University of Colorado
Boulder

HW6

Jesse Hettleman
APPM 4600

College of Engineering and Applied Science
Department of Applied Math

October 11, 2024

1 Problem 1

In this problem, we will use two quasi Newton methods–Lazy Newton (Chord) and Broyden–in addition to the original Newton method to solve the following system:

$$f(x,y) = x^2 + y^2 - 4 = 0 \quad (1)$$
$$g(x,y) = e^x + y - 1 \quad (2)$$

Using the three initial guesses $[(1,1),(1,-1),(0,0)]$, we find that in general Newton has the shortest run time, although Broyden is quite close. For the first initial guess of $(1,1)$, both Newton and Broyden converged to the root $(-1.81626407, 0.8373678)$, whereas Lazy Newton reached the maximum number of iterations and failed to converge. For this initial guess, Newton took 0.000316 seconds to converge on average, while Broyden took 0.000468 seconds on average. Newton took 7 iterations, while Broyden took 12. So even though Broyden took more iterations, it was more efficient per iteration.

For the initial guess of $(1,-1)$ we see similar results. This time all three methods converged to the root $(1.00416874, -1.72963729)$. On average, this took Newton 0.000213 seconds, 0.000485 seconds for Lazy Newton, and 0.000224 for Broyden. It also took 5, 36, and 6 iterations respectively.

The final initial guess $(0,0)$ did not converge for any of these methods due to the fact that the Jacobian matrix evaluated at this point is not invertible.

In conclusion, due to the fact that the Broyden method is more efficient per iteration, I would argue that its performance is better than the Newton method. This would be especially true if the system of equations was larger and it was more difficult to compute the inverse of the Jacobian each iteration.

All code for this problem can be found in 3

2 Problem 2

In this problem, we are asked to approximate solutions to the following nonlinear system using Newton's method, Steepest descent method, and a hybrid of the two:

$$x + cos(xyz) - 1 = 0 \quad (3)$$
$$(1-x)^{1/4} + y + 0.05z^2 - 0.15z - 1 = 0 \quad (4)$$
$$-x^2 - 0.1y^2 + 0.01y + z - 1 = 0 \quad (5)$$

Using $(-.5, 1.5, .5)$ as the initial guess results in the following performance metrics for each method:

Figure 1: Estimates, Average Runtimes, and Iterations by Method

```
Newton Method:

[-4.04339648e-17  1.00000000e-01  1.00000000e+00]
Newton: the error message reads: 0
Newton: took this many seconds: 0.00015019893646240235
Netwon: number of iterations is: 4

Steepest Descent Method:

[8.080086373e-05 1.00021341e-01 1.00001962e+00]
Steepest: g evaluated at this point is: 6.9064621420546366e-09
Steepest: the error message reads: 0
Steepest: took this many seconds: 0.0009596967697143555
Steepest: number of iterations is: 7

Hybrid Method:

[-4.15791533e-17  1.00000000e-01  1.00000000e+00]
Hybrid: the error message reads: 0
Hybrid: took this many seconds: 0.00032196044921875
Hybrid: number of iterations is: 5
```

Evidently, Newton's method took the least amount of time to run on average, as well as the fewest iterations. However, this may just be a circumstantial result based on the specific initial guess. If the initial guess was outside of the basin of convergence, I hypothesize that the hybrid method would take the least amount of time and/or iterations because the steepest descent method could help get the guess closer to the basin of convergence before applying the Newton method.

Further, I hypothesize that we see the steepest descent method taking the longest because it can only have linear convergence at best. This is analogous to how the bisection method can also only be linear at best. On the other hand, the Newton method has quadratic convergence, which is why the Newton and Hybrid methods take fewer iterations and less time overall.

3 Appendix

3.1 Code

```
import numpy as np
import math
import time
from numpy.linalg import inv
from numpy.linalg import norm

# -----Question 1-----

def evalF1(x):

```

```
    F = np.zeros(2)

    F[0] = (x[0]**2) + (x[1]**2) - 4
    F[1] = math.e**x[0] + x[1] - 1

    return F

def evalJ1(x):

    J = np.array([[2*x[0], 2*x[1]],
        [math.e**x[0], 1]])
    return J


def Newton1(x0,tol,Nmax):

    ''' inputs: x0 = initial guess, tol = tolerance, Nmax = max its'''
    ''' Outputs: xstar= approx root, ier = error message, its = num ...
        its'''

    for its in range(Nmax):
        J = evalJ1(x0)
        Jinv = inv(J)
        F = evalF1(x0)

        x1 = x0 - Jinv.dot(F)

        if (norm(x1-x0) < tol):
            xstar = x1
            ier =0
            return[xstar, ier, its]

        x0 = x1

    xstar = x1
    ier = 1
    return[xstar,ier,its]

def LazyNewton1(x0,tol,Nmax):

    ''' Lazy Newton = use only the inverse of the Jacobian for ...
        initial guess'''
    ''' inputs: x0 = initial guess, tol = tolerance, Nmax = max its'''
    ''' Outputs: xstar= approx root, ier = error message, its = num ...
        its'''

    J = evalJ1(x0)
    Jinv = inv(J)
    for its in range(Nmax):

        F = evalF1(x0)
        x1 = x0 - Jinv.dot(F)

        if (norm(x1-x0) < tol):
            xstar = x1
            ier =0
            return[xstar, ier,its]

