HW 1

Bryan Mui - UID 506021334 - 14 April 2025

Loaded packages: ggplot2, tidyverse (include = false for this chunk)

Problem 1

Part a

Find the theoretical min for the function:

$$f(x) = x^4 + 2x^2 + 1$$

Solution: find f'(x) and f' '(x), set f'(x) to 0 and solve, and f' '(x) needs to be > 0 to be a min

Step 1: find f'(x) and f' '(x)

$$\begin{align} f(x) &= x^4 + 2x^2 + 1 \\ f'(x) &= 4x^3 + 4x \\ f''(x) &= 12x^2 + 4 \\ \end{align}$$

Step 2: set f'(x) to 0 and solve

$$\begin{align} f'(x) &= 4x^3 + 4x \\ 0 &= 4x^3 + 4x \\ 0 &= 4x(x^2 + 4) \end{align}$$

We get

$$x = 0$$

and

$$0 = x^2 + 4$$

which has no real solution

Step 3: check that f' '(x) needs to be > 0 to be a min

Our critical point is x = 0,

$$\begin{aligned} f''(0) &= 12(0)^2 + 4 && (8) \\ &= 4 && (9) \end{aligned}$$

Since f'(x) = 0 at 0 and f' '(x) > 0 at that point, **we have a min at x = 0, and plugging into f(0) we get the minimum point**

$$(0, 1)$$

Part b

0)

Use the gradient descent algorithm with **constant step size** and with **back-tracking line search** to calculate x_{min}

Constant step size descent is implemented as follows:

1. Select a random starting point x_0

2. While stopping criteria < tolerance, do:

- Select η_k(as a constant)

- Calculate $x_{(k+1)} = x_k - \eta_k * \nabla(f(x_k))$

- Calculate the value of stopping criterion

Stopping criteria: Stop if $| |\nabla(f(x_k)||_2 \leq \epsilon$

```r
# Gradient descent algorithm that uses backtracking to minimize an objective
 ↪  function

gradient_descent_constant_step <- function(tol = 1e-6, max_iter = 10000,
 ↪  step_size = 0.01) {
  # Step 1: Initialize and select a random stopping point
  # Initialize
  set.seed(777) # example seeding
  last_iter <- 0 # the last iteration ran
  eta <- step_size # step size that is decided manually
  max_iter <- max_iter # max iterations before terminating if mininum isn't
 ↪  found
  tolerance <- tol # tolerance for the stoppign criteria
  obj_values <- numeric(max_iter) # Stores the value of f(x)
  eta_values <- numeric(max_iter)  # To store eta values used each iteration
  eta_values[1] <- step_size
  betas <- numeric(max_iter) # Stores the value of x guesses
  x0 <- runif(1, min=-10, max=10) # our first guess is somewhere between
 ↪  -10-10

  # Set the objective function to the function to be minimized
  # Objective function: f(x)
  obj_function <- function(x) {
    return(x^4 + 2*(x^2) + 1)
  }

  # Gradient function: d/dx of f(x)
  gradient <- function(x) {
    return(4*x^3 + 4*x)
  }

  # Append the first guess to the obj_values and betas vector
  betas[1] <- x0
  obj_values[1] <- obj_function(x0)

  # Step 2: While stopping criteria < tolerance, do:
  for (iter in 1:max_iter) { # the iteration goes n = 1, 2, 3, 4, but the
   ↪  arrays of our output starts at iter = 0 and guess x0
    # Select eta(step size), which is constant
    # There's nothing to do for this step

    # Calculate the next guess of x_k+1, calculate f(x_k+1), set eta(x_k+1)
    betas[iter + 1] <- betas[iter] - (eta * gradient(betas[iter]))
    obj_values[iter + 1] <- obj_function(betas[iter + 1])
    eta_values[iter + 1] <- eta

    # Calculate the value of the stopping criterion
    stop_criteria <- abs(gradient(betas[iter + 1]))

    # If stopping criteria less than tolerance, break
    if(is.na(stop_criteria) || stop_criteria <= tolerance) {
      last_iter <- iter + 1
      break
    }

    # if we never set last iter, then we hit the max number of iterations and
     ↪  need to set
    if(last_iter == 0) { last_iter <- max_iter }

    # end algorithm
  }

  return(list(betas = betas, obj_values = obj_values, eta_values =
   ↪  eta_values, last_iter = last_iter)) # in this case, beta(predictors)
   ↪  are the x values, obj_values are f(x), eta is the step size, last iter
   ↪  is the value in the vector of the final iteration before stopping
}
```

