Zainab Al-Suwaykit
Daniel González Esparza

AMCS CS 212
Numerical Optimization
Assignment 3

1. Minimal Surface. Consider the problem of generating a shape with minimal surface area. The shape is a surface of revolution about a vertical axis of length l (see figure below). Because of circular symmetry, the problem reduces to finding the radii of horizontal cross sections. The radii of the cross sections are fixed on the top and bottom at r_0. We can discretize the problem so the unknowns x represent the radii at n locations equally spaced along the vertical axis at a distance h = l/(n + 1). With this discretization the surface area may be approximated by the following function:

Write a routine that computes the objective function.

```
def fun(x,h):
    n=len(x)
    s = 2 * pi *h *(x[1]*(1+((x[1]-x[0])/h)**2)**(-1/2) +np.sum( x[1:n-1]
       * np.array(1+ np.array(((x[1:n-1]-x[2:n])/h))**2)**(1/2)))
    return s
```

Write a routine that computes the gradient.

```
def grad(x,h):
    n=len(x)-2
    grad = np.zeros((n,1))
    diff_b=((x[1:n+1] - x[0:n])/h)
    diff_a=((x[1:n+1] - x[2:n+2])/h)
    fd = x[0:n] * (( (1+(diff_b)**2)**(-1/2))*  (diff_b/h))
    sd =  (1+diff_a**2)**(1/2) + x[1:n+1] * ((1+diff_a**2)**(-1/2)) *
       (diff_a/h)
    grad=(fd+sd)
    return grad
```

Write a routine that computes the Hessian. Notice that the objective function consists of the sum of terms where each term contains only a pair of adjacent variables. This gives rise to a Hessian that has a tridiagonal structure.

Zainab Al-Suwaykit
Daniel González Esparza

```
def hess(x,h):
    n=len(x)-2
    hess = np.zeros((n,n))
    diff_b=((x[1:n+1] - x[0:n])/h)
    diff_a=((x[1:n+1] - x[2:n+2])/h)
    dd=((2*x[1:n+1] - x[2:n+2])/h**2)
    D=x[0:n]*(((1+diff_b**2)**(1/2))/h**2-(((1+diff_b**2)**(-1/2))*
(diff_b/h)**2))/(1+diff_b**2)\
        +(dd*(1+diff_a**2)**(1/2)   -x[1:n+1]*(1+diff_a**2)**(-1/2)*
(diff_a/h)**2)/(1+diff_a**2)\
            +(1+diff_a**2)**(-1/2)*(diff_a/h)

    Du=x[1:n+1]*( (-1/h**2)*(1+diff_a**2)**(1/2)+((1+diff_a**2)**(-1/2)*
(diff_a/h)**2))/(1+diff_a**2)\
        +((1+diff_a**2)**(-1/2))*(diff_a/-h)
    for i in range(0,n-1):
        hess[i,i+1]=Du[i]
        hess[i+1,i]=Du[i]
        hess[i,i]=D[i]
    hess[n-1,n-1]=D[n-1]
    return 2*pi*h*hess
```

