Stats 102B HW5

Tori Wang

2023-05-28

Problem 1

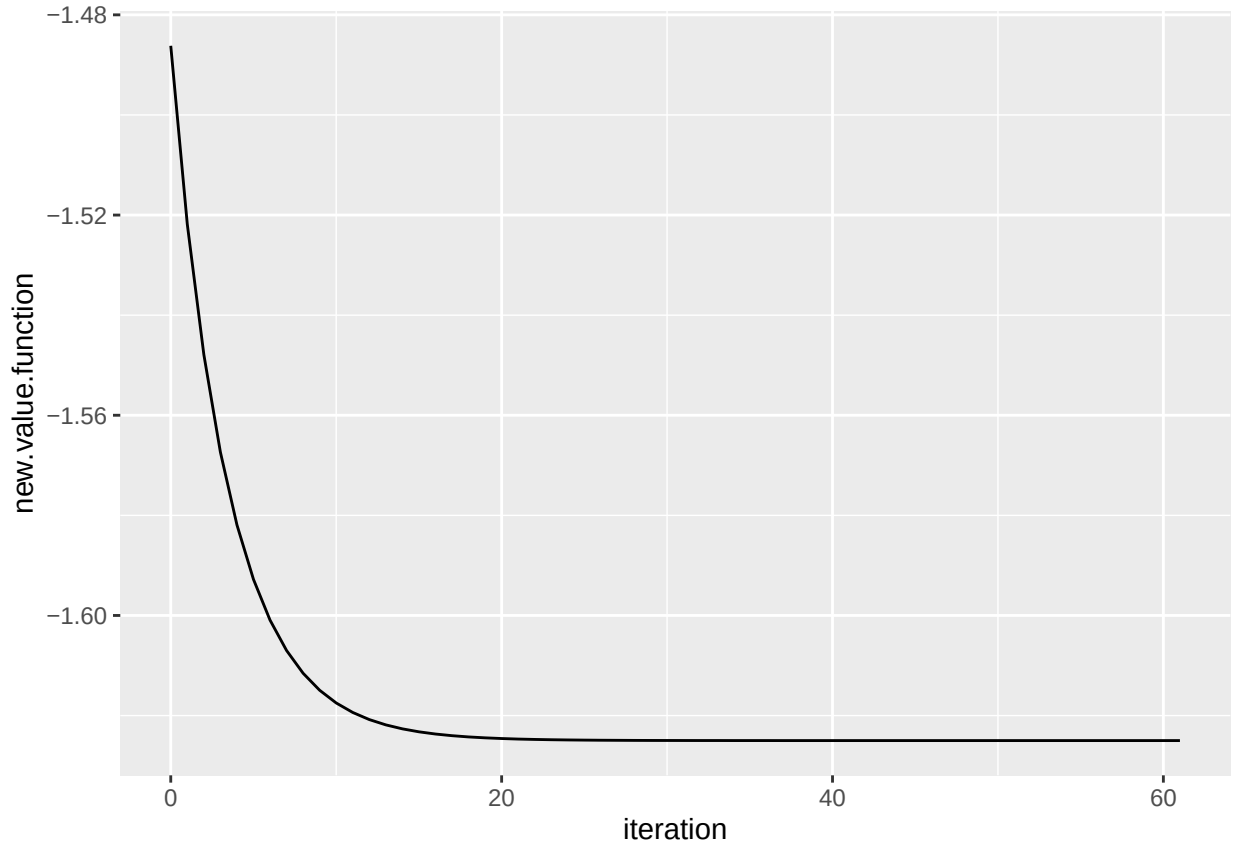
Part (a)