```

```
        x0 = x1

    xstar = x1
    ier = 1
    return[xstar,ier,its]

def Broyden1(x0,tol,Nmax):
    '''tol = desired accuracy
    Nmax = max number of iterations'''

    '''Sherman-Morrison
   (A+xy^T)^{-1} = A^{-1}-1/p*(A^{-1}xy^TA^{-1})
    where p = 1+y^TA^{-1}Ax'''

    '''In Newton
    x_k+1 = xk -(G(x_k))^{-1}*F(x_k)'''


    '''In Broyden
    x = [F(xk)-F(xk-1)-\hat{G}_k-1(xk-xk-1)
    y = x_k-x_k-1/||x_k-x_k-1||^2'''

    ''' implemented as in equation (10.16) on page 650 of text'''

    '''initialize with 1 newton step'''

    A0 = evalJ1(x0)

    v = evalF1(x0)
    A = np.linalg.inv(A0)

    s = -A.dot(v)
    xk = x0+s
    for  its in range(Nmax):
        '''(save v from previous step)'''
        w = v
        ''' create new v'''
        v = evalF1(xk)
        '''y_k = F(xk)-F(xk-1)'''
        y = v-w;
        '''-A_{k-1}^{-1}y_k'''
        z = -A.dot(y)
        ''' p = s_k^tA_{k-1}^{-1}y_k'''
        p = -np.dot(s,z)
        u = np.dot(s,A)
        ''' A = A_k^{-1} via Morrison formula'''
        tmp = s+z
        tmp2 = np.outer(tmp,u)
        A = A+1./p*tmp2
        ''' -A_k^{-1}F(x_k)'''
        s = -A.dot(v)
        xk = xk+s
        if (norm(s)<tol):
            alpha = xk
            ier = 0
            return[alpha,ier,its]
    alpha = xk
    ier = 1
```

```
    return[alpha,ier,its]

def question1():

    print('Question 1:\n')

    f = lambda x,y: 3*(x**2) - (y**2)
    g = lambda x,y: 3*x*(y**2) - (x**3) - 1

    fs = np.array([f,g])

    x0s = np.array([[1, 1],[1, -1],[0, 0]])

    Nmax = 100
    tol = 1e-10

    for x0 in x0s:

        print('\nMethods for initial guess:',x0,'\n')

        if x0[0] == 0 and x0[1] == 0:
            print("ERROR: Initial Guess Causes Non-Invertible ...
                Jacobian\n")
            break

        t = time.time()
        for j in range(50):
            [xstar,ier,its] =  Newton1(x0,tol,Nmax)
        elapsed = time.time()-t
        print(xstar)
        print('Newton: the error message reads:',ier)
        print('Newton: took this many seconds:',elapsed/50)
        print('Netwon: number of iterations is:',its)

        t = time.time()
        for j in range(20):
            [xstar,ier,its] =  LazyNewton1(x0,tol,Nmax)
        elapsed = time.time()-t
        print(xstar)
        print('Lazy Newton: the error message reads:',ier)
        print('Lazy Newton: took this many seconds:',elapsed/20)
        print('Lazy Newton: number of iterations is:',its)

        t = time.time()
        for j in range(20):
            [xstar,ier,its] = Broyden1(x0, tol,Nmax)
        elapsed = time.time()-t
        print(xstar)
        print('Broyden: the error message reads:',ier)
        print('Broyden: took this many seconds:',elapsed/20)
        print('Broyden: number of iterations is:',its)

question1()

# -----Question 2-----

def evalF2(x):