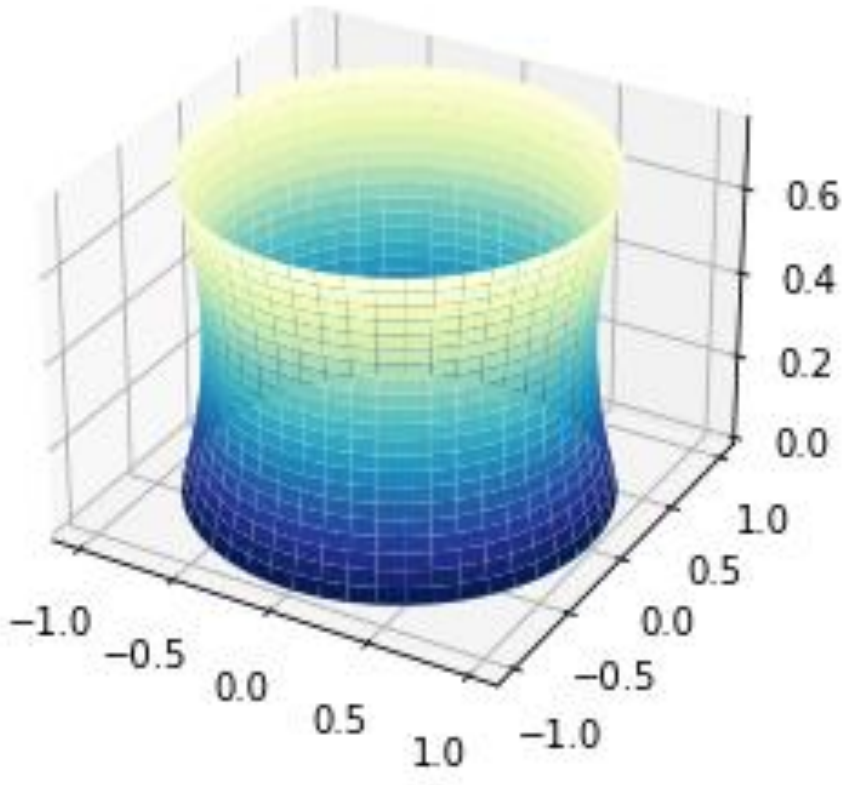
Use your Newton routine with a backtracking line search to find the optimal shape starting from a cylindrical initial shape (i.e. $x_i = r_0$ for all i).

```
def backtrack_r(fun,grad, p,xk,h,alpha,beta):
    print('back')
    t=1
    n=len(xk)
    while(min(xk[1:n-1]+t*p)<0):
        t *=beta
    xk1=np.array(xk)
    xk1[1:n-1]+=t*p
    while(((fun(xk1,h)> fun(xk,h) + alpha* t * (np.transpose(grad) @
p)))):
        t *=beta
        xk1=np.array(xk)
        xk1[1:n-1]+=t*p
    return t
```

Zainab Al-Suwaykit
Daniel González Esparza

```
def newtont_btr(fun, grad, hess, x0,h, tol):
    x=x0
    print('Newton')
    n=len(x)
    Fs=np.array(fun(x,h))
    E=1
    while(E> tol):
        p= la.solve(hess(x,h),-grad(x,h))
        t = backtrack_r(fun,grad(x,h),p,x,h,alpha=0.05,beta=0.9)
        x[1:n-1]+=t*p
        Fs=np.vstack((Fs,fun(x,h)))
        E=np.absolute (Fs[-1]-Fs[-2])
        print('k= ',len(Fs),fun(x,h),E,min(x))
    return Fs,x
```

We tested our code with the parameters given in the homework. The results show a suitable behavior since that we generated a graphic and has the expected properties as symmetry and, a diameter reduction in the center of the domain.