Using the provided code for gradient descent with backtracking line search and obtain the optimal value.

```
#define objective function to be optimized
objective.function = function(x) {
            4*x[1]^2 + 2*x[2]^2 + 4*x[1]*x[2] + 5*x[1] + 2*x[2]
    }

# define the derivative for a 2D objective function
derivative = function(x) {
  c(8*x[1]+4*x[2] + 5, 4*x[2]+4*x[1] + 2)
}

# here we define the epsilon parameter for the backtracking line search
myepsilon = 0.5
# here we define the tau parameter for the backtracking line search
mytau = 0.9
# initial guess for stepsize
stepsize_0 = 1
# here we define the tolerance of the convergence criterion
mytol = 0.000000001
# starting point
mystartpoint = c(1,2)

gradientDesc = function(obj.function, startpoint,
                        stepsize, tau, epsilon, conv_threshold, max_iter) {

  old.point = startpoint

  stepsize=stepsize_0
  gradient = derivative(old.point)


  # here we check how to pick the stepsize at iteration 0

  while (objective.function(old.point - stepsize * gradient) >
         objective.function(old.point) - (epsilon * stepsize *
                                t(gradient) %*% gradient) ){
    stepsize = tau * stepsize
  }
```

```
new.point = c(old.point[1] - stepsize*gradient[1],
              old.point[2] - stepsize*gradient[2])


old.value.function = obj.function(new.point)

converged = F
iterations = 0

while(converged == F) {
  ## Implement the gradient descent algorithm
  old.point = new.point

  gradient = derivative(old.point)

  # here we check how to pick the stepsize at iteration k

  while (objective.function(old.point - stepsize * gradient) >
         objective.function(old.point) - (epsilon * stepsize *
                                            t(gradient) %*% gradient) ){
    stepsize = tau * stepsize
  }

  new.point = c(old.point[1] - stepsize*gradient[1],
                old.point[2] - stepsize*gradient[2])

  new.value.function = obj.function(new.point)


  if( abs(old.value.function - new.value.function) <= conv_threshold) {
    converged = T
  }

  data.output = data.frame(iteration = iterations,
                      old.value.function = old.value.function,
                      new.value.function = new.value.function,
                      old.point=old.point, new.point=new.point,
                      stepsize = stepsize
                      )

  if(exists("iters")) {
    iters <- rbind(iters, data.output)
  } else {
    iters = data.output
  }

  iterations = iterations + 1
  old.value.function = new.value.function

  if(iterations >= max_iter) break
}
return(list(converged = converged,
            num_iterations = iterations,
```

```
              old.value.function = old.value.function,
              new.value.function = new.value.function,
              coefs = new.point,
              stepsize = stepsize,
              iters = iters))
}


results = gradientDesc(objective.function, mystartpoint, stepsize_0,
                       mytau, myepsilon, mytol, 30000)

library(ggplot2)
ggplot(data = results$iters, mapping = aes(x = iteration, y = new.value.function))+
  geom_line()
```




```
print(results$num_iterations)
```

```
## [1] 62
```

```
print(results$coefs)
```

```
## [1] -0.7500312  0.2500505
```

Part (b): Comment on the accuracy of the calculated minimum and compare it to the results obtained for the different choices of the constant step size used in HW 4.

Also, comment on the number of iterations required based on the backtracking line search algorithm for selecting the step size, compared to the number of iterations needed for the three choices of the constant step size used in HW 4.

The number of iterations found using the backtracking line search was 62, and the resulting coefficients are: [-0.7500312, 0.2500505]. Compared to the results for different step sizes in HW4, the minimum found for tolerance: 1e-08 and step size: 1e-01, as [-0.7500886, 0.2501433]. We can observe that the resulting coefficients for the backtracking line search are more accurate than the minimum found using the normal gradient descent.

As recorded in the last hw, we notice that the number of iterations decreases as the step size increases. Thus the number of iterations and the step size have an inverse relationship function. In the backtracking line search, we use a step size of 1 and the number of iterations required is 62. The number of iterations required for gradient descent 1e-08 and step size: 1e-01 was 48. Thus we find that the number of iterations required for the backtracking line search is actually larger.

Problem 2

Consider the function:

$$f(x) = x^4$$

Part (a): Obtain the theoretical minimum of function f(x). Show your work (recall to argue that it is actually a minimum).

We take the derivative

$$f'(x) = 4x^3$$

Setting this equal to 0, we find

$$4x^3 = 0, x = 0$$

The second derivative is:

$$12x^2 \geq 0$$

for all $x \geq 0$

Since the second derivative is always positive, we find the $x = 0$ is indeed a global minimum.

Part (b): Use the gradient descent algorithm with constant step size and with back- tracking line search to calculate ^xmin.