Running the gradient descent algorithm with fixed step size:

```
minimize_constant_step <- gradient_descent_constant_step(tol = 1e-6, max_iter
 ↪  = 10000, step_size = 0.03)
print(minimize_constant_step)
```

```
$betas
 [1]  3.757148134 -3.058091092  0.740763042  0.603094019  0.504399680
 [6]  0.428472253  0.367616075  0.317540520  0.275593469  0.240010435
[11]  0.209550087  0.183299884  0.160564858  0.140800331  0.123569332
[16]  0.108514593  0.095339505  0.083794772  0.073668795  0.064780563
[21]  0.056974273  0.050115167  0.044086243  0.038785612  0.034124337
[26]  0.030024648  0.026418442  0.023246017  0.020454987  0.017999362
[31]  0.015838739  0.013937613  0.012264775  0.010792780  0.009497496
[36]  0.008357693  0.007354700  0.006472088  0.005695405  0.005011934
```

```
[41]  0.004410487  0.003881218  0.003415465  0.003005605  0.002644929
[46]  0.002327535  0.002048229  0.001802441  0.001586147  0.001395809
 [ reached 'max' / getOption("max.print") -- omitted 9950 entries ]

$obj_values
 [1] 228.498357 107.162271   2.398564   1.859739   1.573567   1.400882
 [7]   1.288546   1.211831   1.157672   1.118528   1.089751   1.068327
[13]   1.052227   1.040042   1.030772   1.023689   1.018262   1.014092
[19]   1.010884   1.008411   1.006503   1.005029   1.003891   1.003011
[25]   1.002330   1.001804   1.001396   1.001081   1.000837   1.000648
[31]   1.000502   1.000389   1.000301   1.000233   1.000180   1.000140
[37]   1.000108   1.000084   1.000065   1.000050   1.000039   1.000030
[43]   1.000023   1.000018   1.000014   1.000011   1.000008   1.000006
[49]   1.000005   1.000004
 [ reached 'max' / getOption("max.print") -- omitted 9950 entries ]

$eta_values
 [1] 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03
[16] 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03
[31] 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03
[46] 0.03 0.03 0.03 0.03 0.03
 [ reached 'max' / getOption("max.print") -- omitted 9950 entries ]

$last_iter
[1] 118
```

```
cat("The functions stopped after", minimize_constant_step$last_iter - 1,
 ↪  "iterations \n")
```

```
The functions stopped after 117 iterations
```

```
cat("The function's point of minimization is", "(",
 ↪  minimize_constant_step$betas[minimize_constant_step$last_iter], "," ,
 ↪  minimize_constant_step$obj_values[minimize_constant_step$last_iter], ")
 ↪  \n")
```

```
The function's point of minimization is ( 2.342325e-07 , 1 )
```

Backtracking Line Search is implemented as follows:

1. Select a random starting point x_0

2. While stopping criteria $<$ tolerance, do:

- Select η_k using backtracking line search
- Calculate $x_{(k+1)} = x_k - \eta_k * \nabla(f(x_k))$

- Calculate the value of stopping criterion

Backtracking Line Search:

- Set $\eta^0 > 0$(usually a large value), $\epsilon \in (0, 1)$ and $\tau \in (0, 1)$
- Set $\eta_1 = \eta^0$
- At iteration k, set $\eta_k < -\eta_{k-1}$

  1. Check whether the Armijo Condition holds:

  $$h(\eta_k) \leq h(0) + \epsilon \eta_k h'(0)$$

     where $h(\eta_k) = f(x_k) - \eta_k \nabla f(x_k)$
  2.

  - If yes(condition holds), terminate and keep η_k
  - If no, set $\eta_k = \tau \eta_k$ and go to Step 1

Stopping criteria: Stop if $||\nabla(f(x_k)||_2 \leq \epsilon$

Other note: Since we need h'(0) for the Armijo condition calculation, that is given by:

