testing the gradient Descent Function on the Rosenbrock function

```
# Example usage
# Define a test function, e.g., the Rosenbrock function
rosenbrock <- function(x) {
  sum(100 * (x[2:length(x)] - x[1:(length(x) - 1)]^2)^2 + (1 - x[1:(length(x) - 1)])^2)
}

# Set parameters
start_value <- c(0, 0)  # Initial guess
max_iterations <- 1000   # Maximum number of iterations
learning_rate <- 0.009  # Experiment with different choices

# Run gradient descent
result <- gradient_descent(rosenbrock, start_value, max_iterations, learning_rate)

# Print the minimizer and the path of function values
cat("Minimizer:", result$minimizer, "\n")
```

```
## Minimizer: 0.2013663 -0.0770897
```

```r
#cat("Function values path:", result$values_path, "\n")

# Plot the path of function values
plot(result$values_path, type = 'l', xlab = 'Iteration', ylab = 'Function Value',
     main = 'Gradient Descent Path')
```



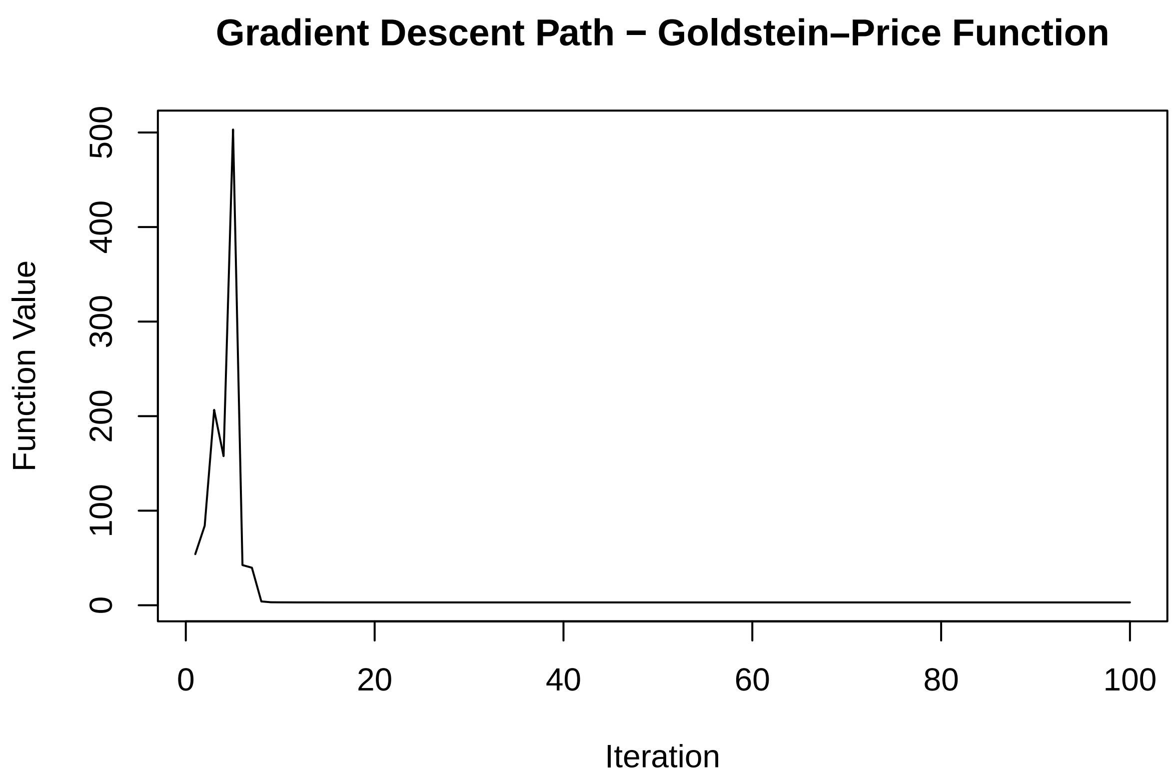
Testing the Gradient Descent Function on the Goldstein-Price Function

```r
# Define the Goldstein-Price function
goldstein_price <- function(x) {
  term1 <- 1 + (x[1] + x[2] + 1)^2 * (19 - 14 * x[1] + 3 * x[1]^2 - 14 * x[2] + 6 * x[1] * x[2] + 3 * x
  term2 <- 30 + (2 * x[1] - 3 * x[2])^2 * (18 - 32 * x[1] + 12 * x[1]^2 + 48 * x[2] - 36 * x[1] * x[2]
  return(term1 * term2)
}

# Set parameters
start_value <- c(0, 0)  # Initial guess
max_iterations <- 100   # Maximum number of iterations
learning_rate <- 0.000595  # Experiment with different choices

# Run gradient descent
# Using optim function for optimization
result_optim <- optim(par = start_value, fn = goldstein_price, method = "L-BFGS-B")
result <- gradient_descent(goldstein_price, start_value, max_iterations, learning_rate)
```

```r
# Print the minimizer and the path of function values
cat("Minimizer:", result$minimizer, "\n")
```

```
## Minimizer: -2.311692e-13 -1
```

```r
#cat("Function values path:", result$values_path, "\n")

# Plot the path of function values
plot(result$values_path, type = 'l', xlab = 'Iteration', ylab = 'Function Value',
     main = 'Gradient Descent Path - Goldstein-Price Function')
```



