Chapter 8: Performance Optimization

Brandon Morgan

1/17/2021

E9.5

We are given the following vector function in reduced form:

$$F(X) = (1 + (x_1 + x_2 - 5)^2)(1 + (3x_1 - 2x_2)^2)$$

1

Newton's method requires knowledge of the gradient and Hessian matrix. The gradient of a function is given by:

$$\nabla F(X) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(x) \\ \frac{\partial}{\partial x_2} F(x) \end{bmatrix}$$

which can be shown to be:

$$\nabla F(X) = \begin{bmatrix} 36x_1^3 + 18x_1^2 x_2 - 270x_1^2 - 22x_1 x_2^2 + 60x_1 x_2 + 470x_1 - 4x_2^3 + 80x_2^2 - 310x_2 - 10 \\ 6x_1^3 - 22x_1^2 x_2 + 30x_1^2 - 12x_1 x_2^2 + 160x_1 x_2 - 310x_1 + 16x_2^3 - 120x_2^2 + 21x_2 - 10 \end{bmatrix}$$

The Hessian matrix is given by:

$$\nabla^2 F(x) = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2} F(x) & \frac{\partial^2}{\partial x_1 \partial x_2} F(x) \\ \frac{\partial^2}{\partial x_2 \partial x_1} F(x) & \frac{\partial^2}{\partial x_2^2} F(x) \end{bmatrix}$$

Where it can be shown from our example that

$$\frac{\partial^2}{\partial x_1^2} F(x) = 108x_1^2 + 36x_1 x_2 - 540x_1 + 60x_2 + 470 - 22x_2^2$$

$$\frac{\partial^2}{\partial x_2^2} F(x) = -22x_1^2 + 160x_1 - 24x_1 x_2 + 48x_2^2 + 210 - 240x_2$$

$$\frac{\partial^2}{\partial x_1 \partial x_2} F(x) = 18x_1^2 + 60x_1 - 44x_1 x_2 + 160x_2 - 12x_2^2 - 310$$

$$\frac{\partial^2}{\partial x_2 \partial x_1} F(x) = 18x_1^2 + 60x_1 - 44x_1 x_2 + 160x_2 - 12x_2^2 - 310$$

Now we can start our iteration of Newton's method from the initial guess $x_0 = [10, 10]^T$. First, we evaluate our gradient and Hessian matrices at this initial point.

$$g_0 = \nabla F([10, 10]^T) = \begin{bmatrix} 16590 \\ -7900 \end{bmatrix}$$

$$A_0 = \nabla^2 F([10, 10]^T) = \begin{bmatrix} 7870 & -1910 \\ -1910 & -390 \end{bmatrix}$$

Next, we will comput our next point x_1 by the following equation: $x_1 = x_0 - A_0^{-1} g_0$:

```
A = matrix(c(7870, -1910, -1910, -390), ncol=2)
g = matrix(c(16590, -7900), ncol=1)
init = matrix(c(10, 10), ncol=1)
init - solve(A)%*%g
```

```
##           [,1]
## [1,] 6.790559
## [2,] 5.461622
```

2

We will repeat part 1 but this time with the initial point $x_0 = [2, 2]^T$. First, we evaluate our gradient and Hessian matrices at this initial point.

$$g_0 = \nabla F([2, 2]^T) = \begin{bmatrix} 14 \\ -404 \end{bmatrix}$$

$$A_0 = \nabla^2 F([2, 2]^T) = \begin{bmatrix} -2 & -22 \\ -22 & 58 \end{bmatrix}$$

Next, we will comput our next point x_1 by the following equation: $x_1 = x_0 - A_0^{-1} g_0$:

```
A = matrix(c(-2, -22, -22, 58), ncol=2)
g = matrix(c(14, -404), ncol=1)
init = matrix(c(2, 2), ncol=1)
init - solve(A)%*%g
```

```
##         [,1]
## [1,] -11.46
## [2,]   3.86
```

3

From Chapter 8, the stationary points of a function can be found by finding the roots of the gradient such that it equals zero, $\nabla F(X) = 0$.

As calculated earlier, here is our gradient function:

$$\nabla F(X) = \begin{bmatrix} 36x_1^3 + 18x_1^2 x_2 - 270x_1^2 - 22x_1 x_2^2 + 60x_1 x_2 + 470x_1 - 4x_2^3 + 80x_2^2 - 310x_2 - 10 \\ 6x_1^3 - 22x_1^2 x_2 + 30x_1^2 - 12x_1 x_2^2 + 160x_1 x_2 - 310x_1 + 16x_2^3 - 120x_2^2 + 21x_2 - 10 \end{bmatrix}$$

Unfortunately, at this time the roots of these two equations cannot be calculated by hand. Instead Newton's algorithm will be implemented and ran until convergence.