$$h'(0) = -[\nabla f(x_k)]^\top \nabla f(x_k)$$

Since we are minimizing x, we have a one dimensional beta, we can simplify to

$$h'(0) = -||\nabla f(x_k)||^2$$

To summarize, backtracking line search chooses the step size by ensuring the Armijo condition always holds. If the Armijo condition doesn't hold, we are probably overshooting, hence the step size gets updated iteratively

```
gradient_descent_backtracking <- function(tol = 1e-6, max_iter = 10000,
 ↪  epsilon = 0.5, tau = 0.5, init_step_size = 1) {
  # Step 1: Initialize and select a random stopping point
  # Initialize
  set.seed(777) # example seeding
  last_iter <- 0 # the last iteration ran
```

```
max_iter <- max_iter # max iterations before terminating if minimum isn't
↪  found
tolerance <- tol # tolerance for the stopping criteria
epsilon <- epsilon # Epsilon used in the step size criteria calculation
tau <- tau # tau used in the step size criteria calculation
obj_values <- numeric(max_iter) # Stores the value of f(x)
eta_values <- numeric(max_iter)  # To store eta values used each iteration
eta_values[1] <- init_step_size
betas <- numeric(max_iter) # Stores the value of x guesses
x0 <- runif(1, min=-10, max=10) # our first guess is somewhere between -10
↪  to 10
eta <- init_step_size # our initial step size

# Set the objective function to the function to be minimized
# Objective function: f(x)
obj_function <- function(x) {
  return(x^4 + 2*(x^2) + 1)
}

# Gradient function: d/dx of f(x)
gradient <- function(x) {
  return(4*x^3 + 4*x)
}

# Armijo condition function
# returns TRUE or FALSE whether the condition is satisfied or not
armijo_stepsize <- function(beta, eta, grad, f, epsilon, tau, max_iter) {
  subiter <- 1 # set a hard limit of iterations
  # calc armijo
  beta_new <- beta - (eta)*grad(beta)
  armijo <- f(beta_new) > (f(beta) - epsilon*eta*sum(grad(beta)^2))
  while (armijo && (iter <= max_iter)) {
    #update eta
    eta <- tau * eta

    #recalculate armijo
    beta_new <- beta - (eta)*grad(beta)
    armijo <- f(beta_new) > (f(beta) - epsilon*eta*sum(grad(beta)^2))

    subiter <- subiter + 1
  }
  return(eta)
}
```

```r
# Append the first guess to the obj_values and betas vector
betas[1] <- x0
obj_values[1] <- obj_function(x0)

# Step 2: While stopping criteria < tolerance, do:
for (iter in 1:max_iter) { # the iteration goes n = 1, 2, 3, 4, but the
 ↪  arrays of our output starts at iter = 0 and guess x0
  beta <- betas[iter]

  # use BLS to calculate eta
  eta <- armijo_stepsize(beta = beta, eta = eta, grad = gradient, f =
↪  obj_function, epsilon = epsilon, tau = tau, max_iter = max_iter)
  eta_values[iter + 1] <- eta

  # Calculate the next guess of x_k+1
  beta_new <- beta - (eta * gradient(beta))
  betas[iter + 1] <- beta_new

  # calculate f(x_k+1), to keep track obj values
  obj_values[iter + 1] <- obj_function(beta_new)

  # Calculate the value of the stopping criterion
  stop_criteria <- abs(gradient(beta_new))

  # If stopping criteria less than tolerance, break
  if(is.na(stop_criteria) || stop_criteria <= tolerance) {
    last_iter <- iter + 1
    break
  }

  # if we never set last iter, then we hit the max number of iterations and
   ↪  need to set
  if(last_iter == 0) { last_iter <- max_iter }

  # end algorithm
}

return(list(betas = betas, obj_values = obj_values, eta_values =
 ↪  eta_values, last_iter = last_iter)) # in this case, beta(predictors)
 ↪  are the x values, obj_values are f(x), eta is the step size, last iter
 ↪  is the value in the vector of the final iteration before stopping
```

```
}
```

Running the gradient descent algorithm with backtracking:

```
minimize_backtrack <- gradient_descent_backtracking(tol = 1e-6, max_iter =
 ↪  10000, epsilon = 0.5, tau = 0.8, init_step_size = 1)
print(minimize_backtrack)
```

```
$betas
 [1] 3.7571481 2.0808951 1.7535339 1.5426377 1.3887564 1.2687146 1.1709946
 [8] 1.0890410 1.0187765 0.9574987 0.9033291 0.8549116 0.8112373 0.7715364
[15] 0.7352094 0.7017805 0.6708666 0.6421547 0.6153861 0.5903448 0.5668485
[22] 0.5447423 0.5238933 0.5041868 0.4855230 0.4678148 0.4509856 0.4349676
[29] 0.4197007 0.4051313 0.3912114 0.3778977 0.3651513 0.3529370 0.3412225
[36] 0.3299788 0.3191791 0.3087989 0.2988156 0.2892087 0.2799588 0.2710482
[43] 0.2624606 0.2541805 0.2461937 0.2384869 0.2310477 0.2238643 0.2169259
[50] 0.2102221
 [ reached 'max' / getOption("max.print") -- omitted 9950 entries ]

$obj_values
 [1] 228.498357  28.410229  16.604656  11.422582   8.576958   6.810204
 [7]   5.622724   4.778641   4.153059   3.674137   3.297869   2.995924
[13]   2.749315   2.544881   2.373241   2.227544   2.102680   1.994768
[19]   1.900814   1.818471   1.745879   1.681546   1.624259   1.573028
[25]   1.527035   1.485597   1.448143   1.414189   1.383326   1.355202
[31]   1.329516   1.306007   1.284449   1.264645   1.246422   1.229628
[37]   1.214129   1.199806   1.186554   1.174279   1.162897   1.152332
[43]   1.142516   1.133390   1.124896   1.116987   1.109616   1.102742
[49]   1.096328   1.090340
 [ reached 'max' / getOption("max.print") -- omitted 9950 entries ]

$eta_values
 [1] 1.000000000 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698
 [7] 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698
[13] 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698
[19] 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698
[25] 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698
[31] 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698
[37] 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698
[43] 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698 0.007378698
[49] 0.007378698 0.007378698
```

```
 [ reached 'max' / getOption("max.print") -- omitted 9950 entries ]

$last_iter
[1] 505
```

```
cat("The functions stopped after", minimize_backtrack$last_iter - 1,
 ↪  "iterations \n")
```

```
The functions stopped after 504 iterations
```

```
cat("The function's point of minimization is", "(",
 ↪  minimize_backtrack$betas[minimize_backtrack$last_iter], ",",
 ↪  minimize_backtrack$obj_values[minimize_backtrack$last_iter], ") \n")
```

```
The function's point of minimization is ( 2.470678e-07 , 1 )
```

1) For the constant step size version of gradient descent, discuss how you selected the step size used in your code