Constant Step size:

```
#define objective function to be optimized
objective.function = function(x) {
            x^4
    }
```

```r
# define the derivative for a 2D objective function
derivative = function(x) {
  4*x^3
}

# here we define the epsilon parameter for the backtracking line search
myepsilon = 0.5
# here we define the tau parameter for the backtracking line search
mytau = 0.9
# initial guess for stepsize
stepsize_0 = 0.05
# here we define the tolerance of the convergence criterion
mytol = 0.000000001
# starting point
mystartpoint = 1

gradientDesc = function(obj.function, startpoint,
                        stepsize, tau, epsilon, conv_threshold, max_iter) {

  old.point = startpoint
  gradient = derivative(old.point)

  new.point = old.point - stepsize*gradient


  old.value.function = obj.function(new.point)

  converged = F
  iterations = 0

  while(converged == F) {
    ## Implement the gradient descent algorithm
    old.point = new.point

    gradient = derivative(old.point)

    new.point = old.point - stepsize*gradient

    new.value.function = obj.function(new.point)

    if( abs(old.value.function - new.value.function) <= conv_threshold) {
      converged = T
    }

    data.output = data.frame(iteration = iterations,
                         old.value.function = old.value.function,
                         new.value.function = new.value.function,
                         old.point=old.point, new.point=new.point,
                         stepsize = stepsize
                         )

    if(exists("iters")) {
      iters <- rbind(iters, data.output)
```

```
    } else {
      iters = data.output
    }

    iterations = iterations + 1
    old.value.function = new.value.function

    if(iterations >= max_iter) break
  }
  return(list(converged = converged,
              num_iterations = iterations,
              old.value.function = old.value.function,
              new.value.function = new.value.function,
              coefs = new.point,
              stepsize = stepsize,
              iters = iters))
}


constant_stepsize_results = gradientDesc(objective.function, mystartpoint, stepsize_0,
                        mytau, myepsilon, mytol, 30000)
```

Backtracking line search

```
# here we define the epsilon parameter for the backtracking line search
myepsilon = 0.5
# here we define the tau parameter for the backtracking line search
mytau = 0.9
# initial guess for stepsize
stepsize_0 = 0.5
# here we define the tolerance of the convergence criterion
mytol = 0.000000001
# starting point
mystartpoint = 1

gradientDesc = function(obj.function, startpoint,
                        stepsize, tau, epsilon, conv_threshold, max_iter) {

  old.point = startpoint

  stepsize=stepsize_0
  gradient = derivative(old.point)


  # here we check how to pick the stepsize at iteration 0

  while (objective.function(old.point - stepsize * gradient) >
         objective.function(old.point) - (epsilon * stepsize *
                              t(gradient) %*% gradient) ){
    stepsize = tau * stepsize
  }

  new.point = old.point - stepsize*gradient
```

```
old.value.function = obj.function(new.point)

converged = F
iterations = 0

while(converged == F) {
  ## Implement the gradient descent algorithm
  old.point = new.point

  gradient = derivative(old.point)

  # here we reset the stepsize to the constant and check how to pick the stepsize at iteration k
  stepsize = stepsize_0
  while (objective.function(old.point - stepsize * gradient) >
         objective.function(old.point) - (epsilon * stepsize *
                                            t(gradient) %*% gradient) ){
    stepsize = tau * stepsize
  }

  new.point = old.point[1] - stepsize*gradient

  new.value.function = obj.function(new.point)


  if( abs(old.value.function - new.value.function) <= conv_threshold) {
    converged = T
  }

  data.output = data.frame(iteration = iterations,
                      old.value.function = old.value.function,
                      new.value.function = new.value.function,
                      old.point=old.point, new.point=new.point,
                      stepsize = stepsize
                      )

  if(exists("iters")) {
    iters <- rbind(iters, data.output)
  } else {
    iters = data.output
  }

  iterations = iterations + 1
  old.value.function = new.value.function

  if(iterations >= max_iter) break
}
return(list(converged = converged,
            num_iterations = iterations,
            old.value.function = old.value.function,
            new.value.function = new.value.function,
            coefs = new.point,
            stepsize = stepsize,
```

```
              iters = iters))
}


backtracking_results = gradientDesc(objective.function, mystartpoint, stepsize_0,
                       mytau, myepsilon, mytol, 30000)

print(constant_stepsize_results$num_iterations)
```

```
## [1] 2313
```

```
print(backtracking_results$num_iterations)
```

```
## [1] 496
```

