HW 1

Bryan Mui - UID 506021334 - 14 April 2025

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)

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

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

$$\begin{align} f'(x) &= 4x^3 + 4x \tag{5} \\ 0 &= 4x^3 + 4x \tag{6} \\ 0 &= 4x(x^2 + 4) \tag{7} \end{align}$$

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,

$$f''(0) = 12(0)^2 + 4 \quad (8)$$
$$= 4 \quad (9)$$

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 $||\nabla(f(x_k)||_2 \leq \epsilon$

```
# 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
```

```r
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 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 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 getOption("max.print") -- omitted 9950 entries ]

$last_iter
[1] 118
```

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

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

```r
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 = 0.01) {
  # 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
calc_armijo_stepsize <- function(beta_new, beta, eta) {
  iter <- 1 # set a hard limit of 10000 iterations
  while (obj_function(beta_new) > (obj_function(beta) - (epsilon * eta *
   ↪  gradient(beta)^2)) && iter <= max_iter) {
    eta <- tau * eta
    iter <- iter + 1
  }
  return(eta)
}

# 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

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

  # Select eta(step size) for next step(k+1) using backtracking line search
```

```
    eta <- calc_armijo_stepsize(beta_new = beta_new, beta = beta, eta = eta)
    eta_values[iter + 1] <- eta

    # 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 = 0.02)
print(minimize_backtrack)
```

```
$betas
 [1]  3.7571481 -0.7863447 -0.7446451 -0.7067145 -0.6719909 -0.6400276
 [7] -0.6104641 -0.5830058 -0.5574085 -0.5334683 -0.5110128 -0.4898952
[13] -0.4699897 -0.4511872 -0.4333931 -0.4165242 -0.4005076 -0.3852786
[19] -0.3707798 -0.3569597 -0.3437725 -0.3311765 -0.3191343 -0.3076118
[25] -0.2965782 -0.2860051 -0.2758667 -0.2661392 -0.2568006 -0.2478308
[31] -0.2392111 -0.2309241 -0.2229537 -0.2152848 -0.2079034 -0.2007963
[37] -0.1939514 -0.1873569 -0.1810021 -0.1748767 -0.1689711 -0.1632761
[43] -0.1577833 -0.1524843 -0.1473715 -0.1424376 -0.1376755 -0.1330786
[49] -0.1286407 -0.1243556
```

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

$obj_values
 [1] 228.498357   2.619018   2.416459   2.248337   2.107061   1.987072
 [7]   1.884213   1.795321   1.717946   1.650167   1.590459   1.537593
[13]   1.490573   1.448581   1.410939   1.377084   1.346543   1.318914
[19]   1.293855   1.271076   1.250325   1.231385   1.214066   1.198204
[25]   1.183654   1.170289   1.157996   1.146677   1.136242   1.126613
[31]   1.117718   1.109496   1.101888   1.094843   1.088316   1.082264
[37]   1.076649   1.071437   1.066597   1.062099   1.057918   1.054029
[43]   1.050411   1.047044   1.043908   1.040989   1.038268   1.035733
[49]   1.033371   1.031168
 [ reached getOption("max.print") -- omitted 9950 entries ]

$eta_values
 [1] 0.020000 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192
 [9] 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192
[17] 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192
[25] 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192
[33] 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192
[41] 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192 0.008192
[49] 0.008192 0.008192
 [ reached getOption("max.print") -- omitted 9950 entries ]

