Chapter 8: Performance Optimization

Brandon Morgan

1/17/2021

E9.10/11

Introduction

This is not the particular answer to either question E9.10 nor E9.11; however, Steepest Descent, Newton's Method, and all three forms of Conjugate Gradient will be used on vector function given from E8.5.

Here we have our vector function:

$$F(X) = (x_1 + x_2)^4 - 12x_1x_2 + x_1 + x_2 - 1$$

We can find the stationary points by finding the roots of the gradient, $\nabla F(X) = 0$. 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 calculated to

$$\nabla F(X) = \begin{bmatrix} 4(x_1 + x_2)^3 - 12x_2 + 1 \\ 4(x_1 + x_2)^3 - 12x_1 + 1 \end{bmatrix}$$

The roots of where the gradient equals zero was given in part 1 of E8.5. The following are all stationary points:

$$x^1 = \begin{bmatrix} -0.6504 \\ -0.6504 \end{bmatrix}, x^2 = \begin{bmatrix} 0.085 \\ 0.085 \end{bmatrix} x^3 = \begin{bmatrix} 0.5655 \\ 0.5655 \end{bmatrix}$$

These points satisfy the first order requirement. For the second order requirement, we must evaluate the Hessian matrix.

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

$$\frac{\partial^2}{\partial x_1^2} F(x) = 12(x_1 + x_2)^2$$

$$\frac{\partial^2}{\partial x_2^2} F(x) = 12(x_1 + x_2)^2$$

$$\frac{\partial^2}{\partial x_1 \partial x_2} F(x) = \frac{\partial^2}{\partial x_2 \partial x_1} F(x) = 12(x_1 + x_2)^2 - 12$$

which together forms the following complete Hessian Matrix:

$$\nabla^2 F(X) = \begin{bmatrix} 12(x_1 + x_2)^2 & 12(x_1 + x_2)^2 - 12 \\ 12(x_1 + x_2)^2 - 12 & 12(x_1 + x_2)^2 \end{bmatrix}$$

If the Hessian matrix is positive definite at the stationary point, then the point is a strong minimum. A matrix is positive definite if all its eigen values are positive, greater than zero. If the Hessian matrix is semi-positive definite at the stationary point, then the point is either a weak or strong minimum. A matrix is positive definite if all its eigen values are positive or equal to zero. If any of the eigen values for the Hessian matrices evaluated at the stationary point are negative, then the stationary point is a saddle point.

Point $x^1 = \begin{bmatrix} -0.6504 \\ -0.6504 \end{bmatrix}$

$$\nabla^2 F(\begin{bmatrix} -0.6504 \\ -0.6504 \end{bmatrix}) = \begin{bmatrix} 20.305 & 8.305 \\ 8.305 & 20.305 \end{bmatrix}$$

```
A = matrix(c(20.305, 8.305, 8.305, 20.305), ncol=2)
eigen(A)
```

```
## eigen() decomposition
## $values
## [1] 28.61 12.00
##
## $vectors
##            [,1]       [,2]
## [1,] 0.7071068 -0.7071068
## [2,] 0.7071068  0.7071068
```

As we can see from all the eigen values being positive, the Hessian matrix evaluated at this stationary point is positive definite; therefore, this stationary point is a strong minimum.

Point $x^2 = \begin{bmatrix} 0.085 \\ 0.085 \end{bmatrix}$

$$\nabla^2 F(\begin{bmatrix} -0.6504 \\ -0.6504 \end{bmatrix}) = \begin{bmatrix} 0.3468 & -11.6532 \\ -11.6532 & 0.3468 \end{bmatrix}$$

```
A = matrix(c(0.3468, -11.6532, -11.6532, 0.3468), ncol=2)
eigen(A)
```

```
## eigen() decomposition
## $values
## [1]  12.0000 -11.3064
##
## $vectors
##            [,1]       [,2]
## [1,] -0.7071068 -0.7071068
## [2,]  0.7071068 -0.7071068
```

As we can see from one of the eigen values being positive, the Hessian matrix evaluated at this stationary point is neither full nor semi positive definite; therefore, this stationary point is a saddle point, it is a minimum along the first eigenvector ($\lambda_1 > 0$), but a maximum along the second eigenvector ($\lambda_2 < 0$).