Theoretical Analysis proves that for functions with a unique global minimum, the step size should be within 0 to 1/L to converge to the unique global minimum, where L is the Lipchitz constant, given by:

$$||\nabla f(x) - \nabla f(y)||_2 \leq L||x-y||_2$$

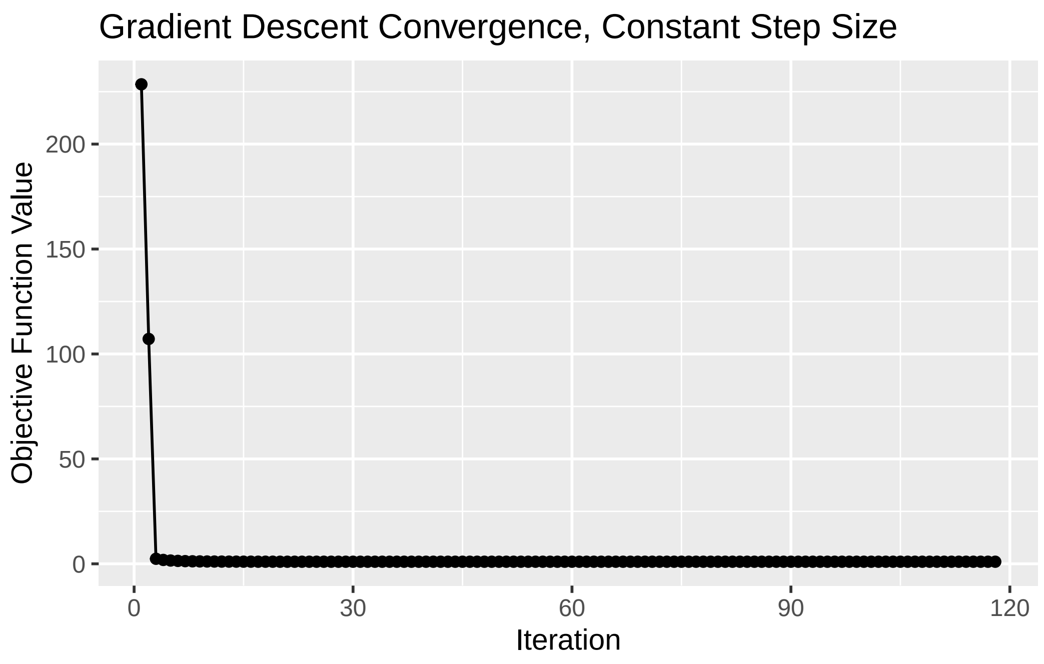
Since this cannot be calculated in practice, usually a small step size of 0.01 is what to begin with. From there, manually fine tuning to try 0.02 and 0.03 is a good idea to see if there's any better iterations. Starting from a big step size is usually unsafe do the algorithm overshooting and diverging instead of converging. Ultimately, the step size of 0.02 seemed best to reduce the number of iterations.

2) For both versions of the gradient descent algorithm, plot the value of $f(x_k)$ as a function of k the number of iterations

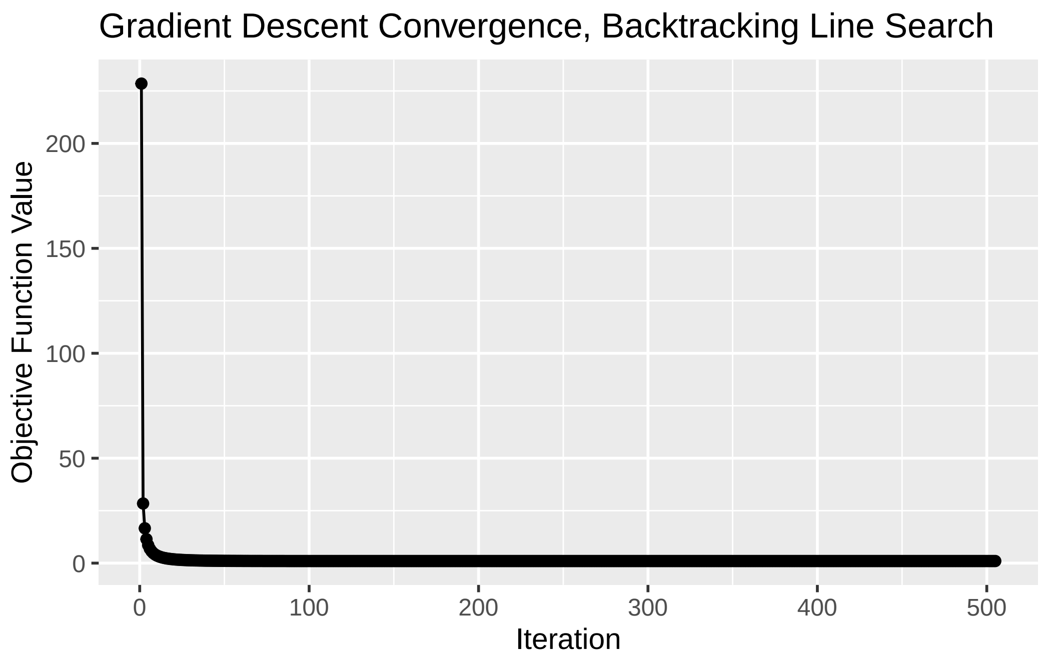
```
# constant step size
iterations <- 1:minimize_constant_step$last_iter
obj_values <- (minimize_constant_step$obj_values)[iterations]
f_k_constant <- cbind(obj_values, iterations)
```

```r
iterations <- 1:minimize_backtrack$last_iter
obj_values <- (minimize_backtrack$obj_values)[iterations]
f_k_backtrack <- cbind(obj_values, iterations)

ggplot(f_k_constant, aes(x=iterations, y=obj_values)) +
  geom_point() +
  geom_line() +
  ggtitle("Gradient Descent Convergence, Constant Step Size") +
  xlab("Iteration") + ylab("Objective Function Value")
```



