APPM 4600 Homework 6

11 October 2024

1. The code used in this question is listed at the end of the question.

   We have the system

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

   with Jacobian

   $$J(x,y) = \begin{bmatrix} 2x & 2y \\ e^x & 1 \end{bmatrix}.$$

   (a) We apply Newton's method to the system with various initial guesses and get convergence to within 10^{-10} in the following number of iterations:
   - i. $(x_0, y_0) = (1,1)$: Root $(x,y) \approx (-1.81626407, 0.8373678)$, 7 iterations, 0.00025 s.
   - ii. $(x_0, y_0) = (1,-1)$: Root $(x,y) \approx (1.00416874 - 1.72963729)$, 5 iterations, 0.00019 s.
   - iii. $(x_0, y_0) = (0,0)$: Jacobian is singular at the origin, so we cannot apply Newton's method.

   (b) We apply Lazy Newton's method to the system with various initial guesses and get the following results:
   - i. $(x_0, y_0) = (1,1)$: Does not converge to the root (walks off to infinity).
   - ii. $(x_0, y_0) = (1,-1)$: Root $(x,y) \approx (1.00416874 - 1.72963729)$, 36 iterations, 0.000050 s.
   - iii. $(x_0, y_0) = (0,0)$: Jacobian is singular at the origin, so we have no initial J_0.

   (c) We apply Broyden to the system with various initial guesses and get convergence to within 10^{-10} in the following number of iterations:
   - i. $(x_0, y_0) = (1,1)$: Root $(x,y) \approx (-1.81626407, 0.8373678)$, 12 iterations, 0.00018 s.
   - ii. $(x_0, y_0) = (1,-1)$: Root $(x,y) \approx (1.00416874 - 1.72963729)$, 6 iterations, 0.00024 s.
   - iii. $(x_0, y_0) = (0,0)$: Jacobian is singular at the origin, so we have no initial B_0.

   In general, we observe that the Quasi-Newton methods take more iterations to converge but converge in the same or shorter time as compared to Newton's method. This makes sense — Newton's method has better convergence order than the quasi-Newton methods but has a relatively expensive iterative step (it requires a matrix inversion).

   Since our system has only 2 dimensions, matrix inversion is still relatively cheap. This effect may be seen better in higher dimensional systems.

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

def driver():

    x0 = np.array([1e-9, 1e-9])

    Nmax = 100
    tol = 1e-6

    t = time.time()
    for j in range(50):
      [xstar,ier,its] =  Newton(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] =  LazyNewton(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] = Broyden(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)

def evalF(x):

    F = np.zeros(2)

    F[0] = x[0]**2 + x[1]**2 - 4
    F[1] = np.exp(x[0]) + x[1] - 1
    return F

def evalJ(x):
    J = np.array([[2*x[0], 2*x[1]], [np.exp(x[0]), 1]])
    return J


def Newton(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 = evalJ(x0)
       Jinv = inv(J)
       F = evalF(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 LazyNewton(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 = evalJ(x0)
    Jinv = inv(J)
    for its in range(Nmax):
        F = evalF(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 Broyden(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 = evalJ(x0)

    v = evalF(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 = evalF(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]


