Piotr's Assignment: Coninuous Optimization

Sebastian Wolf

20 November 2018

Contents

Step 1: Implementing the gradient descent function:

My answer regarding this step consists of a number of functions as displayed below. * The objective function * The gradient * The inverse of the Hessian (for the implementation of the Newton algorithm)

The gradient descent I implement uses basic backtracking, reducing the stepsize as long as the step to be taken does not reduce the function value.

The auxiliary functions (objective function, gradient, inverse hessian)

```
# Objective function
objective_function <- function(A,x,m,n) {
  (1/2 * t(x-m) %*% A %*% (x-m) - log(diag(x%*%t(x))) %*% matrix(1,n,1))[[1]]
}

# Gradient of objective function
gradient <- function(A,x,m){
  gradient <- (A %*% (x-m) - 2/x)
}

# Inv Hessian of objective function (for Newton method)
inv_hessian <- function(A,x){
  hessian <- (A + diag(2/diag(x%*%t(x))))
  inv_hessian <- solve(hessian)
}
```

The gradient descent function

```
# Gradient descent function
gradient_descent <- function(A,x,m,n,s) {

    # initialise iteration count and dataframe to record path
    count <- 1
    path <- data.frame(t(x))
    path$Z = objective_function(A,x,m,n)
    path$step = s

    # start iterations, continue until gradient vector is close to zero
    while (sum(abs(gradient(A,x,m))) > 0.0001) {

      # Stepsize function (I only use it in gradient descent)
      stepsize <- function(A,x,m,n,s) {
        step <- s
        while (objective_function(A,x - step * gradient(A,x,m),m,n) >
               objective_function(A,x,m,n))
          step = step/2
          if (step<0.0000001) {
            break
          }
        return(step)
      }
```

```
      # take step
      x <- x - stepsize(A,x,m,n,s) * gradient(A,x,m)
      count = count + 1

      # record step taken
      path[count,] = c(t(x),objective_function(A,x,m,n),stepsize(A,x,m,n,s))

      # break loop if iteration count passes 10000
      if (count==10000) {
        break
      }
    }
    return(path)
}
```

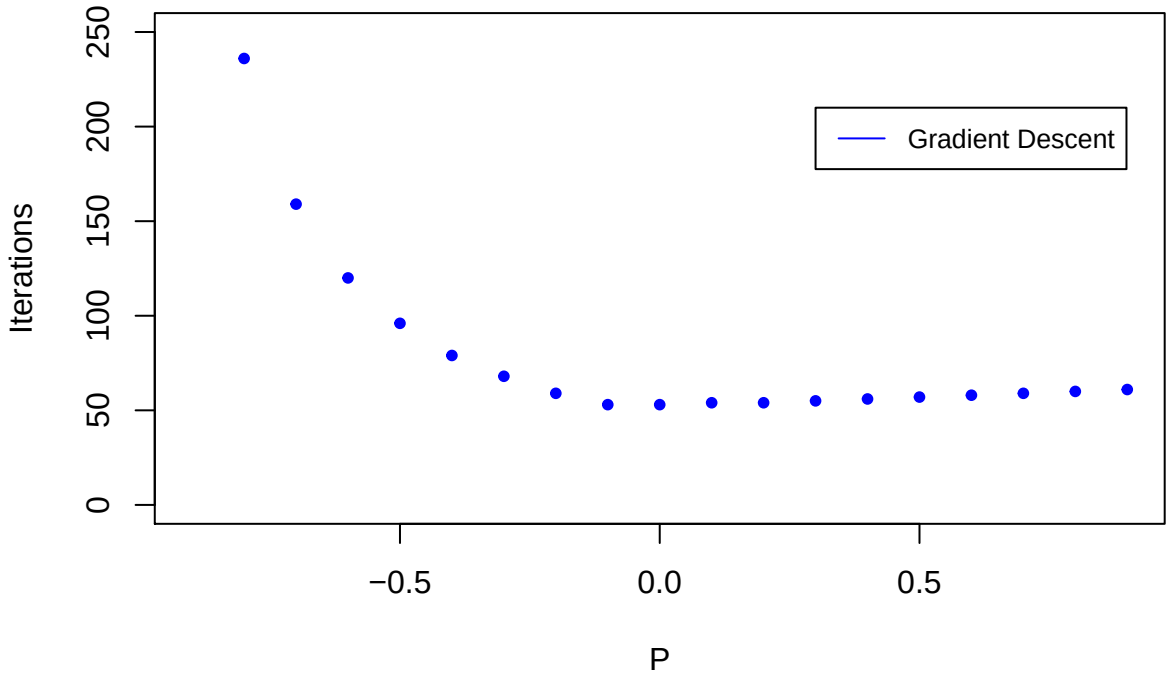
Step 2: Convergence of gradient descent in two-dimensional case