Point $x^3 = \begin{bmatrix} 0.5655 \\ 0.5655 \end{bmatrix}$

$$\nabla^2 F(\begin{bmatrix} -0.6504 \\ -0.6504 \end{bmatrix}) = \begin{bmatrix} 15.35 & 3.35 \\ 3.35 & 15.35 \end{bmatrix}$$

```
A = matrix(c(15.35, 3.35,3.35, 15.35), ncol=2)
eigen(A)
```

```
## eigen() decomposition
## $values
## [1] 18.7 12.0
##
## $vectors
##           [,1]       [,2]
## [1,] 0.7071068 -0.7071068
## [2,] 0.7071068  0.7071068
```

Plots

In Figure 1 we have a three dimensional rendering of the quadtratic vector function.

```
library(purrr) # used for map2_dbl function
library(plotly) # used for 3D plotting
library(ggplot2) # used for plotting
```

```
fun = function(x,y) { # our original function
  (x + y)^4 - 12*x*y+x+y+1
}
```

Here we have our contour plot with all three of our stationary points. Notice that the two outermost points lie in the low valleys, showing that they are minimums, while the middle point is not, showing that it is a saddle point.

```
# BOUNDS USED FOR CONTOUR
pointsX = seq(-2, 2, length=200)# create a 200x1 vector of values for x axis
pointsY = seq(-2, 2, length=200) # create a 200x1 vector of values for y axis
myGrid = expand.grid(pointsX, pointsY) # create 200x200 grid of points
```