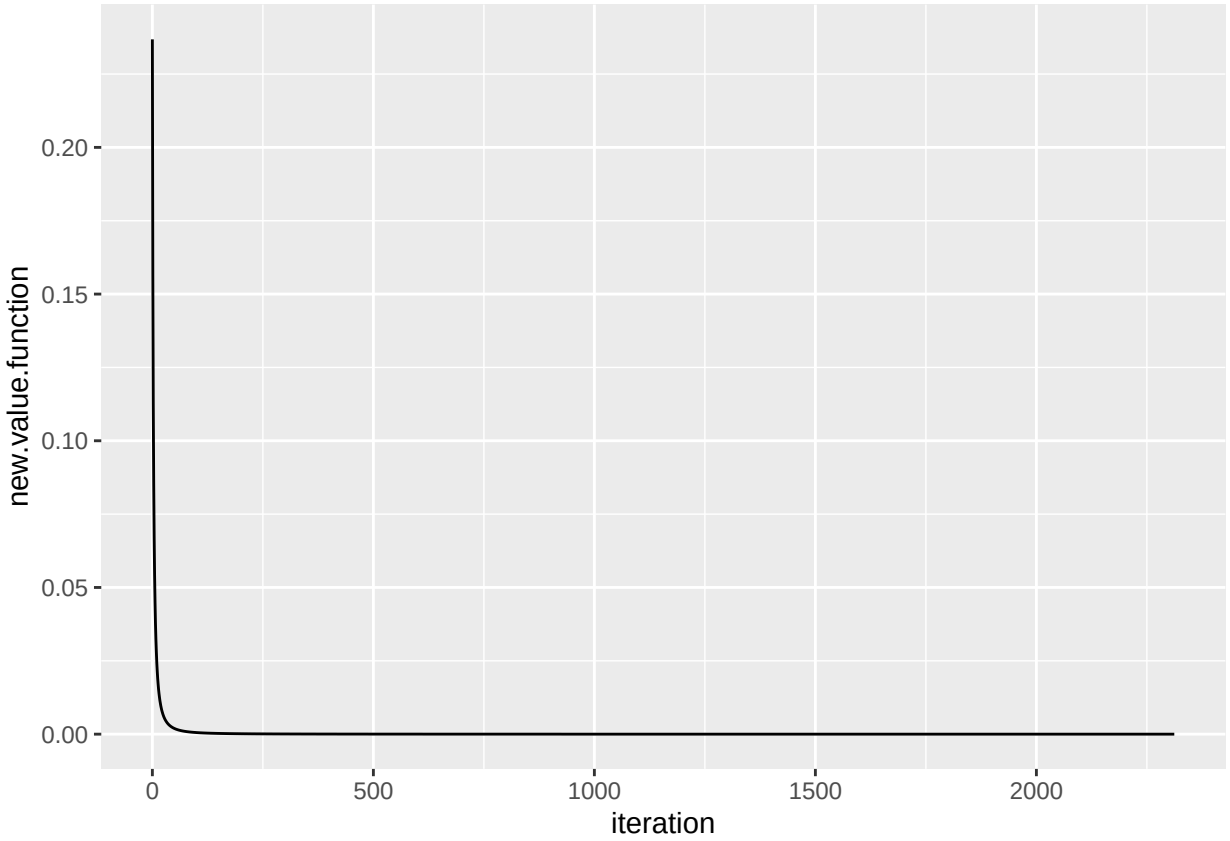
1.

For the constant step size version of gradient descent, discuss how you selected the step size used in your code. For the constant step size version of gradient descent, I decided to select a step size that fell within the range that would be guaranteed convergence based on the Hessian and the chosen first x value. We used 1 as our starting point and it can be observed that for any starting point x_0 thats <=1, we have that the maximum Hessian eigenvalue is 1/12, which 0.05 is less than. By doing this, we guarantee convergence