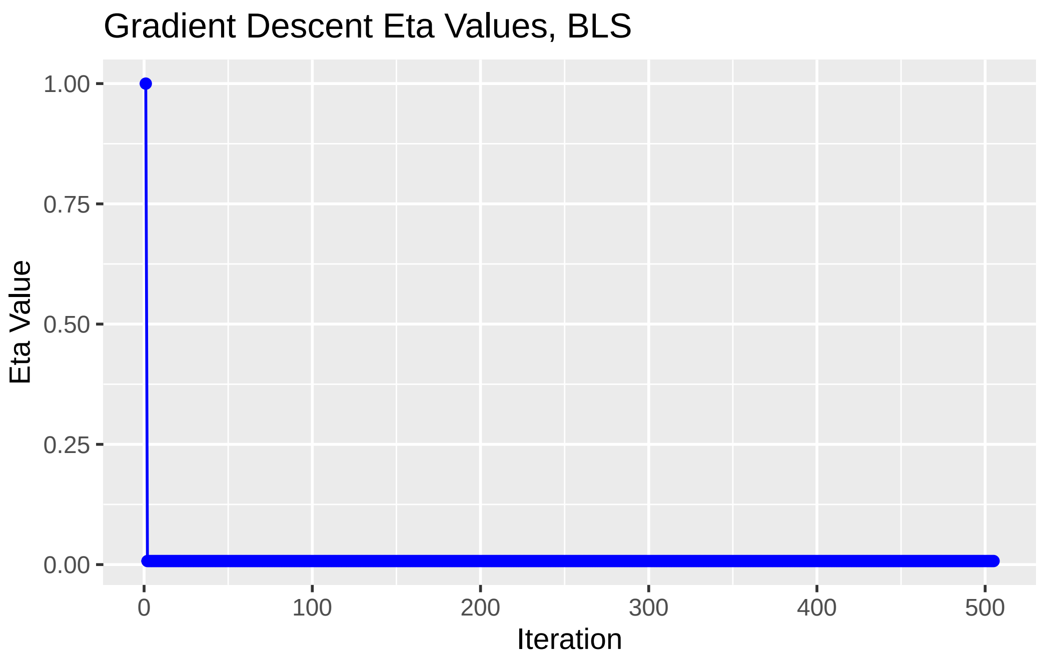
```r
ggplot(f_k_backtrack, aes(x=iterations, y=obj_values)) +
  geom_point() +
  geom_line() +
  ggtitle("Gradient Descent Convergence, Backtracking Line Search") +
  xlab("Iteration") + ylab("Objective Function Value")
```



3) For the the gradient descent method with backtracking line search, plot the step size η_k selected at step k as a function of k. Comment on the result

```r
iterations <- 1:minimize_backtrack$last_iter
eta_values <- minimize_backtrack$eta_values[iterations]
eta_backtrack <- cbind(eta_values, iterations)

ggplot(eta_backtrack, aes(x=iterations, y=eta_values)) +
  geom_point(color = "blue") +
  geom_line(color = "blue") +
  ggtitle("Gradient Descent Eta Values, BLS") +
  xlab("Iteration") + ylab("Eta Value")
```



We can see that the step size was initially 0.02, but at the very first few iterations the Armijo condition immediately reduced the step size to a very small number < 0.005 in order to prevent overshooting. This condition held for the rest of the iterations until the algorithm converged eventually, after around 443 iterations. Compared to the constant step size gradient descent, the step size was much smaller for all the iterations, meaning that it converged with > 300 more iterations than the constant step size gradient descent. Although using a large step size like 0.02 would be much faster, it seems like the step sizes were chosen to be a safer bound so that the algorithm would not overshoot

Problem 2

To understand the sensitivity of the gradient descent algorithm and its variants to the "shape" of the function, the two data sets provided (dataset1.csv, dataset2.csv) will be used

They contain 100 observations for a response y and 20 predictors $x_j, j = 1, \cdots, 20$

Part a

Using the gradient descent code provided (both in R and Python) obtain the estimates of the regression coefficient, using both a constant step size and backtracking line search.