$last_iter
[1] 444
```

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

```
The functions stopped after 443 iterations
```

```r
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 ( -2.456074e-07 , 1 )
```

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

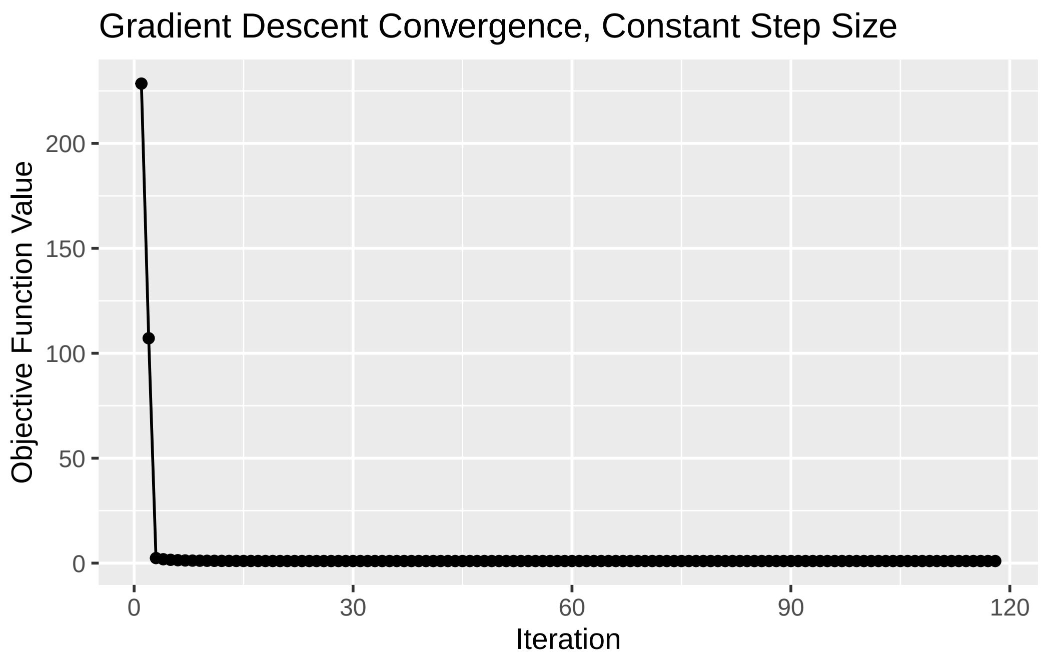
Theoretical Analysis proves that for functions with a unique global minimum, the step size should be within 0 to 1/L to converge to the unique global minimum, where L is the Lipchitz constant, given by:

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

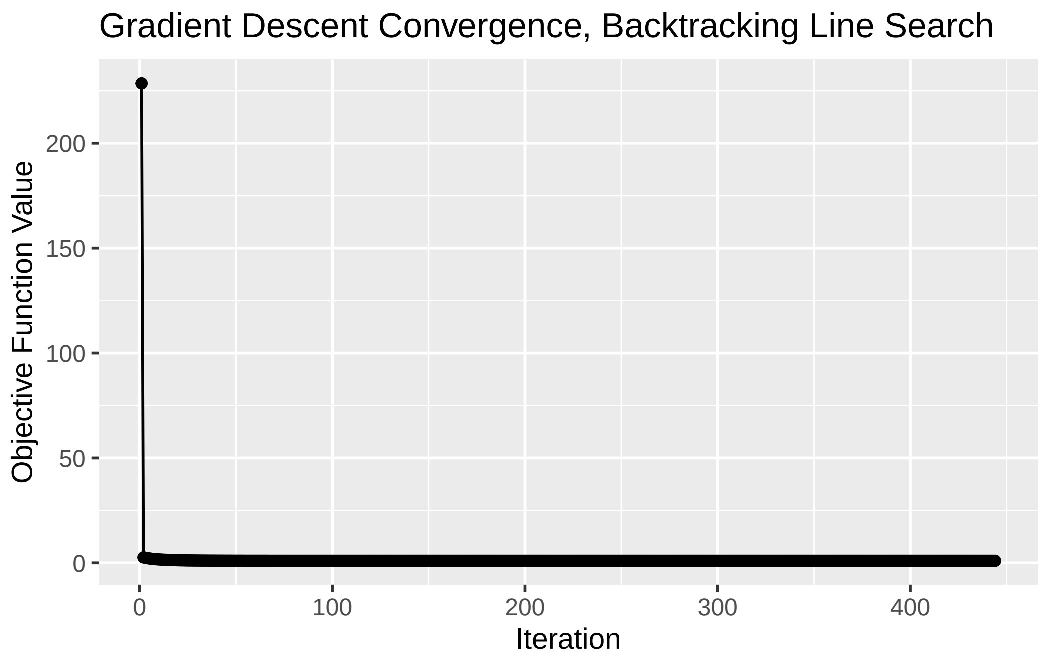
Since this cannot be calculated in practice, usually a small step size of 0.01 is what to begin with. From there, manually fine tuning to try 0.02 and 0.03 is a good idea to see if there's any better iterations. Starting from a big step size is usually unsafe do the algorithm overshooting and diverging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

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

par(mfrow=c(1, 2))

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")
```

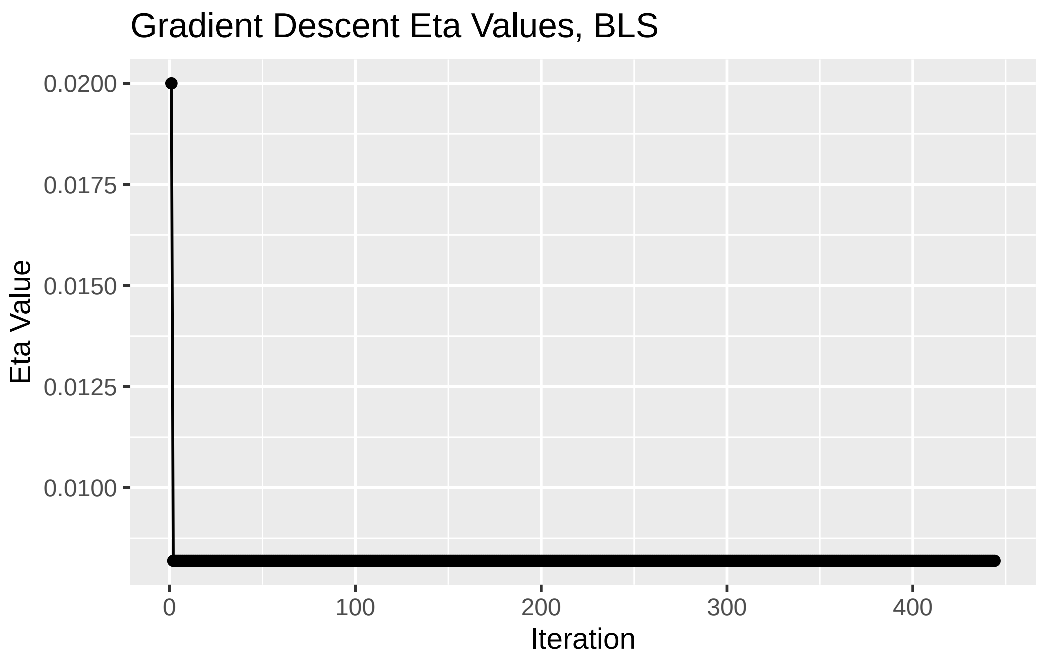


```r
ggplot(f_k_backtrack, aes(x=iterations, y=obj_values)) +
  geom_point() +
  geom_line() +
  ggtitle("Gradient Descent Convergence, Backtracking Line Search") +
  xlab("Iteration") + ylab("Objective Function Value")
```



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

```
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() +
  geom_line() +
  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