I implement a testing procedure that loops over 19 different values of ρ, each time calling the gradient descent function to minimise the function with the respective ρ-value, and counting the number of iterations that gradient descent requires. It then outputs a dataframe which I use to graph the iterations by value of ρ. I only use one starting point, very close to m. I start close to m because it is minimum for the first part of the function, and should thus be relatively close to the minimum of the full function. I cannot start exactly at m in this case because the function is not defined on any $x = 0$.

I only look for a local minimum, as for cases where n is large it is not clear how one should search for the global minimum, and for the 2-dimensional case finding the global minimum is trivial (comparing the four local minima found when departing from starting points lying in the four quadrants of x_1, x_2).

The testing function

```
# Test function that returns number of iterations by rho value
test_algorithm <- function(x,m,P,s,algorithm){

   # initialise result vector and iteration count
   result <- list()
   iterations <- 0
   iteration_path <- data.frame(P)
   count <- 0
   # loop over rho values and run algorithm
   for (p in P) {
     n <- 2
     a <- c(1,p,p,1)
     A <- matrix(a,2,2)
     count = count + 1
     result = append(result, algorithm(A,x,m,n,s))
     iterations = iterations + length(algorithm(A,x,m,n,s)[[1]])
     iteration_path$Iterations[[count]] = length(algorithm(A,x,m,n,s)[[1]])
   }
   return(iteration_path)
}

# Set variable values used for testing
x <- c(0.5,0.1)
m <- c(0.5,0)
P <- seq(-0.9, 0.9, 0.1)
s <- 0.1

# run test with above values
test <- test_algorithm(x,m,P,s,gradient_descent)
```