1

```
library(purrr) # used for map2_dbl function
library(plotly) # used for 3D plotting
```

```
## Warning: package 'plotly' was built under R version 3.6.3
```

```
## Loading required package: ggplot2
```

```
## Warning: package 'ggplot2' was built under R version 3.6.3
```

```
##
## Attaching package: 'plotly'
```

```
## The following object is masked from 'package:ggplot2':
##
##     last_plot
```

```
## The following object is masked from 'package:stats':
##
##     filter
```

```
## The following object is masked from 'package:graphics':
##
##     layout
```

```
library(ggplot2) # used for plotting
```

```
fun = function(x, y) { # our original function
  (1+(x+y-5)^2)*(1+(3*x-2*y)^2)
}
```

In figure 1 we have a three dimensional rendering of our function to add interpretation of our contour plot. In figure 2 we have the same rendering of our function but on a shrunken domain about where we expect the minumum to be located.

Here we have the contour of our function:

```
# BOUNDS USED FOR CONTOUR
pointsX = seq(-10, 15, length=200)# create a 200x1 vector of values for x axis
pointsY = seq(-10, 15, length=200) # create a 200x1 vector of values for y axis
myGrid = expand.grid(pointsX, pointsY) # create 200x200 grid of points
z = map2_dbl(myGrid$Var1, myGrid$Var2, ~fun(.x, .y)) # maps all possible combinations
# of the grid through the function
x = myGrid$Var1
y =  myGrid$Var2
myPlot = plot_ly(x=~x, y=~y, z=~z, type="mesh3d", intensity=~z,
        colors = colorRamp(c("black","red","yellow","chartreuse3")))
#myPlot
myGrid$z = z

v = ggplot( myGrid,aes(x, y, z=z)) +geom_contour_filled()
v
```