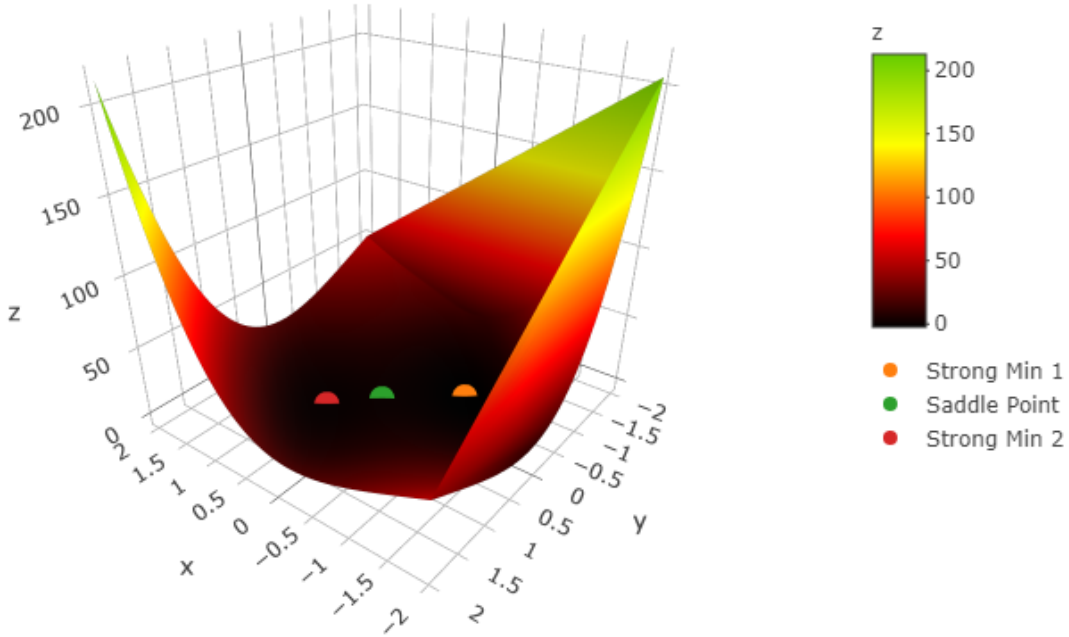
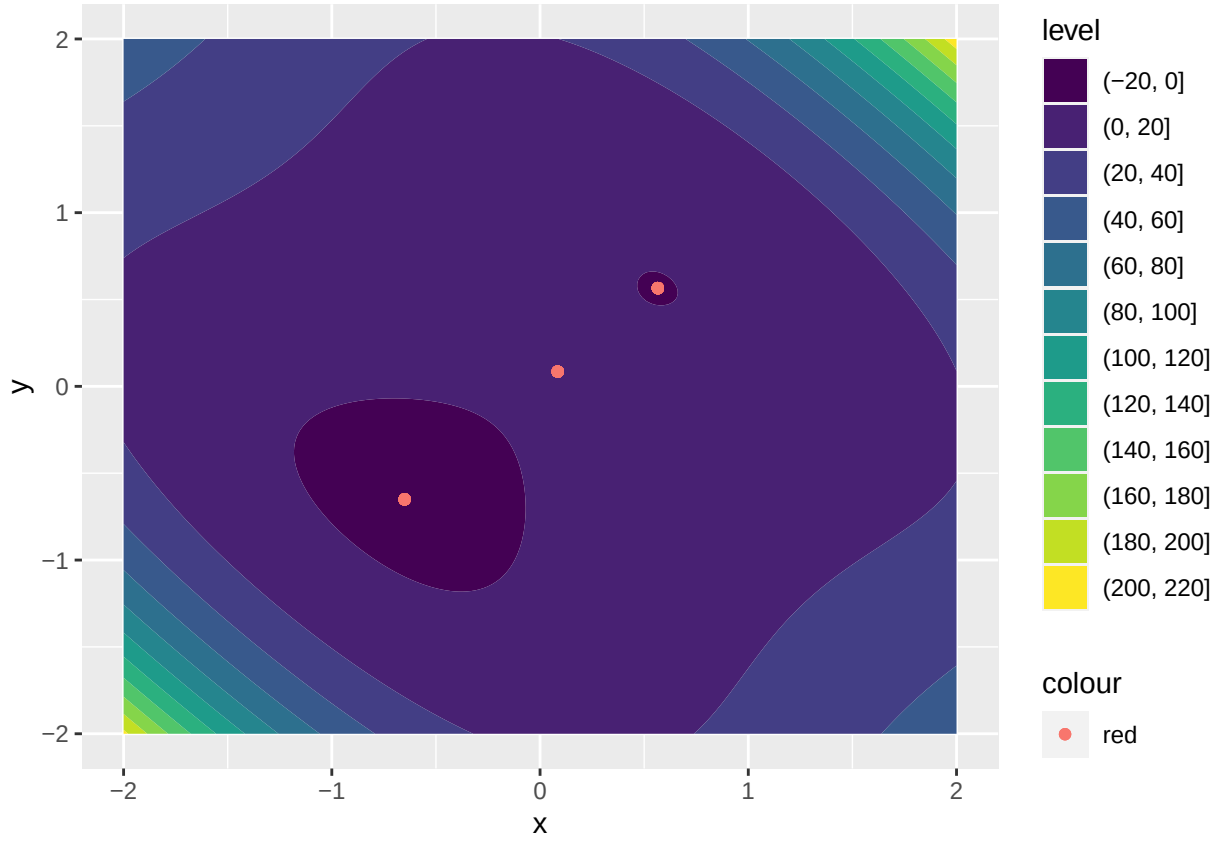


Figure 1: Function 3D graph

```
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")))
myGrid$z = z

# add another layer onto the 3D rendering, in this case a scatterplot
# of a single point, the stationary point, AKA minima
myPlot = add_trace(p=myPlot, z = c(fun(-0.6504, -0.6504)), x=c(-0.6504), y=c(-0.6504),
                   type="scatter3d", name="Strong Min 1")
myPlot = add_trace(p=myPlot, z = c(fun(0.085, 0.085)), x=c(0.085), y=c(0.085),
                   type="scatter3d",name="Saddle Point")
myPlot = add_trace(p=myPlot, z = c(fun(.5655, .5655)), x=c(.5655), y=c(.5655),
                   type="scatter3d",name="Strong Min 2")

#myPlot
v = ggplot( myGrid,aes(x, y, z=z)) +geom_contour_filled()
v+geom_point(aes(x=-0.6504, y=-0.6504, colour="red"))+
  geom_point(aes(x=0.085, y=0.085, colour="red"))+
  geom_point(aes(x=0.5655, y=0.5655, colour="red"))
```



We discussed earlier that the are two strong minimums, but which one is the global minimum? We now take the function of each strong minimum and the value with the lesser value is global:

```r
fun(-0.6504, -0.6504) # point 1
```

```
## [1] -2.513905
```

```r
fun(.5655, .5655) # point 2
```

```
## [1] -0.07023014
```

As we can see, the point $x^1 = [-0.6504, -0.6504]$ has the minimum function value, therefore it is the global minimum. Now compare this point to the contour plot above.