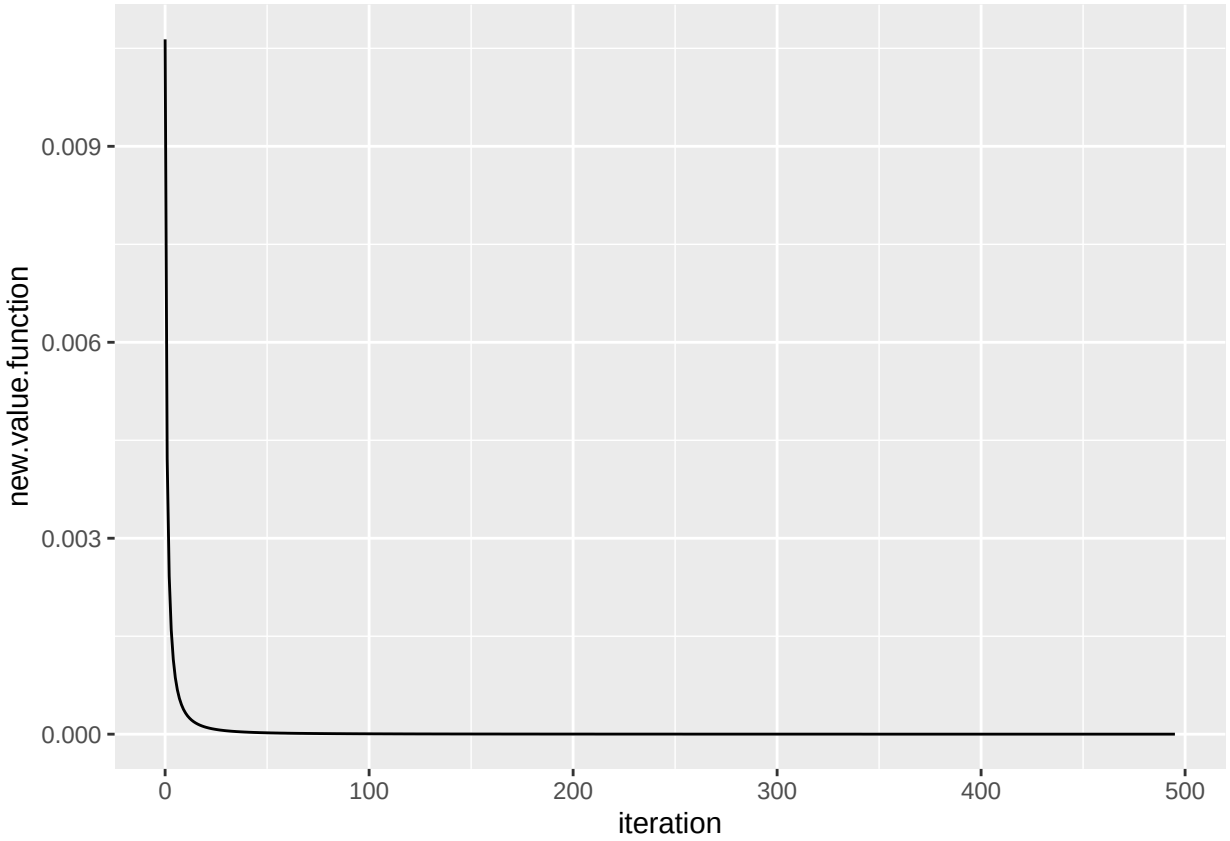
2.

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

```
library(ggplot2)

ggplot(data = constant_stepsize_results$iters, mapping = aes(x = iteration, y = new.value.function))+
  geom_line()
```