Read in the data and the gradient descent function:

```r
dataset1 <- read_csv("dataset1.csv")
```

```
Rows: 100 Columns: 21
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (21): Y, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12, X13, X14, X1...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
dataset2 <- read_csv("dataset2.csv")
```

```
Rows: 100 Columns: 21
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (21): Y, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12, X13, X14, X1...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
# gradient descent given in class
gradient_descent_class <- function(X, y, eta = NULL, tol = 1e-6, max_iter =
↪ 10000, backtracking = TRUE, epsilon = 0.5, tau = 0.8) {
  # Initialize
  n <- nrow(X)
  p <- ncol(X)
  beta <- rep(0, p)
  obj_values <- numeric(max_iter)
  eta_values <- numeric(max_iter)  # To store eta values used each iteration
  eta_bt <- 1  # Initial step size for backtracking

  # Objective function: Mean Squared Error (MSE)
  obj_function <- function(beta) {
    sum((X %*% beta - y)^2) / (2 * n)
  }

  # Gradient function
  gradient <- function(beta) {
    t(X) %*% (X %*% beta - y) / n
  }

  for (iter in 1:max_iter) {
    grad <- gradient(beta)

    if (backtracking) {
      if (iter == 1) eta_bt <- 1  # Reset only in the first iteration
      beta_new <- beta - eta_bt * grad

      while (obj_function(beta_new) > obj_function(beta) - epsilon * eta_bt *
       ↪  sum(grad^2)) {
        eta_bt <- tau * eta_bt
        beta_new <- beta - eta_bt * grad
      }
      eta_used <- eta_bt
    } else {
      if (is.null(eta)) stop("When backtracking is FALSE, a fixed eta must be
       ↪  provided.")
      beta_new <- beta - eta * grad
      eta_used <- eta
    }

    eta_values[iter] <- eta_used

    obj_values[iter] <- obj_function(beta_new)

    if (sqrt(sum((beta_new - beta)^2)) < tol) {
      obj_values <- obj_values[1:iter]
      eta_values <- eta_values[1:iter]
      break
    }

    beta <- beta_new
  }

  return(list(beta = beta, obj_values = obj_values, eta_values = eta_values))
}
```

```r
X_dataset_1 <- dataset1 %>%
  select(-Y) %>%
  as.matrix()
```

```r
y_dataset_1 <- dataset1 %>%
  select(Y) %>%
  as.matrix()
X_dataset_2 <- dataset2 %>%
  select(-Y) %>%
  as.matrix()
y_dataset_2 <- dataset2 %>%
  select(Y) %>%
  as.matrix()

reg_const_dataset_1 <- gradient_descent_class(X_dataset_1, y_dataset_1,
 ↪  backtracking = FALSE, eta = 5)

cat("Constant Step Size: dataset1 \n")
```

```
Constant Step Size: dataset1
```

```r
print("Beta Values:")
```

```
[1] "Beta Values:"
```

```r
print(reg_const_dataset_1$beta)
```

```
               Y
X1   0.200284535
X2   0.071172300
X3   1.136549069
X4   0.143783956
X5  -0.007077920
X6  -0.012022200
X7   0.467646270
X8   0.038900577
X9   0.375594201
X10  1.193204076
X11  0.234086463
X12  1.001887238
X13  0.008026558
X14  0.647194983
X15 -0.075084887
```

```
X16  0.804818808
X17  1.021098020
X18  0.129799690
X19 -0.273712644
X20  0.401570242
```

```r
print("Obj Function Values:")
```

```
[1] "Obj Function Values:"
```

```r
print(reg_const_dataset_1$obj_values)
```

```
 [1] 0.4321102 0.3895214 0.3732853 0.3653793 0.3609190 0.3581722 0.3563922
 [8] 0.3552028 0.3543925 0.3538331 0.3534435 0.3531702 0.3529775 0.3528412
[15] 0.3527443 0.3526753 0.3526261 0.3525909 0.3525657 0.3525476 0.3525346
[22] 0.3525253 0.3525185 0.3525137 0.3525102 0.3525077 0.3525059 0.3525045
[29] 0.3525036 0.3525029 0.3525024 0.3525020 0.3525018 0.3525016 0.3525015
[36] 0.3525014 0.3525013 0.3525012 0.3525012 0.3525012 0.3525011 0.3525011
[43] 0.3525011 0.3525011 0.3525011 0.3525011 0.3525011 0.3525011 0.3525011
[50] 0.3525011
 [ reached 'max' / getOption("max.print") -- omitted 28 entries ]
```

```r
print("Eta Values:")
```

```
[1] "Eta Values:"
```

```r
print(reg_const_dataset_1$eta_values)
```

```
 [1] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
[39] 5 5 5 5 5 5 5 5 5 5 5 5
 [ reached 'max' / getOption("max.print") -- omitted 28 entries ]
```

```r
cat("The functions stopped after",
↪  max(which(!is.na(reg_const_dataset_1$eta_values))), "iterations \n \n")
```

```
The functions stopped after 78 iterations
```

```r
reg_const_dataset_2 <- gradient_descent_class(X_dataset_2, y_dataset_2,
 ↪  backtracking = FALSE, eta = 0.02)

cat("Constant Step Size: dataset2 \n")
```

```
Constant Step Size: dataset2
```

```r
print("Beta Values:")
```

```
[1] "Beta Values:"
```

```r
print(reg_const_dataset_2$beta)
```

```
             Y
X1   0.2404371
X2   0.2959701
X3   0.5315576
X4   0.2003784
X5   0.4659575
X6   0.1774883
X7   0.3343755
X8  -0.0174628
X9   0.1653269
X10  0.7304544
X11  0.2279370
X12  0.3597918
X13  0.1830929
X14  0.2943043
X15 -0.1367969
X16  0.3184976
X17  0.4370496
X18  0.4642748
X19  0.2362962
X20  0.2229727
```

```r
print("Obj Function Values:")
```

```
[1] "Obj Function Values:"
```

```r
print(reg_const_dataset_2$obj_values)
```

```
 [1] 4.7108920 2.0434136 1.3308690 0.9980908 0.8045051 0.6817554 0.6001823