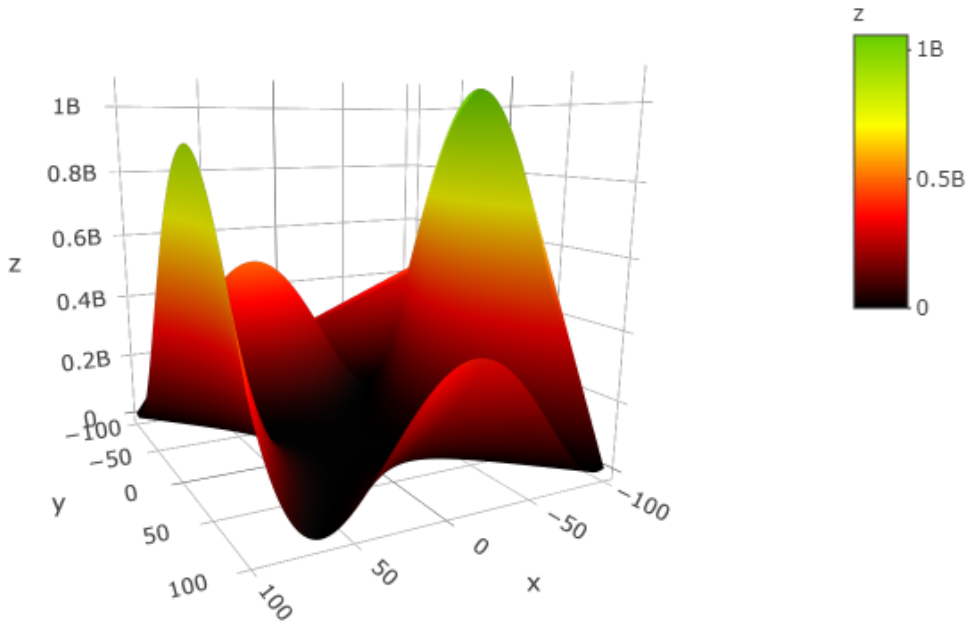


Figure 1: Function 3D graph

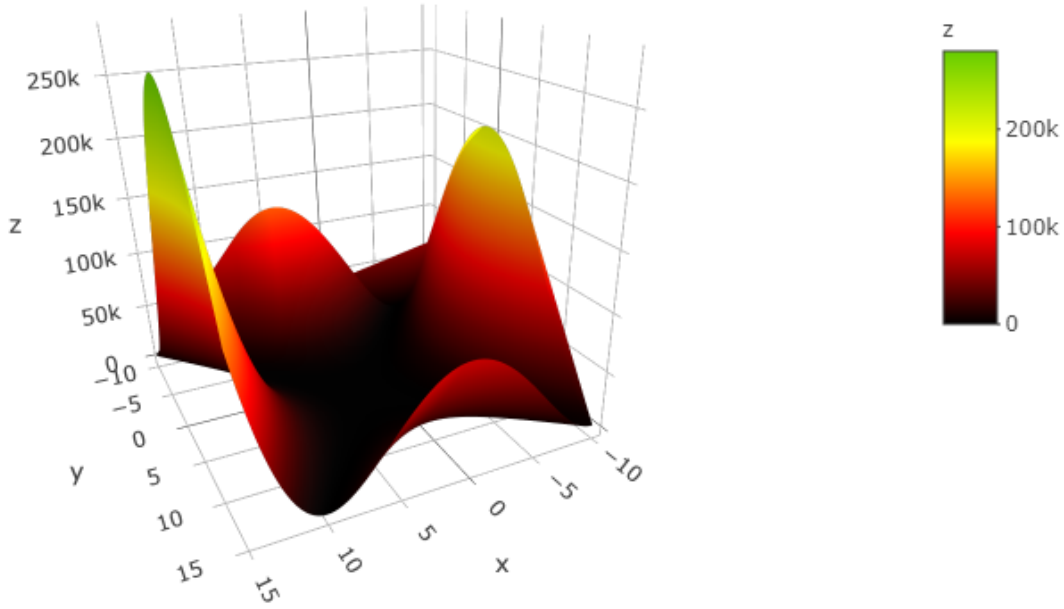
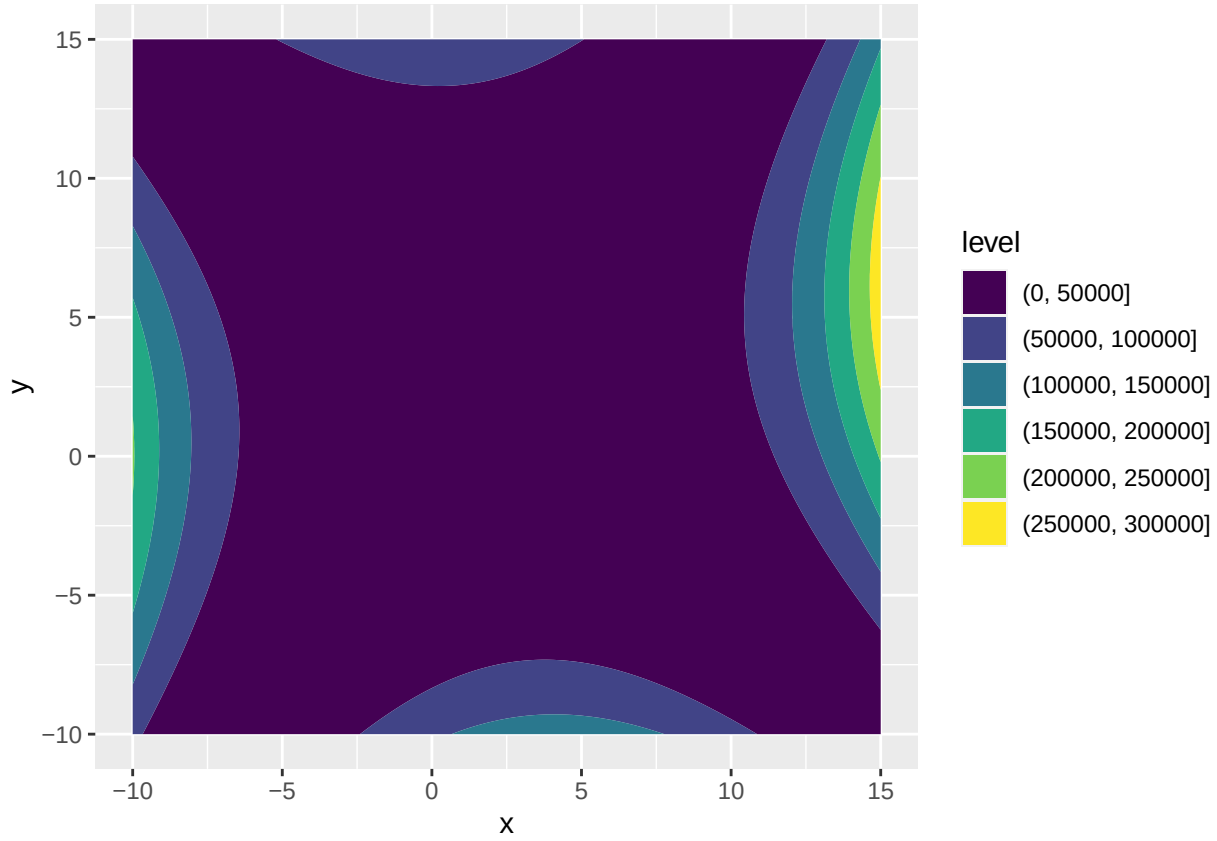


Figure 2: Function 3D graph (smaller bounds)



6

Instead of MATLAB code, R code will be used instead to create Newton's Method function.