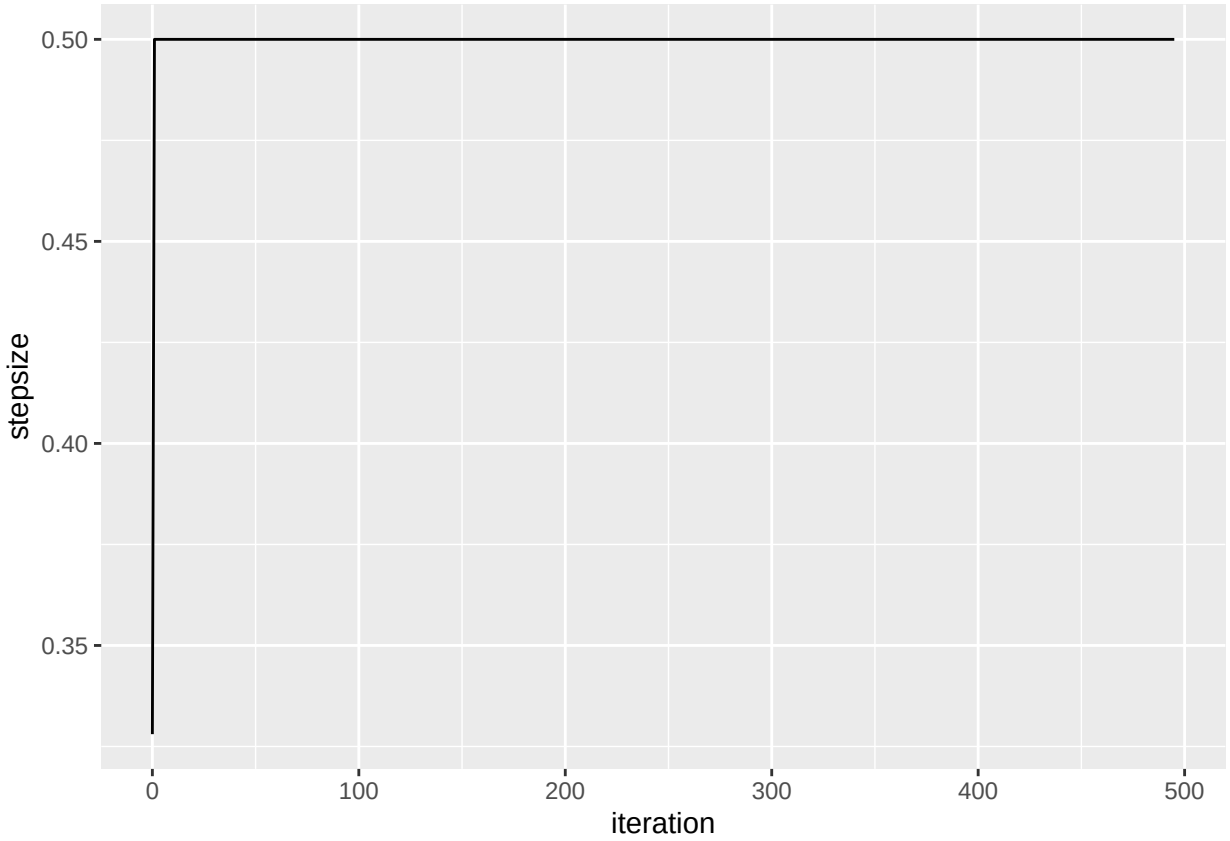

```
ggplot(data = backtracking_results$iters, mapping = aes(x = iteration, y = new.value.function))+
  geom_line()
```



3.

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

```
ggplot(data = backtracking_results$iters, mapping = aes(x = iteration, y = stepsize))+
  geom_line()
```



We can see that the step size decreases for the first iteration and then returns to 0.5. This is because I utilized a backtracking line search where the step size is returned to it's original in each iteration. We can see that in the first iteration, the backtrackign line search changes the step size to a smaller step, and then in the later iterations it stays the same.

Part (c):

The function f(x) has very different curvature (second derivative) in different regions of its domain. It could benefit from adding a momentum term in the gradient descent update.

1.

Implement the momentum adjustment in the gradient descent code with back- tracking line search.

```
# here we define the epsilon parameter for the backtracking line search
myepsilon = 0.5
# here we define the tau parameter for the backtracking line search
mytau = 0.9
# initial guess for stepsize
stepsize_0 = 0.5
# here we define the tolerance of the convergence criterion
mytol = 0.000000001
# starting point
mystartpoint = 1

mymomentum = 0.5

gradientDesc_momentum = function(obj.function, startpoint,
```

```
                      stepsize, tau, epsilon, conv_threshold, max_iter,momentum) {

old.point = startpoint

stepsize=stepsize_0
gradient = derivative(old.point)


# here we check how to pick the stepsize at iteration 0

while (objective.function(old.point - stepsize * gradient) >
       objective.function(old.point) - (epsilon * stepsize *
                           t(gradient) %*% gradient) ){
  stepsize = tau * stepsize
}

new.point = old.point - stepsize*gradient


old.value.function = obj.function(new.point)

converged = F
iterations = 0

while(converged == F) {
  ## Implement the gradient descent algorithm
  older_x_point = old.point
  old.point = new.point

  gradient = derivative(old.point)

  # here we reset the stepsize to the constant and check how to pick the stepsize at iteration k
  stepsize = stepsize_0
  while (objective.function(old.point - stepsize * gradient) >
         objective.function(old.point) - (epsilon * stepsize *
                                            t(gradient) %*% gradient) ){
    stepsize = tau * stepsize
  }

  new.point = old.point - stepsize*gradient + momentum*(old.point- older_x_point)

  new.value.function = obj.function(new.point)


  if( abs(old.value.function - new.value.function) <= conv_threshold) {
    converged = T
  }

  data.output = data.frame(iteration = iterations,
                      old.value.function = old.value.function,
                      new.value.function = new.value.function,
                      old.point=old.point, new.point=new.point,
                      stepsize = stepsize
```