Algorithms

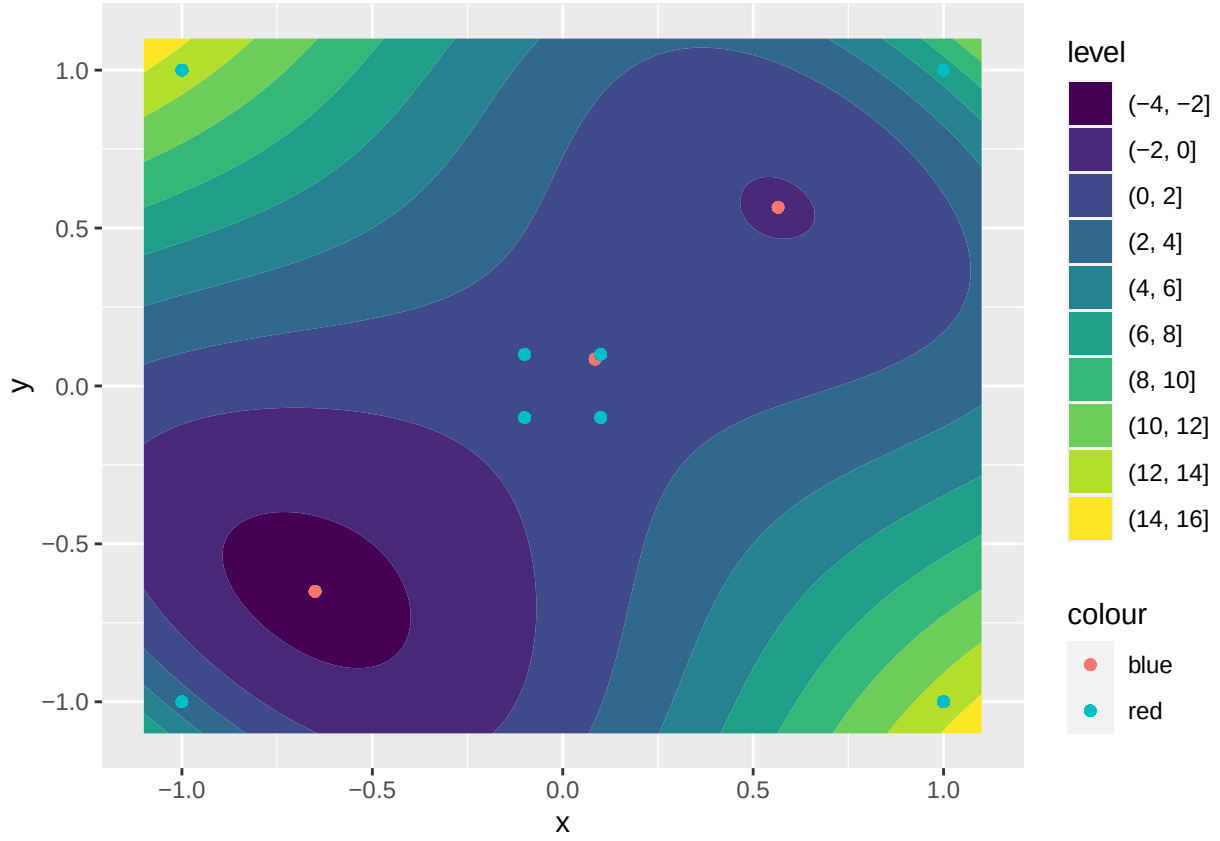
Instead of MATLAB code, R code will be used instead to create the algorithms.

```r
# external functions to be passed into our algorithms
hessian = function(X) {
  x = X[1,1]
  y = X[2,1]
  fxx = 12*(x + y)^2
  fxy = 12*(x + y)^2 - 12
  matrix(c(fxx, fxy, fxy, fxx), ncol=2)
}

gradient = function(X) {
    x = X[1,1]
    y = X[2,1]
    g1 = 4*(x+y)^3-12*y+1
    g2 = 4*(x+y)^3-12*x+1
    matrix(c(g1, g2), ncol=1)
}
# initial points
p1 = matrix(c(-1, -1), ncol=1)
p2 = matrix(c(-1, 1), ncol=1)
p3 = matrix(c(1, -1), ncol=1)
p4 = matrix(c(1, 1), ncol=1)
p5 = matrix(c(-.1, -.1), ncol=1)
p6 = matrix(c(-.1, .1), ncol=1)
p7 = matrix(c(.1, -.1), ncol=1)
p8 = matrix(c(.1, .1), ncol=1)
```