 [8] 0.5440527 0.5042603 0.4752878 0.4536824 0.4372228 0.4244423 0.4143483
[15] 0.4062534 0.3996715 0.3942518 0.3897372 0.3859359 0.3827035 0.3799294
[22] 0.3775285 0.3754344 0.3735948 0.3719685 0.3705223 0.3692296 0.3680689
[29] 0.3670223 0.3660753 0.3652158 0.3644335 0.3637198 0.3630673 0.3624698
[36] 0.3619217 0.3614183 0.3609555 0.3605295 0.3601371 0.3597753 0.3594416
[43] 0.3591335 0.3588490 0.3585862 0.3583432 0.3581185 0.3579107 0.3577184
[50] 0.3575404
 [ reached 'max' / getOption("max.print") -- omitted 8109 entries ]
```

```r
print("Eta Values:")
```

```
[1] "Eta Values:"
```

```r
print(reg_const_dataset_2$eta_values)
```

```
 [1] 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
[16] 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
[31] 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
[46] 0.02 0.02 0.02 0.02 0.02
 [ reached 'max' / getOption("max.print") -- omitted 8109 entries ]
```

```r
cat("The functions stopped after",
 ↪  max(which(!is.na(reg_const_dataset_2$eta_values))), "iterations \n \n")
```

```
The functions stopped after 8159 iterations
```

```r
reg_bls_data1 <- gradient_descent_class(X_dataset_1, y_dataset_1, eta = NULL,
 ↪  tol = 1e-6, max_iter = 10000, backtracking = TRUE, epsilon = 0.5, tau =
 ↪  0.8)

cat("BLS: dataset1 \n")
```

```
BLS: dataset1
```

```r
print("Beta Values:")
```

```
[1] "Beta Values:"
```

```r
print(reg_bls_data1$beta)
```

```
               Y
X1   0.200288160
X2   0.071168997
X3   1.136538577
X4   0.143781908
X5  -0.007076641
X6  -0.012017800
X7   0.467649231
X8   0.038900395
X9   0.375589135
X10  1.193194040
X11  0.234088137
X12  1.001888404
X13  0.008031824
X14  0.647189722
X15 -0.075073783
X16  0.804823660
X17  1.021092124
X18  0.129789683
X19 -0.273703280
X20  0.401566025
```

```r
print("Obj Function Values:")
```

```
[1] "Obj Function Values:"
```

```r
print(reg_bls_data1$obj_values)
```

```
 [1] 0.5908489 0.5345036 0.4950110 0.4663575 0.4449979 0.4287255 0.4161031
 [8] 0.4061594 0.3982185 0.3917990 0.3865514 0.3822180 0.3786058 0.3755688
[15] 0.3729947 0.3707970 0.3689075 0.3672727 0.3658498 0.3646045 0.3635090
[22] 0.3625407 0.3616809 0.3609143 0.3602282 0.3596118 0.3590562 0.3585539
```

```
[29] 0.3580984 0.3576843 0.3573068 0.3569619 0.3566461 0.3563564 0.3560902
[36] 0.3558451 0.3556190 0.3554103 0.3552173 0.3550386 0.3548729 0.3547192
[43] 0.3545764 0.3544437 0.3543201 0.3542051 0.3540979 0.3539980 0.3539047
[50] 0.3538176
 [ reached 'max' / getOption("max.print") -- omitted 305 entries ]
```

```
print("Eta Values:")
```

```
[1] "Eta Values:"
```

```
print(reg_bls_data1$eta_values)
```

```
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[39] 1 1 1 1 1 1 1 1 1 1 1 1
 [ reached 'max' / getOption("max.print") -- omitted 305 entries ]
```

```
cat("The functions stopped after",
↪  max(which(!is.na(reg_bls_data1$eta_values))), "iterations \n \n")
```

```
The functions stopped after 355 iterations
```

```
reg_bls_data2 <- gradient_descent_class(X_dataset_2, y_dataset_2, eta = NULL,
↪  tol = 1e-6, max_iter = 10000, backtracking = TRUE, epsilon = 0.5, tau =
↪  0.8)

cat("BLS: dataset2 \n")
```

```
BLS: dataset2
```

```
print("Beta Values:")
```

```
[1] "Beta Values:"
```

```
print(reg_bls_data2$beta)
```

```
               Y
X1   0.24043923
X2   0.29598085
X3   0.53158006
X4   0.20040437
X5   0.46595899
X6   0.17748224
X7   0.33432770
X8  -0.01747586
X9   0.16533621
X10  0.73052099
X11  0.22793299
X12  0.35975400
X13  0.18307123
X14  0.29433152
X15 -0.13688470
X16  0.31846923
X17  0.43704902
X18  0.46434151
X19  0.23625194
X20  0.22297328
```

```
print("Obj Function Values:")
```

```
[1] "Obj Function Values:"
```

```
print(reg_bls_data2$obj_values)
```

```
 [1] 3.9233028 1.7852238 1.1730966 0.8802603 0.7137618 0.6115344 0.5454485
 [8] 0.5008908 0.4697418 0.4472740 0.4306231 0.4179897 0.4082051 0.4004876
[15] 0.3943000 0.3892646 0.3851109 0.3816415 0.3787103 0.3762080 0.3740515
[22] 0.3721774 0.3705363 0.3690897 0.3678071 0.3666643 0.3656416 0.3647229
[29] 0.3638952 0.3631472 0.3624700 0.3618554 0.3612969 0.3607887 0.3603255
[36] 0.3599031 0.3595175 0.3591653 0.3588432 0.3585486 0.3582791 0.3580323
[43] 0.3578062 0.3575990 0.3574091 0.3572350 0.3570753 0.3569287 0.3567942
[50] 0.3566707
 [ reached 'max' / getOption("max.print") -- omitted 7348 entries ]
```

```
print("Eta Values:")
```

```
[1] "Eta Values:"
```

```r
print(reg_bls_data2$eta_values)
```

```
 [1] 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518
 [9] 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518
[17] 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518
[25] 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518
[33] 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518
[41] 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518 0.022518
[49] 0.022518 0.022518
 [ reached 'max' / getOption("max.print") -- omitted 7348 entries ]
```

```r
cat("The functions stopped after",
 ↪  max(which(!is.na(reg_bls_data2$eta_values))), "iterations \n \n")
```

```
The functions stopped after 7398 iterations
```