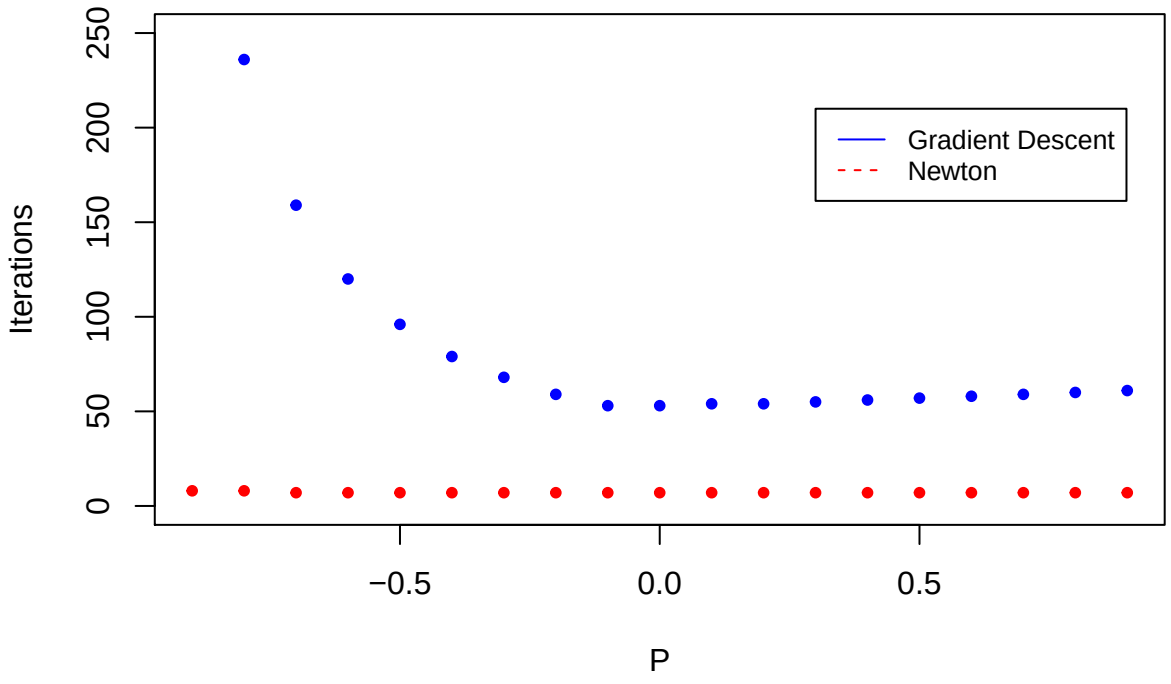


Interpretation: The flatter the bowl (i.e. smaller p), the longer the implementation of gradient descent takes. In the gradient descent function I set a precision threshold that stops the search only when the sum of the gradients becomes smaller than 0.0001. With this threshold in place, the gradient descent algorithm takes in between 459 iterations for ρ = -0.9 and 61 iterations for ρ = 0.9. Summing up iterations for all iterations of ρ, the gradient descent requires 1896 iterations with this specification.

Step 3: Comparison with a second line search method

The Newton algorithm function

```
# Newton's algorithm as a function
newton <- function(A,x,m,n,s) {

    # initialise iteration count and dataframe to record path
    count <- 1
    path <- data.frame(t(x))
    path$Z = objective_function(A,x,m,n)

    # start iterations, continue until gradient vector is close to zero
    while (sum(gradient(A,x,m)^2) > 0.0001) {

      # take step
      x <- x - inv_hessian(A,x) %*% gradient(A,x,m)
      count = count + 1

      # record step taken
      path[count,] = c(t(x),objective_function(A,x,m,n))

      # break loop if iteration count passes 10000
      if (count==10000) {
        break
      }
    }
    # output result
    return(path)
}
```



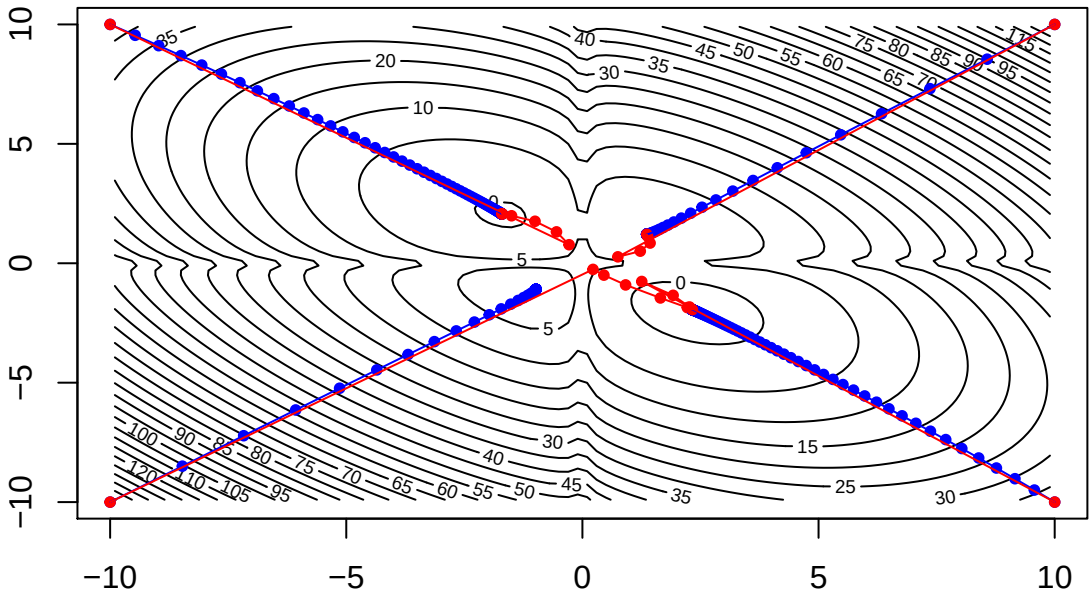
As can easily be seen from the above plot, Newton's algorithm performs significantly better. It requires only 135 iterations in total to minimise all 19 functions in comparison to the 1896 iterations required by gradient descent. This is because it makes much larger steps. The reason it performs so much better is that it does not use a scalar step size, but instead takes into account the 'speed' with which the gradient is changing at its current position, and it does so in every dimension.