We will examine each algorithm at eight different possible initial points, we will overlay these initial points onto the contour plot below along with the stationary points in blue:

```r
# BOUNDS USED FOR CONTOUR
pointsX = seq(-1.1, 1.1, length=200)# create a 200x1 vector of values for x axis
pointsY = seq(-1.1, 1.1, 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")))
myGrid$z = z
# add another layer onto the 3D rendering, in this case a scatterplot
# of a single point, the stationary point, AKA minima
#myPlot
v = ggplot( myGrid,aes(x, y, z=z)) +geom_contour_filled()+
  geom_point(aes(x=-0.6504, y=-0.6504, color='blue'))+
  geom_point(aes(x=0.085, y=0.085, color='blue'))+
  geom_point(aes(x=0.5655, y=0.5655, color='blue'))+
  geom_point(aes(x=-1.0, y=-1.0, color='red'))+
  geom_point(aes(x=-1.0, y=1.0, color='red'))+
  geom_point(aes(x=1.0, y=-1.0, color='red'))+
  geom_point(aes(x=1.0, y=1.0, color='red'))+
  geom_point(aes(x=-.1, y=-.1, color='red'))+
  geom_point(aes(x=-.1, y=.1, color='red'))+
  geom_point(aes(x=.1, y=-.1, color='red'))+
  geom_point(aes(x=.1, y=.1, color='red'))
v
```



Steepest Descent

Here we have our steepest descent function. For this problem, the function will requiring input for the initial point, gradient, hessian, algorithm maximum iteration, and tolerance. 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.

Because our function is in quadtratic form, we will use a learning rate to minimize along the line $x_{k+1} = x_k + \alpha_k p_k$ by

$$\alpha_k = \frac{g_k^T p_k}{p_k^T A p_k}$$

```
steepest_descent = 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)
    }

    g = gradient(x) # find gradient at x

    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)
    }

    # creating the search direction vectors
    p = -g

    alpha = -(t(g)%*%p) / (t(p)%*%hessian(x)%*%p) # learning rate
    x = x - alpha[1]*g # find next x


  }

  print("MAXIMUM ITERATIONS REACHED")
```

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

Here is our algorithm ran for each point:

```
min = steepest_descent(p1, gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##             [,1]
## [1,] -0.6504198
## [2,] -0.6504198
## [1] "Iterations taken: 7"
```

```
min = steepest_descent(p2, gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##             [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 80"
```

```
min = steepest_descent(p3, gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##             [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 80"
```

```
min = steepest_descent(p4, gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.5654506
## [2,] 0.5654506
## [1] "Iterations taken: 7"
```

```
min = steepest_descent(p5, gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##             [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 5"
```

```
min = steepest_descent(p6, gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] -0.6504198
## [2,] -0.6504198
## [1] "Iterations taken: 22"
```

```
min = steepest_descent(p7, gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] -0.6504198
## [2,] -0.6504198
## [1] "Iterations taken: 22"
```

```
min = steepest_descent(p8, gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 4"
```

Newton's Method

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.

```
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)
}
```

```
min = newtons_method(p1,gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] -0.6504198
## [2,] -0.6504198
## [1] "Iterations taken: 7"
```

```
min = newtons_method(p2,gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 4"
```

```
min = newtons_method(p3,gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 4"
```

```
min = newtons_method(p4,gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##           [,1]
## [1,] 0.5654506
## [2,] 0.5654506
## [1] "Iterations taken: 7"
```

```
min = newtons_method(p5,gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 5"
```

```
min = newtons_method(p6,gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 4"
```

```
min = newtons_method(p7,gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 4"
```

```
min = newtons_method(p8,gradient, hessian, 1000, 1e-10, 1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 4"
```

Conjugate gradient

Here we have our Conjugate Gradient function. For this problem, the function will requiring input for the initial point, hessian, gradient, algorithm maximum iteration, and tolerance. 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.