Here we have our Netwon's Method function. It requires two function arguments, the hessian and gradient to which a vector can be passed in; this allows the function to be re-used for different examples as the hessian and gradient are hard coded not inside the function, but by the user. The maximum iteration is given so as to not enter into an infinity loop and to see how long it will take to compute the stationary point. The tolerance value is given as another stopping criterion where if the gradient at the x value is less than the tolerance, then the stationary point has been found to within said tolerance. To compare the tolerance with the matrix value of the gradient, the Frobenius Norm was used to measure its length, which can be calculated by $||A||_F = \sqrt{\Sigma a_i^2}$ for vectors.

```
hessian = function(X) { # hard coded for our example
    x = X[1,1]
    y = X[2,1]
    fxx = 108*x^2+36*x*y-540*x+60*y+470-22*y^2
    fyy = -22*x^2+160*x-24*x*y+48*y^2+210-240*y
    fxy = 18*x^2+60*x-44*x*y+160*y-12*y^2-310
    matrix(c(fxx, fxy, fxy, fyy),ncol=2)
}

gradient = function(X) { # hard coded for our example
    x = X[1,1]
    y = X[2,1]
```

```
    g1 = 36*x^3+18*x^2*y-270*x^2-22*x*y^2+60*x*y+470*x-4*y^3+80*y^2-310*y-10
    g2 = 6*x^3-22*x^2*y+30*x^2-12*x*y^2+160*x*y-310*x+16*y^3-120*y^2+21*y-10
    matrix(c(g1, g2), ncol=1)
}

newtons_method = function(init_point, gradient, hessian, maxIter, tol, mask) {


  x = init_point


  for(i in 1:maxIter) {

    if(!mask) { # if the user does not want print out statements
      print(sprintf("      Iteration: %d", i))
      print("x value:")
      print(x)
    }

    A = hessian(x)
    g = gradient(x)
    x = x - solve(A)%*%g

    if(!mask) { # if the user does not want print out statements
       print("gradient value:")
      print(g)
    }


    if(norm(g, "F") < tol) { # convert gradient into single value
                             # through frobenius norm
      print("Stationary Point Found at point:")
      print(x)
      print(sprintf("Iterations taken: %d", i))
      return(x)
    }
  }

  print("MAXIMUM ITERATIONS REACHED")
  print("Current point is: ")
  print(x)
  print("Error: ")
  print(abs(norm(g, "F")-tol)) # computing the error
  return(x)
}
```

Part 1

Here we have the first 2 iterations of our method for the initial pont $x_0 = [10, 10]^T$.

```
init = matrix(c(10,10), ncol=1)
min = newtons_method(init,gradient, hessian, 2, 1e-10, 0)
```

```
## [1] "      Iteration: 1"
## [1] "x value:"
##      [,1]
## [1,]   10
## [2,]   10
## [1] "gradient value:"
##       [,1]
## [1,] 16590
## [2,] -7900
## [1] "      Iteration: 2"
## [1] "x value:"
##          [,1]
## [1,] 6.790559
## [2,] 5.461622
## [1] "gradient value:"
##           [,1]
## [1,]  4347.633
## [2,] -1748.418
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##          [,1]
## [1,] 5.035475
## [2,] 2.551052
## [1] "Error: "
## [1] 4686.031
```

Now if we let the algorithm run until convergence, we see that it will take 123 iterations to converge to the minimum to within the tolerance level.

```
min = newtons_method(init,gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##          [,1]
## [1,] 1.163907
## [2,] 7.221783
## [1] "Iterations taken: 123"
```

Part 2

Here we have the first 2 iterations of our method for the initial pont $x_0 = [2, 2]^T$.

```
init = matrix(c(2,2), ncol=1)
min = newtons_method(init,gradient, hessian, 2, 1e-10, 0)
```

```
## [1] "      Iteration: 1"
## [1] "x value:"
##      [,1]
## [1,]    2
## [2,]    2
## [1] "gradient value:"
##      [,1]
## [1,]   14
```

```
## [2,] -404
## [1] "      Iteration: 2"
## [1] "x value:"
##        [,1]
## [1,] -11.46
## [2,]   3.86
## [1] "gradient value:"
##           [,1]
## [1,] -85045.31
## [2,] -18515.89
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##           [,1]
## [1,] -6.937036
## [2,]  3.437385
## [1] "Error: "
## [1] 87037.59
```

Now if we let the algorithm run until convergence, we see that it will take 38 iterations to converge to the minimum to within the tolerance level.

```
min = newtons_method(init,gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##          [,1]
## [1,] 1.163907
## [2,] 7.221783
## [1] "Iterations taken: 38"
```

Conclusion

As we can see from the number of iterations needed above to converge, the initial point $x_0 = [2, 2]$ converged in almost a third of the time as the initial point $x_0 = [10, 10]$, this is was not due to the point $x_0 = [2, 2]$ being closer to the minimum point, in terms of Euclidean distance, but because it was located at a point on the gradient which had a steeper slope.