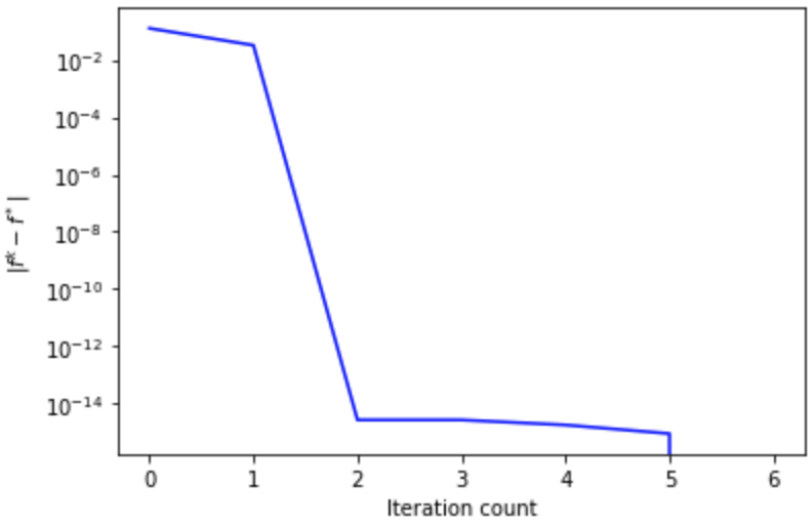


Zainab Al-Suwaykit
Daniel González Esparza

```
r0 = 1
l = 0.75
n = 20
h = l/(n+1)
x= r0+np.zeros((n+2,1))

Fs,xr=newtont_btr(fun, grad, hess, x,h, tol = 1e-20)
theta=np.linspace(0,2*pi,50)
R, P = np.meshgrid(xr, theta)
z=np.zeros((len(R),len(xr)))
z[0:]=np.linspace(0,l,len(xr))
X, Y = R*np.cos(P), R*np.sin(P)
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.plot_surface(X, Y, z, cmap=plt.cm.YlGnBu_r)
```

Plot the convergence behavior.

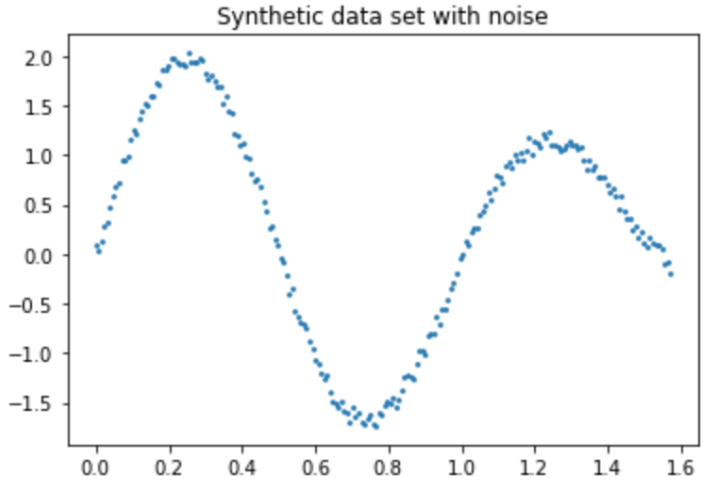
```
Fs_graph=np.zeros((len(Fs)-1,1))
for i in range (0,len(Fs)-1):
    Fs_graph[i]=Fs[i+1]-Fs[i]

plt.figure('Convergence 2')
plt.semilogy(np.arange(0,len(Fs_graph)), np.absolute(Fs_graph),'b')
plt.semilogy(0,Fs_graph[0],'ro')
plt.xlabel('Iteration count')
plt.ylabel(r'$|f^k - f^*|$')
plt.show()
```



Zainab Al-Suwaykit
Daniel González Esparza

2. Gauss-Newton. Nonlinear Least Squares (NLLS) problems arise in a large number of practical scientific and engineering contexts and represent an important class of numerical optimization problems. NLLS problems commonly arise when trying to fit a model to a measured data set in a way that minimizes the discrepancy between model predictions and measured data. The simplest objective to minimize is the sum of the squares of the discrepancies at the measured data points.

```
#HW3-2
import matplotlib.pyplot as plt
m = 200
t = np.linspace(0, pi/2, m)
p = lambda t: 0.8*np.sin(2*pi*1.15*t) + 1.2*np.sin(2*pi*0.9*t)
noise = 0.2 * np.random.uniform(-0.5, 0.5, m)
y = p(t) + noise
plt.scatter(t, y, s=3)
plt.title('Synthetic data set with noise')
plt.show()
```