Because our function is in quadtratic form, we will use a learning rate to minimize along the line $x_{k+1} = x_k + \alpha_k p_k$ by

$$\alpha_k = \frac{g_k^T p_k}{p_k^T A p_k}$$

```
conjuage_gradient = function(init_point, gradient, hessian, maxIter, tol, mask, betaMethod) {

  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)
    }

    g = gradient(x) # find gradient at x

    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)
    }

    # creating the search direction vectors
    if(i==1) { # first search direction
      p = -g
    }
    else { # use beta
      if(betaMethod == 1) { # Common choice
        beta = (t(g-g0)%*%g)/(t(g0)%*%p)
      }
      else if (betaMethod == 2) { # hestens and stiefel
         beta = (t(g)%*%g)/(t(g0)%*%g)
      }
      else { # Fletcher and Reeves
         beta = (t(g-g0)%*%g)/(t(g0)%*%g0)
      }

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

```
      p = -g+beta[1]*p
    }


    alpha = -(t(g)%*%p) / (t(p)%*%hessian(x)%*%p) # learning rate
    x = x - alpha[1]*g # find next x

    g0 = g # update previous gradient to current
  }

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

Algorithm with β calculation agorithm 1:

```
min = conjuage_gradient(p1,gradient, hessian, 1000, 1e-10, 1,1)
```

```
## [1] "Stationary Point Found at point:"
##             [,1]
## [1,] -0.6504198
## [2,] -0.6504198
## [1] "Iterations taken: 29"
```

```
min = conjuage_gradient(p2,gradient, hessian, 1000, 1e-10, 1,1)
```

```
## [1] "Stationary Point Found at point:"
##             [,1]
## [1,] -0.6504198
## [2,] -0.6504198
## [1] "Iterations taken: 37"
```

```
min = conjuage_gradient(p3,gradient, hessian, 1000, 1e-10, 1,1)
```

```
## [1] "Stationary Point Found at point:"
##             [,1]
## [1,] -0.6504198
## [2,] -0.6504198
## [1] "Iterations taken: 37"
```

```
min = conjuage_gradient(p4,gradient, hessian, 1000, 1e-10, 1,1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.5654506
## [2,] 0.5654506
## [1] "Iterations taken: 29"
```

```
min = conjuage_gradient(p5,gradient, hessian, 1000, 1e-10, 1,1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 25"
```

```
min = conjuage_gradient(p6,gradient, hessian, 1000, 1e-10, 1,1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] -0.6504198
## [2,] -0.6504198
## [1] "Iterations taken: 30"
```

```
min = conjuage_gradient(p7,gradient, hessian, 1000, 1e-10, 1,1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] -0.6504198
## [2,] -0.6504198
## [1] "Iterations taken: 30"
```

```
min = conjuage_gradient(p8,gradient, hessian, 1000, 1e-10, 1,1)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 21"
```

Algorithm with β calculation agorithm 2:

```
min = conjuage_gradient(p1,gradient, hessian, 1000, 1e-10, 1,2)
```

```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##            [,1]
## [1,] -0.6505976
## [2,] -0.6505976
## [1] "Error: "
## [1] 0.007203272
```

```
min = conjuage_gradient(p2,gradient, hessian, 1000, 1e-10, 1,2)
```

```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##            [,1]
```

```
## [1,] -0.1455571
## [2,] -0.1650647
## [1] "Error: "
## [1] 3.881178
```

```
min = conjuage_gradient(p3,gradient, hessian, 1000, 1e-10, 1,2)
```

```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##             [,1]
## [1,] -0.1650647
## [2,] -0.1455571
## [1] "Error: "
## [1] 3.881178
```

```
min = conjuage_gradient(p4,gradient, hessian, 1000, 1e-10, 1,2)
```

```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##          [,1]
## [1,] 0.565798
## [2,] 0.565798
## [1] "Error: "
## [1] 0.009203747
```

```
min = conjuage_gradient(p5,gradient, hessian, 1000, 1e-10, 1,2)
```

```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##             [,1]
## [1,] 0.08498046
## [2,] 0.08498046
## [1] "Error: "
## [1] 0.0001799488
```

```
min = conjuage_gradient(p6,gradient, hessian, 1000, 1e-10, 1,2)
```

```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##             [,1]
## [1,] -0.6489986
## [2,] -0.6518405
## [1] "Error: "
## [1] 0.02413794
```

```
min = conjuage_gradient(p7,gradient, hessian, 1000, 1e-10, 1,2)
```

```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##             [,1]
```

```
## [1,] -0.6518405
## [2,] -0.6489986
## [1] "Error: "
## [1] 0.02413794
```

```
min = conjuage_gradient(p8,gradient, hessian, 1000, 1e-10, 1,2)
```

```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##            [,1]
## [1,] 0.08496904
## [2,] 0.08496904
## [1] "Error: "
## [1] 2.987559e-06
```