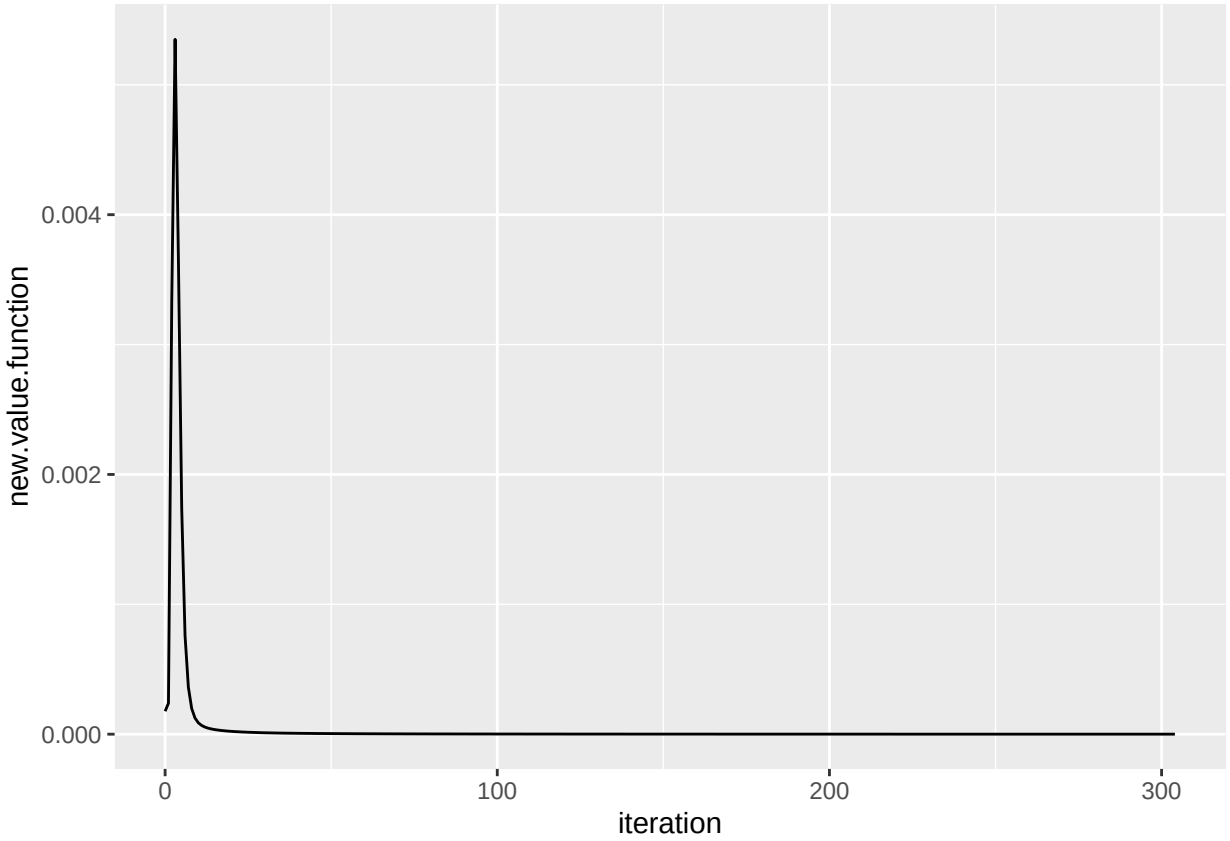
```
                        )

    if(exists("iters")) {
      iters <- rbind(iters, data.output)
    } else {
      iters = data.output
    }

    iterations = iterations + 1
    old.value.function = new.value.function

    if(iterations >= max_iter) break
  }
  return(list(converged = converged,
              num_iterations = iterations,
              old.value.function = old.value.function,
              new.value.function = new.value.function,
              coefs = new.point,
              stepsize = stepsize,
              iters = iters))
}


backtracking_results_momentum = gradientDesc_momentum(objective.function, mystartpoint, stepsize_0,
                        mytau, myepsilon, mytol, 30000, mymomentum)

backtracking_results_momentum$num_iterations
```

```
## [1] 305
```

2.