1) Discuss how you selected the constant step size. Also, discuss which convergence criterion you used and the tolerance parameter used

Because we cannot find the Lipchitz constant, constant step size is tuned manually. For data set 1, the step size initially was set small = 0.01, but it did not converge, so I increased the step size manually until it converged at step size = 1. From there, I pushed the step size until the function would not converge anymore. The max step size I could use without 5. For data set 2, I started with 0.01 step size, and eventually a step size of 0.02 was chosen in order to make the algorithm converge in 8159 iterations. For the tolerance parameter, a tolerance of $1e^{-}6$ was used, which is a standard tolerance parameter and is very close approximation to the real solution. For the convergence criterion, the stop criteria given in the code was to terminate when the difference of beta between iterations is less than the tolerance, indicating that the function has converged.

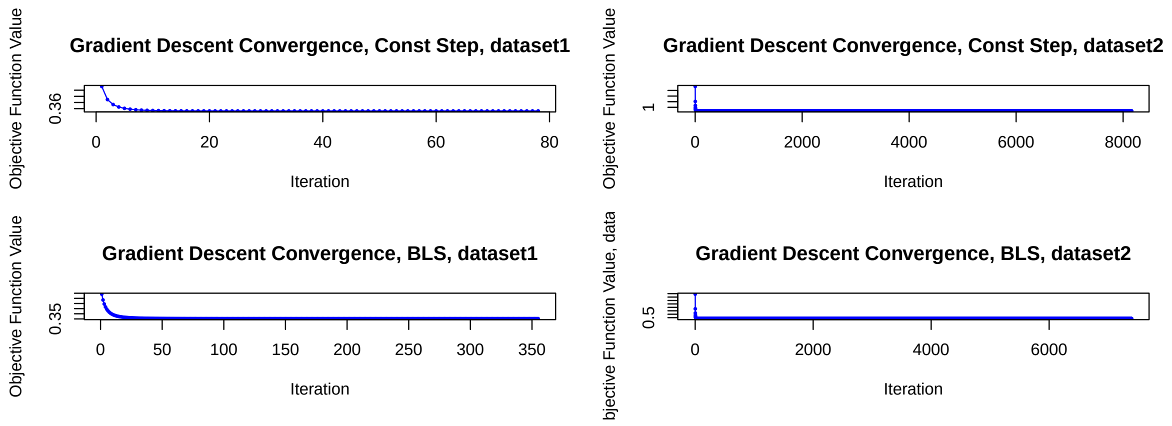
2) Compare the results with those obtained from the lm command in R or from the class LinearRegression from the sklearn.linear model in Python.

Specifically, calculate $\| \widehat{\beta_{GD}} - \hat{\beta} \|_2$, where $\widehat{\beta_{GD}}$ is the estimate of the regression coefficient obtained from the gradient descent algorithm (both with constant step size and backtracking line search) and $\hat{\beta}$ obtained from the least squares solution implemented in R or Python

3) Plot the value of the objective function as a function of the number of iterations required

```r
par(mfrow=c(2,2))

plot(reg_const_dataset_1$obj_values, type = "o", col = "blue", pch = 16, cex
 ↪  = 0.5,
     xlab = "Iteration", ylab = "Objective Function Value",
     main = "Gradient Descent Convergence, Const Step, dataset1")
plot(reg_const_dataset_2$obj_values, type = "o", col = "blue", pch = 16, cex
 ↪  = 0.5,
     xlab = "Iteration", ylab = "Objective Function Value",
     main = "Gradient Descent Convergence, Const Step, dataset2")
plot(reg_bls_data1$obj_values, type = "o", col = "blue", pch = 16, cex = 0.5,
     xlab = "Iteration", ylab = "Objective Function Value",
     main = "Gradient Descent Convergence, BLS, dataset1")
plot(reg_bls_data2$obj_values, type = "o", col = "blue", pch = 16, cex = 0.5,
     xlab = "Iteration", ylab = "Objective Function Value, dataset2",
     main = "Gradient Descent Convergence, BLS, dataset2")
```



Part b

Implement the Polyak and Nesterov momentum methods and obtain the estimates of the regression coefficients, using both a constant step size and backtracking line search