Plot function

```
# initialise grid
grid_frame <- matrix(0,100,100)

# generate grid values
plot_size <- seq(-9.9,10,0.2)

# plotting function
draw_plot <-function(A,x,m,s) {
    # loop over grid and fill with values of obective function to generate surface
  n <- 2
  count_row = 0
  for (ii in plot_size) {
    count_row = count_row + 1
    count_column = 0
    for (iii in plot_size) {
      count_column = count_column + 1
      x <- c(ii, iii)
      grid_frame[count_row, count_column] = objective_function(A, x, m, n)
    }
  }
  # function to add paths
  add_paths <- function(x) {
  # generate paths to draw
    path_g <- gradient_descent(A, x, m, n, s)
    path_n <- newton(A, x, m, n, s)
    points(path_g$X1, path_g$X2, col = 'blue', pch = 20)
    points(path_n$X1, path_n$X2, col = 'red', pch = 20)
    lines(path_g$X1, path_g$X2, col = 'blue')
    lines(path_n$X1, path_n$X2, col = 'red')
  }
  # add contour
  contour(x = plot_size,
  y = plot_size,
  z = grid_frame,
  nlevels = 30)

  # add paths
  add_paths(c(10, 10))
  add_paths(c(-10, 10))
  add_paths(c(-10, -10))
  add_paths(c(10, -10))
}
```

Example plot 1

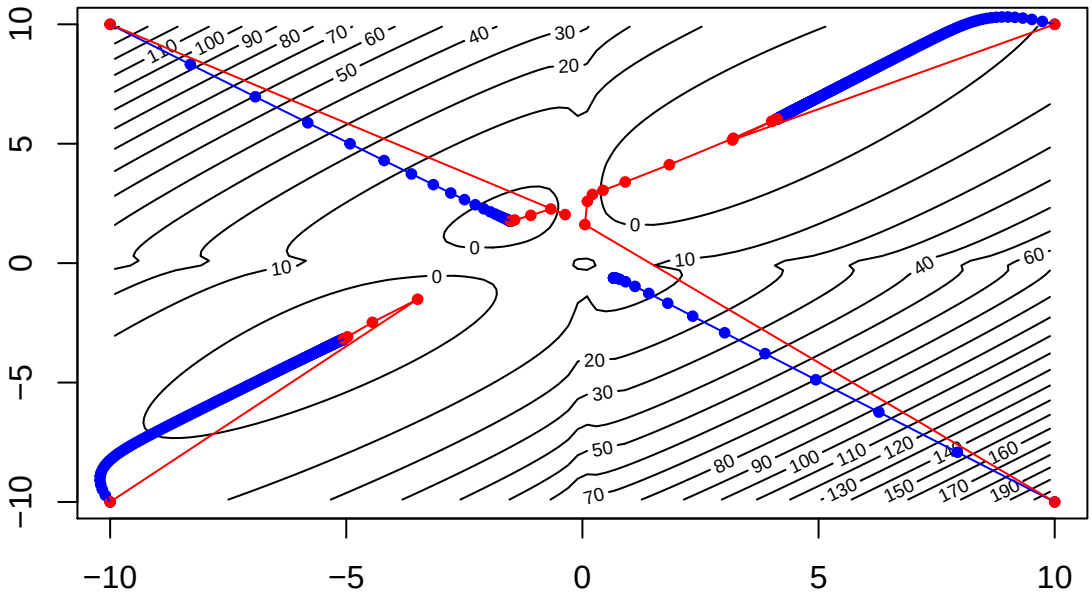
```
# Parameters for example plot 1
A <- matrix(0.5,2,2)
diag(A) <- 1
x <- c(0.5,0.1)
m <- c(0.5,0)

# plot 1
comparison_plot <- draw_plot(A,x,m,s)
```