```

```
    F = np.zeros(3)

    F[0] = x[0] + math.cos(x[0]*x[1]*x[2]) - 1
    F[1] = (1-x[0])**(1/4) + x[1] + 0.05*(x[2]**2) - 0.15*x[2] - 1
    F[2] = -(x[0]**2) - 0.1*(x[1]**2) + 0.01*x[1] + x[2] - 1
    return F

def evalJ2(x):


    J = np.array([[1 - x[1]*x[2]*math.sin(x[0]*x[1]*x[2]), ...
        (-1)*x[0]*x[2]*math.sin(x[0]*x[1]*x[2]), ...
        (-1)*x[0]*x[1]*math.sin(x[0]*x[1]*x[2])],
         [(-1/4)*(1-x[0])**(-3/4), 1, 0.1*x[2] - 0.15],
         [-2*x[0], -0.2*x[1] + 0.01, 1]])
    return J


def Newton2(x0,tol,Nmax):

    ''' inputs: x0 = initial guess, tol = tolerance, Nmax = max its'''
    ''' Outputs: xstar= approx root, ier = error message, its = num ...
        its'''

    for its in range(Nmax):
        J = evalJ2(x0)
        Jinv = inv(J)
        F = evalF2(x0)

        x1 = x0 - Jinv.dot(F)

        if (norm(x1-x0) < tol):
            xstar = x1
            ier =0
            return[xstar, ier, its]

        x0 = x1

    xstar = x1
    ier = 1
    return[xstar,ier,its]

def evalg(x):

    F = evalF2(x)
    g = F[0]**2 + F[1]**2 + F[2]**2
    return g

def eval_gradg(x):
    F = evalF2(x)
    J = evalJ2(x)

    gradg = np.transpose(J).dot(F)
    return gradg

def SteepestDescent(x,tol,Nmax):

    count = 0
```

```
    for its in range(Nmax):
        g1 = evalg(x)
        z = eval_gradg(x)
        z0 = norm(z)

        if z0 == 0:
            print("zero gradient")
        z = z/z0
        alpha1 = 0
        alpha3 = 1
        dif_vec = x - alpha3*z
        g3 = evalg(dif_vec)

        while g3≥g1:
            alpha3 = alpha3/2
            dif_vec = x - alpha3*z
            g3 = evalg(dif_vec)

        if alpha3<tol:
            print("no likely improvement")
            ier = 0
            count += 1
            return [x,g1,ier,count]

        alpha2 = alpha3/2
        dif_vec = x - alpha2*z
        g2 = evalg(dif_vec)

        h1 = (g2 - g1)/alpha2
        h2 = (g3-g2)/(alpha3-alpha2)
        h3 = (h2-h1)/alpha3

        alpha0 = 0.5*(alpha2 - h1/h3)
        dif_vec = x - alpha0*z
        g0 = evalg(dif_vec)

        if g0≤g3:
            alpha = alpha0
            gval = g0

        else:
            alpha = alpha3
            gval =g3

        x = x - alpha*z

        if abs(gval - g1)<tol:
            ier = 0
            count += 1
            return [x,gval,ier,count]

        count += 1

    print('max iterations exceeded')
    ier = 1
    return [x,g1,ier,count]
```

```
def Hybrid(x0,tol,Nmax):

    steepest_return = SteepestDescent(x0,5e-2,Nmax)
    steepest_guess = steepest_return[0]

    newton_return = Newton2(steepest_guess,tol,Nmax)
    its = steepest_return[3] + newton_return[2]

    return [newton_return[0],newton_return[1],its]

def question2():

    print('Question 2:\n')

    x0 = np.array([-.5, 1.5, .5])

    Nmax = 100
    tol = 1e-6

    print("\nNewton Method:\n")

    t = time.time()
    for j in range(50):
        [xstar,ier,its] =  Newton2(x0,tol,Nmax)
    elapsed = time.time()-t
    print(xstar)
    print('Newton: the error message reads:',ier)
    print('Newton: took this many seconds:',elapsed/50)
    print('Netwon: number of iterations is:',its)

    print("\nSteepest Descent Method:\n")

    t = time.time()
    for j in range(50):
        [xstar,gval,ier,its] = SteepestDescent(x0,tol,Nmax)
    elapsed = time.time()-t
    print(xstar)
    print("Steepest: g evaluated at this point is:", gval)
    print("Steepest: the error message reads:", ier )
    print('Steepest: took this many seconds:',elapsed/50)
    print('Steepest: number of iterations is:',its)

    print("\nHybrid Method:\n")

    t = time.time()
    for j in range(50):
        [xstar,ier,its] =  Hybrid(x0,tol,Nmax)
    elapsed = time.time()-t
    print(xstar)
    print('Hybrid: the error message reads:',ier)
    print('Hybrid: took this many seconds:',elapsed/50)
    print('Hybrid: number of iterations is:',its)

    print('\n')



question2()
```

NOTE: ALL CODE FOR THIS ASSIGNMENT IS ORIGINAL, WRITTEN BY JESSE HETTLE-MAN