if __name__ == '__main__':
    # run the drivers only if this is called from the command line
    driver()
```

2. The code used for this question is listed at the end of the question.

We have the system

$$\begin{aligned} 0 &= f(x,y,z) = x + \cos(xyz) - 1 \\ 0 &= g(x,y,z) = (1-x)^{\frac{1}{4}} + y + 0.05z^2 - 0.15z - 1 \\ 0 &= h(x,y,z) = -x^2 - 0.1y^2 + 0.01y + z - 1. \end{aligned}$$

The system has Jacobian

$$J(x,y,z) = \begin{bmatrix} 1 - yz\sin(xyz) & -xz\sin(xyz) & -xy\sin(syz) \\ -\frac{1}{4}(1-x)^{-\frac{3}{4}} & 1 & 0.1z - 0.15 \\ -2x & -0.2y + 0.01 & 1 \end{bmatrix}.$$

(a) Applying Newton's method and an initial guess of $(x_0, y_0, z_0) = (0,0,0)$ yields the root $(x,y,z) = (0, 0.1, 1)$ in 3 iterations, taking 0.00022s.

(b) To apply steepest descent, we want to minimize the function

$$\phi(x,y,z) = f(x,y,z)^2 + g(x,y,z)^2 + h(x,y,z)^2.$$

All roots of our system are also minimums of ϕ.

We have that the gradient of ϕ is

$$\nabla\phi(x,y,z) = 2J^T(x,y,z) \cdot \begin{bmatrix} f(x,y,z) \\ g(x,y,z) \\ h(x,y,z) \end{bmatrix},$$

so our iterative step is

$$\mathbf{x_{k+1}} = \mathbf{x_k} - \lambda_k J^T(\mathbf{x_k}) \cdot \mathbf{F}(\mathbf{x_k}),$$

where we pick λ_k via line search.

Applying steepest descent with an initial guess of $(x_0, y_0, z_0) = (0,0,0)$ yields the approximate root $(x,y,z) \approx (-0.0000628, 0.099968563, 0.0.9999844)$ in 5 iterations, taking 0.00093s.

(c) We apply steepest descent to within $5e-2$ and use the resulting value as a starting point for Newton. This yields an approximate root $(x,y,z) \approx (5.44 \times_1 0^{-17}, 0.1, 1)$ in 1 iteration of steepest descent and 2 iterations of Newton, taking 0.0021s.

This hybrid method is attractive because steepest descent can converge in areas outside of Newton's basin of convergence, but it still allows for the quadratic convergence of Newton nearby the root (as compared to steepest descent's linear convergence).

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

def question6_2():

    x0 = np.array([0, 0, 0])

    Nmax = 100
    tol = 1e-6
    switch_tol = 5e-2
```

```
    t = time.time()
    for j in range(50):
      [xstar,ier,its] =  Newton(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(50):
      [xstar,g1,ier,its] =  SteepestDescent(x0,tol,Nmax)
    elapsed = time.time()-t
    print(xstar)
    print('SD: the error message reads:',ier)
    print('SD: took this many seconds:',elapsed/50)
    print('SD: number of iterations is:',its)

    t = time.time()
    for j in range(50):
      [xstar,ier,its] =  hybrid(x0,tol,switch_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)


def evalF(x):

    F = np.zeros(3)

    F[0] = x[0] + np.cos(x[0] * x[1] * x[2]) - 1
    F[1] = (1-x[0])**(0.25) + 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 evalJ(x):
    J = np.array([[1-x[1]*x[2]*np.sin(x[0] * x[1] * x[2]),-x[0]*x[2]*np.sin(x[0] * x[1] * x[2]), -x
    [-0.25*(1-x[0])**(-0.75), 1, 0.1*x[2] - 0.15],
    [-2*x[0], -0.2*x[1] + 0.01, 1]])
    return J

def evalPhi(x):
    f = evalF(x)
    return f[0]**2 + f[1]**2 + f[2]**2

def evalGradPhi(x):
    return np.transpose(evalJ(x)).dot(evalF(x))


###############################
### steepest descent code

def SteepestDescent(x,tol,Nmax):

    for its in range(Nmax):
        g1 = evalPhi(x)
        z = evalGradPhi(x)
```

```
        z0 = norm(z)

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

        while g3>=g1:
            alpha3 = alpha3/2
            dif_vec = x - alpha3*z
            g3 = evalPhi(dif_vec)

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

        alpha2 = alpha3/2
        dif_vec = x - alpha2*z
        g2 = evalPhi(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 = evalPhi(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
            return [x,gval,ier,its]

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


def Newton(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 = evalJ(x0)
       Jinv = inv(J)
```

```
        F = evalF(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]

# Hybrid steepest descent followed by Newton
def hybrid(x0, tol, switch_tol, Nmax):
    [x1, g1, ier1, its1] = SteepestDescent(x0, switch_tol, Nmax)
    if not ier1 == 0:
        return [x1, ier1, its1]
    [x2, ier2, its2] = Newton(x1, tol, Nmax)
    return [x2, ier2, its2]

question6_2()
```