Algorithm with β calculation agorithm 3:

```
min = conjuage_gradient(p1,gradient, hessian, 1000, 1e-10, 1,3)
```

```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##               [,1]
## [1,] -1.396278e+43
## [2,] -1.396278e+43
## [1] "Error: "
## [1] 1.231914e+131
```

```
min = conjuage_gradient(p2,gradient, hessian, 1000, 1e-10, 1,3)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 85"
```

```
min = conjuage_gradient(p3,gradient, hessian, 1000, 1e-10, 1,3)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] 0.08496922
## [2,] 0.08496922
## [1] "Iterations taken: 85"
```

```
min = conjuage_gradient(p4,gradient, hessian, 1000, 1e-10, 1,3)
```

```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##              [,1]
## [1,] 1.488656e+40
## [2,] 1.488656e+40
## [1] "Error: "
## [1] 1.492959e+122
```

```
min = conjuage_gradient(p5,gradient, hessian, 1000, 1e-10, 1,3)
```

```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##              [,1]
## [1,] -2.80341e+39
## [2,] -2.80341e+39
## [1] "Error: "
## [1] 9.970682e+119
```

```
min = conjuage_gradient(p6,gradient, hessian, 1000, 1e-10, 1,3)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] -0.6504198
## [2,] -0.6504198
## [1] "Iterations taken: 27"
```

```
min = conjuage_gradient(p7,gradient, hessian, 1000, 1e-10, 1,3)
```

```
## [1] "Stationary Point Found at point:"
##            [,1]
## [1,] -0.6504198
## [2,] -0.6504198
## [1] "Iterations taken: 27"
```

```
min = conjuage_gradient(p8,gradient, hessian, 1000, 1e-10, 1,3)
```

```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##              [,1]
## [1,] 8.055794e+37
## [2,] 8.055794e+37
## [1] "Error: "
## [1] 2.365866e+115
```

Analysis

Here we will gather all the possible initial points along with each function and their number of iterations needed. For conjugate gradient, β algorithm 1 will be used as the second algorithm never converged for any of the input points and the third failed miserably for four of the points.

Here we have our data from the algorithms, each initial point was labeled form 1-8, iteration count was kept, algorithm type (1: steepest, 2: newton, 3: conjugate), and the converged points (1: first min (neg), 2: saddle, 3: second min (pos)).