Write a routine that returns the objective function f(a) = 0.5 r(a)^{t}r(a) where r(a) is an m × 1 residual vector with entries $r_i(a) = \varphi(a; t_i) - y_i$.

```
def NLLS_F(r,a,t):

    f = (1/2) *( np.transpose(r(a,t)) @ r(a,t) )
    return f
```

```
def res(a,t):
    a0=a
    r = [None] * 200
    for i in range (0,m):
        r[i] = (a0[0] * np.sin(2*pi*a0[1]*t[i]) + a0[2]* np.sin(2*pi*a0[3]*t[i])) - y[i]
    return r
```

Zainab Al-Suwaykit
Daniel González Esparza

• **Write a routine that returns the gradient of the objective function $\nabla f(a) = J(a)^t r(a)$ where $J_{m\times n}$ is the Jacobian of r with entries** $J_{ij} = \frac{\partial r_i}{\partial a_j}$.

```
def Jac(a,t,m,n=4):
    jac = np.zeros((m,n))

    for i in range (0,m):
        for j in range (0,n,2):
            jac[i][j]= np.sin(2*pi*a[j+1]*t[i])
            jac[i][j+1] = a[j] * np.cos(2*pi * a[j+1]*t[i]) * 2 *pi * t[i]

    return jac
```

```
def NLLS_grad(Jac,a,m,t,res, n=4):
    grad = [None] * 200
    grad = np.transpose(Jac(a, t, m,n)) @ res(a,t)
    return grad
```

Write a routine that returns $H(a) = J(a)^tJ(a)$, the Gauss-Newton approximation of the Hessian $\nabla^2 f(a)$ of the objective function.

```
def NLLS_hess(Jac,a,m,t,n=4):
    hess = np.zeros((n,1))
    hess = np.transpose(Jac(a, t, m,n)) @ Jac(a, t, m,n)
    return hess
```

Use a Gauss-Newton method to fit the data set above using the starting point $a^0 = [1.0\ 1.0\ 1.0\ 1.0]^T$
.

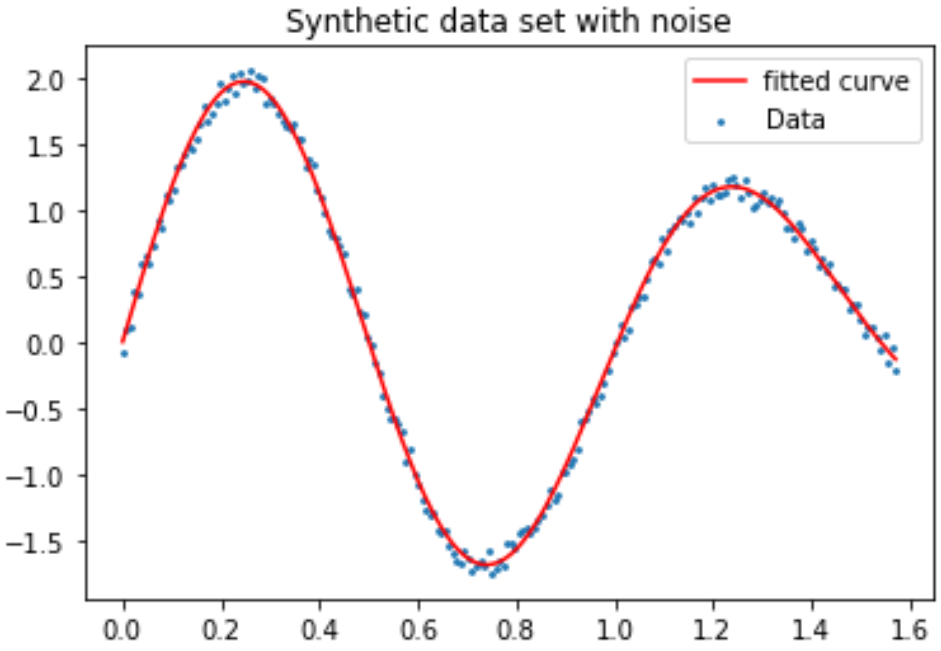
```
from numpy import linalg as la
def gauss_newton(f, grad,Jac, hess, a,res,m,t,n=4, tol = 1e-5):
    Xs=np.array([a])
    while(la.norm(grad(Jac,a,m,t,res,n=4))> tol):

        p= la.solve(hess(Jac,a,m,t,n=4),-1*grad(Jac,a,m,t,res,n=4))
        print(p)
        st = backtrack(f,grad(Jac,a,m,t,res,n=4),p,a,t,res,n=4,alpha=0.1,beta=0.9)
        print(st)
        mul = (st*p)
        b=np.add(a,mul)
        a=b
        Xs=np.vstack((Xs,a))
        print('k= ',len(Xs))#,la.norm(grad(Jac,a,m,t,res,n=4)))
    return Xs
```