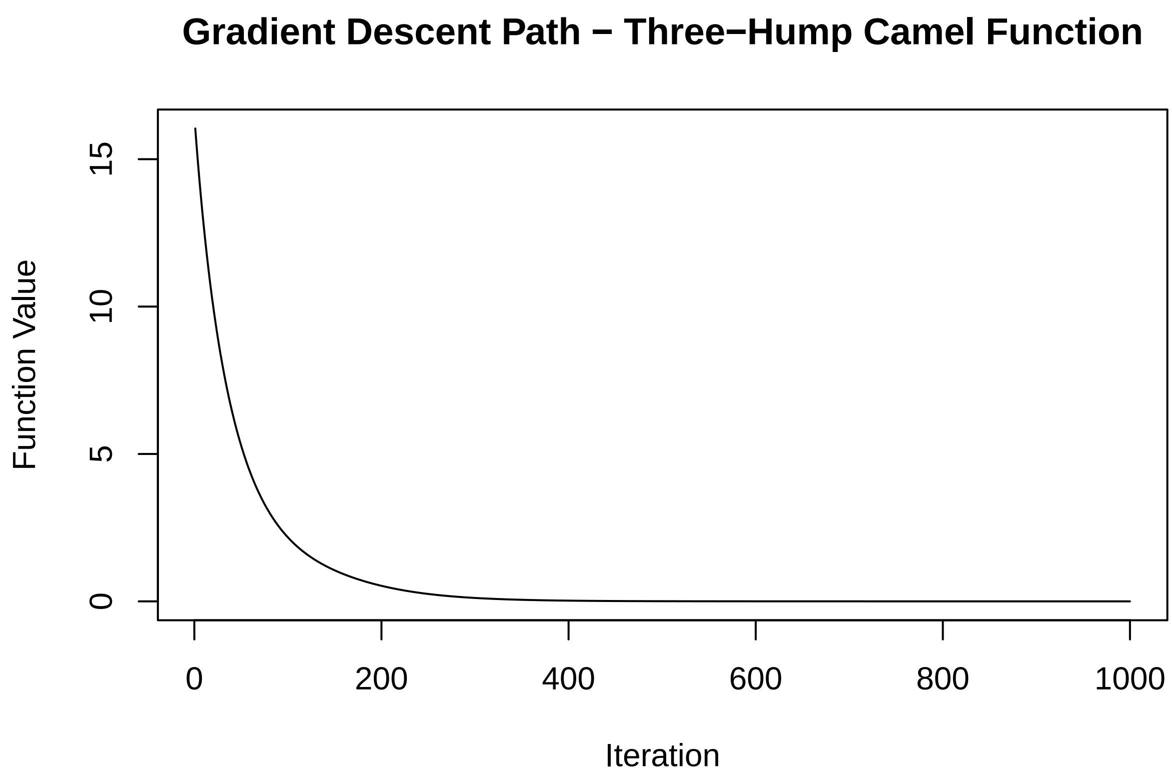
Testing the Gradient Descent Function on the Three-Hump Camel function

```r
# 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)
}

# Set parameters
start_value <- c(4, 4)  # Initial guess
max_iterations <- 1000   # Maximum number of iterations
learning_rate <- 0.005    # Experiment with different choices

# Run gradient descent
result <- gradient_descent(three_hump_camel, start_value, max_iterations, learning_rate)
```

```r
# Print the minimizer and the path of function values
cat("Minimizer:", result$minimizer, "\n")
```

```
## Minimizer: -0.0005569528 0.001344584
```

```r
#cat("Function values path:", result$values_path, "\n")

# Plot the path of function values
plot(result$values_path, type = 'l', xlab = 'Iteration', ylab = 'Function Value',
     main = 'Gradient Descent Path - Three-Hump Camel Function')
```



Problem B

```r
ridge_estimator <- function(y, X, initial_values, max_iterations, lambda, tol = 1e-6) {
  n <- nrow(X)
  p <- ncol(X)
  a <- initial_values

  for (m in 1:max_iterations) {
    # Sample an index i at random from {1, ..., n}
    i <- sample(1:n, 1)

    # Compute the gradient of gi at a
    gi_gradient <- compute_gradient(X[i, ], y[i], a, lambda)
```

```
    # Update a using stochastic gradient descent
    gamma_m <- 1 / m
    a_new <- a - gamma_m * gi_gradient

    # Check for convergence
    if (sum((a_new - a)^2) < tol) {
      message("Convergence achieved after ", m, " iterations.")
      break
    }

    # Update coefficients
    a <- a_new
  }

  return(a)
}

compute_gradient <- function(xi, yi, a, lambda) {
  # Compute the gradient of gi at a
  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)
}

# Example usage:
set.seed(100)  # Set seed for reproducibility
n <- 100
p <- 5
X <- matrix(rnorm(n * p), n, p)
beta_true <- c(2, 1.5, -1, 0.5, -2)
y <- X %*% beta_true + rnorm(n)

initial_values <- rep(0, p)
max_iterations <- 1000
lambda <- 0.1

ridge_result <- ridge_estimator(y, X, initial_values, max_iterations, lambda)
```

```
## Convergence achieved after 451 iterations.
```

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

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

```
print(ridge_result)
```

```
## [1]  1.960826  1.284147 -1.084287  0.634058 -1.699416
```

```r
# Compare with OLS estimator
ols_result <- lm(y ~ X - 1)
print("OLS Estimator:")
```

```
## [1] "OLS Estimator:"
```

```r
print(coef(ols_result))
```

```
##        X1        X2        X3        X4        X5
##  1.946267  1.490350 -1.108681  0.562548 -1.857924
```