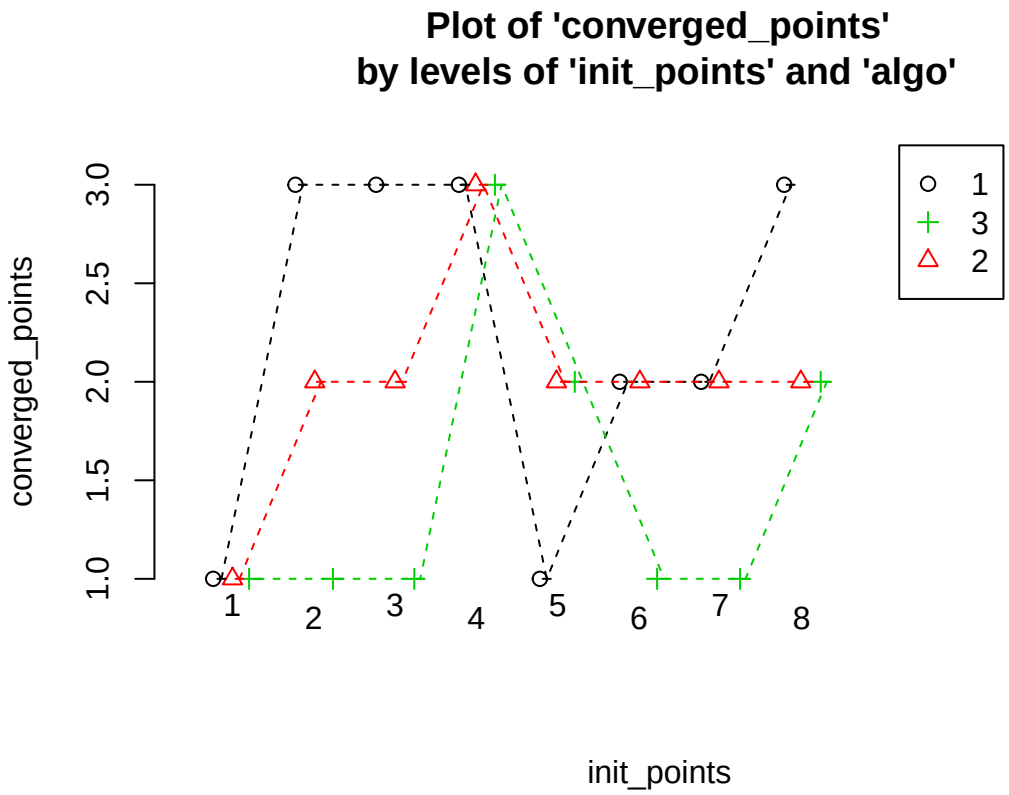
```
points = c(1, 2, 3, 4, 5, 6, 7, 8)

# steepest descent
mins = c(1, 3, 3, 3, 1, 2, 2, 3)
```

```
iter = c(13, 32, 32, 13, 7, 80, 80, 7)

# newton's method
mins = c(1, 2, 2, 3, 2, 2, 2, 2)
iter = c(7, 4, 4, 7, 5, 4, 4, 4)

# conjugate gradient, beta method 1
mins = c(1, 1, 1, 3, 2, 1, 1, 2)
iter = c(29, 37, 37, 29, 25, 30, 30, 21)

# combined into the following dataframe
df = data.frame(init_points=c(1,1,1,2,2,2,3,3,3,4,4,4,5,5,5,6,6,6,7,7,7,8,8,8),
                iter=c(13, 7, 29, 32, 4, 37, 32, 4, 37, 13, 7, 29, 7, 5, 25, 80, 4, 30, 80, 4, 30, 7, 4
df
```

```
##    init_points iter converged_points algo
## 1            1   13                1    1
## 2            1    7                1    2
## 3            1   29                1    3
## 4            2   32                3    1
## 5            2    4                2    2
## 6            2   37                1    3
## 7            3   32                3    1
## 8            3    4                2    2
## 9            3   37                1    3
## 10           4   13                3    1
## 11           4    7                3    2
## 12           4   29                3    3
## 13           5    7                1    1
## 14           5    5                2    2
## 15           5   25                2    3
## 16           6   80                2    1
## 17           6    4                2    2
## 18           6   30                1    3
## 19           7   80                2    1
## 20           7    4                2    2
## 21           7   30                1    3
## 22           8    7                3    1
## 23           8    4                2    2
## 24           8   21                2    3
```

Here we have an interaction plot of our results. We are plotting the converged_points against the initial points where each color represents the algorithm type. We can definetly see that for the first three initial points, each algorithm gets different results for the converged points. The fact that each of the algorithms cross when outputting converged points for different initial pionts show that no matter the initial point, the outputted converged point depends upon the algorithm. Notice that we have not measured what the algorithm should have converged to, theres been no talk of finding an error for the outputted converged point.

```
library(s20x) # used for analysizing interaction
interactionPlots(data=df,converged_points~init_points+algo)
```


Plot of 'converged_points'
by levels of 'init_points' and 'algo'
converged_points
1.0
1.5
2.0
2.5
3.0
1
2
3
4
5
6
7
8
init_points
1
3
2