Plot the value of f(xk) as a function of k the number of iterations.

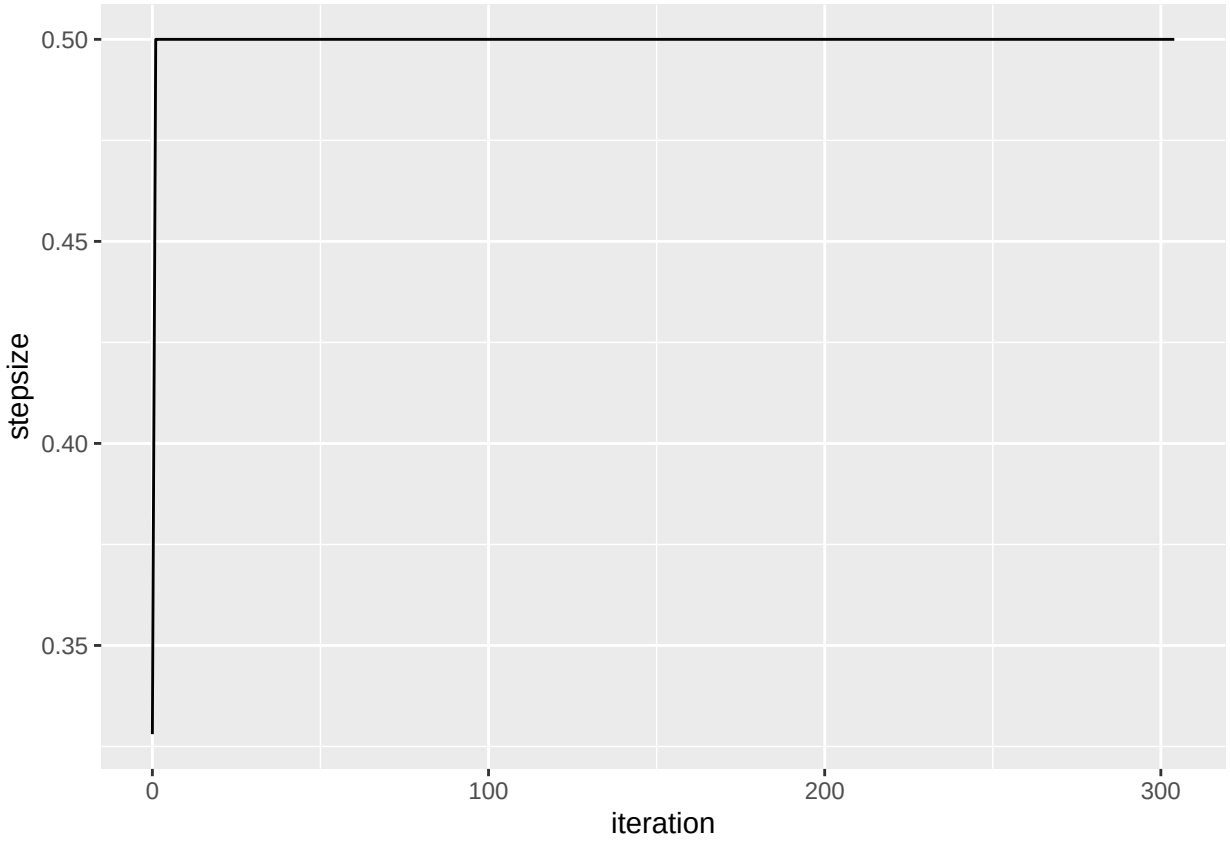
```
ggplot(data = backtracking_results_momentum$iters, mapping = aes(x = iteration, y = new.value.function))
  geom_line()
```



3.

Plot the step size k selected at step k as a function of k. Comment on the result.

```
ggplot(data = backtracking_results_momentum$iters, mapping = aes(x = iteration, y = stepsize))+
  geom_line()
```



We see again that using the step size found during the backtracking line search in which we reset the step size to the origianl in each iteration, has the same pattern as the gradient descent that does not use momentum. Namely, the step size increases back to it's original after the first iteration.

4. Comment on whether the momentum adjustment helps.

We see from the graphs and the number of iterations, that the momentum adjustement does help. The number of iterations for using the same method, backtracking line search, but without momentum adjustment, leads to a greater number of iterations.