Zainab Al-Suwaykit
Daniel González Esparza

```
def backtrack(fun,grad, p,xk,t,res,n=4,alpha=0.1,beta=0.9):
    st=1
    #print(alpha* st * (np.transpose(grad) @ p))
    while(fun(res,xk + st*p,t)> fun(res,xk,t) + alpha* st * (np.transpose(grad) @ p)):
        st *=beta
        #print(st)
    return st
```

```
Fs=gauss_newton(NLLS_F, NLLS_grad,Jac, NLLS_hess, a0,res,m,t,n=4, tol = 1e-5)
```

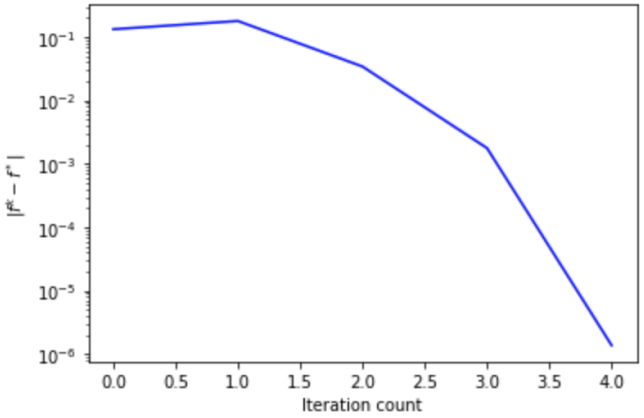
As result, we obtained the constants that fit to the disperse points, this is a proof that the newton method was applied perfectly to the proposed problem, which can be easily used in our current experiments for fit data to a curve.



Finally, we observe the hoped behavior in our convergence plot, which seems to have a quadratic curvature. Moreover, the iteration number to reach the convergence so small.

Zainab Al-Suwaykit
Daniel González Esparza

```
#print(Fs)
Fs_graph=np.zeros((len(Fs)-1,1))
for i in range (0,len(Fs)-1):
    Fs_graph[i]=min(Fs[i+1]-Fs[i])
    #print(Fs_graph)
    #print(i)
plt.figure('Convergence 2')
plt.semilogy(np.arange(0,len(Fs_graph)), np.absolute(Fs_graph),'b')
plt.semilogy(0,Fs_graph[0],'ro')
plt.xlabel('Iteration count')
plt.ylabel(r'$|f^k - f^*|$')
plt.show()
```



3. Quasi-Newton Methods. When the Hessian of the objective is not available, is too expensive, or is too cumbersome to compute, an approximation of it may be obtained by a low-rank update at every iteration. The resulting methods are known as quasi-Newton methods. Among the many quasi-Newton methods, the BFGS method is perhaps the most popular.

Describe very briefly the insight into this formula and how we might derive it.
The quasi-Newton method is based on the approximation of the hessain. All we need is Sk, the differnece between the last and current step, and Yk, the differnce between the gradient at the last step and the current gradient. The formula is obtained by simply applying the Sherman–Morrison–Woodbury formula to the hessian approximation, which means that is not necessary to calculate a hessian in every step since we could do it only adding a pair of increments to a actual hessian and save some computational work.