I plot two examples. The first example uses $\rho = 0.5$, which makes all quadrants look quite similar. I plot the two algorithms starting from the extreme ends of the plot, to show them with longer path size. We can see that the Newton takes much larger steps initially than the Gradient Descent. Interestingly, in one case (lower left), the Newton's step is so large it jumps from one quadrant into the next (lower right), and finds the local minimum there. This shows that depending on the shape of A and the starting point used, the two algorithms can give different local results.

Example plot 2

```
# Parameters for example plot 2
A <- matrix(-0.9,2,2)
diag(A) <- 1
x <- c(0.1,2)
m <- c(0,2)

# plot 2
comparison_plot <- draw_plot(A,x,m,s)
```



In this example, i chose ρ = -0.9, which stretches the function contours into ellipsoids. Here again we see that Newton's algorithm jumps from the lower right into the upper right quadrant.

Step 4: Study how the performance of the two methods scales

To study the performance of the two algorithms for functions of higher dimensions, I use two functions: the simulate function randomises parameters for A, x, m depending on the size of n, and calls the two algorithms. The simulate_multiple function then calls the simulate function for different sizes of n.

```
# function that randomises A, x and m and runs the two algorithms
simulate <- function(n, s, algorithm1, algorithm2) {
  m <- runif(n, min=-1, max=1)
  x <- m
  A <- matrix(data = runif(n^2,-0.2,0.2), nrow = n, ncol = n)
  A <- t(A)%*%A
  a1_results <- algorithm1(A,x,m,n,s)
  a2_results <- algorithm2(A,x,m,n,s)
  iterations1 <- length(a1_results$Z)
  iterations2 <- length(a2_results$Z)
  results <- list(iterations1, iterations2)
  return(results)
}

# function that runs simulation 10 times and averages the results
simulate_multiple <- function(start, end, by, step){
  simulation_seq <- seq(start,end,by)
  simulation_result <- data.frame(simulation_seq,0,0)
  count <- 1
  for (sim in simulation_seq) {
    simulation_result[count,] = c(sim,simulate(sim, step, gradient_descent, newton))
    count = count + 1
  }
  return(simulation_result)
}

final_result <- simulate_multiple(100,800,100,1)
```

Table 1: Simulations

n	Gradient Descent Iterations	Newton Iterations
100	141	12
200	219	9
300	215	10
400	270	15
500	333	13
600	140	12
700	265	22
800	310	11

I find that the Newton algorithm scales better than the gradient descent in terms of computing time. The time gradient descent takes to run increases significantly with n, though the number of iterations required does not. However, I expect that in terms of computing cost, the Newton algorithm could be more expensive at some point, because it has to invert A and multiply it, which is quadratic cost for large n. In general, for large n the Newton method will likely be too costly, unless we have a matrix that is very easy to invert.

The Newton algorithm is much better in terms of number of iterations because it chooses a different stepsize in every dimension, based on the Hessian, which tell us by how much the gradient is changing in any particular

point, and so the algorithm knows when it can afford to take large steps or small steps, and that in every dimension.