From a computational perspective, it is often convenient to store and update the inverse (or the Cholesky factors) of this approximate Hessian since this is what is used in computing the quasi-Newton direction. What kind of computational savings can we obtain as a result? Describe (in a few words) how this formula is obtained.

Zainab Al-Suwaykit
Daniel González Esparza

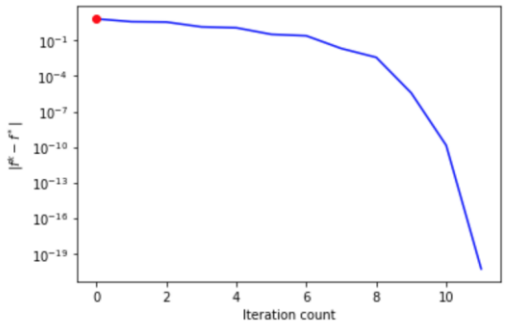
The first thing is that we do not have to compute the second derivatives, which means that we can save computational work by estimating our hessian by information about the change of gradient in every iteration. Moreover, using this method we are taking advantage of the Newton method increasing the convergence rate (quadratic approach) with same order of calculation that steepest descent method. Therefore, the more convergence rate and lower calculation order, less computational work.

Implement a BFGS with a backtracking line search and compare its performance to that of Newton's direction on the Rosenbrock function of the previous homework.

```
def app_hess(sk,yk,B_inv,rho):
    A=((np.transpose(sk)@yk + ((np.transpose(yk) @ B_inv)@ yk)) * (sk@np.transpose(sk))) / ((np.transpose(sk)@yk)**2
    B=- ((B_inv@ (yk@np.transpose(sk)) + (sk@np.transpose(yk))@B_inv))/ (np.transpose(sk)@yk)
    B_inv+=A+B
    return B_inv

def Q_backtrack(fun,grad, p,xk,alpha,beta):
    t=1
    while(fun(xk + t*p)> fun(xk) + alpha* t * (grad @ p)):
        t *=beta
    return t
```

```
def Quasi_newton_bt(f, grad, app_hess, x0, tol = 1e-5):
    x=x0
    Xs=np.array([x0])
    B_inv= np.identity(2)
    sk = np.zeros((2,1))
    yk = np.zeros((2,1))
    c=0
    while(la.norm(grad(x))> tol):
        p= B_inv @(-1 * np.transpose(grad(x)))
        t = Q_backtrack(f,grad(x),p,x,alpha=0.1,beta=0.9)
        x+=t*p
        Xs=np.vstack((Xs,x))
        sk[:,0]= x - Xs[-2]
        yk[:,0]= grad(x) - grad(Xs[-2])
        rho=1/np.transpose(yk)@sk
        if (rho<0):
            B_inv= np.identity(2)
            c+=1
        else:
            B_inv = app_hess(sk,yk,B_inv,rho)
    return Xs,c

x0= np.transpose(np.array([-1.2,1.0]))
Ys,c=Quasi_newton_bt(fun, rosen_grad,app_hess,x0,tol = 1e-5)
```

Zainab Al-Suwaykit
Daniel González Esparza



Newton Method

Quasi-Newton Method

Use the last three iterates to estimate the rate of convergence of BFGS on this problem. Does the result make sense?

From the equation 1:

$$\left| f^{k+1} - f^{*} \right| = C \left| f^{k+1} - f^{*} \right|^{\alpha}$$

It can be easily show that:

$$\alpha = \frac{\log\left(\frac{\left| f^{n} - f^{*} \right|}{\left| f^{n-1} - f^{*} \right|} \right)}{\log\left(\frac{\left| f^{n-1} - f^{*} \right|}{\left| f^{n-2} - f^{*} \right|} \right)}$$

Using the last three values of our solution we have:

$$\alpha = 2.0511$$

In this specific case, we abtained the hoped value of convergence rate, however, the convergence depends on the backtrack method and with different parameters in the backtrack method, we found values of canvergence rate for the last three steps between 1 and 7.