Zainab Al-Suwaykit
Daniel González Esparza

AMCS CS 212
Numerical Optimization
Assignment 2

1. Hessian Computation.
Compute the second derivatives of the Rosenbrock function analytically, and use them to write a function that returns the Hessian of the Rosenbrock function:

```
def rosen_grad(x):

    x1=x[0]
    x2=x[1]
    grad=[0,0]
    grad[1]=20*(x2-x1**2)
    grad[0]=-40*x[0]*(x[1] - x[0]**2) + 2*x[0] - 2
    return np.transpose(grad)
```

```
#Hessian Matrix
def rosen_hessian(x):
    x1=x[0]
    x2=x[1]
    hx = np.array([[-40*(x2-x1**2)+(80*x1**2)+2, -40*x1],
                   [-40*x1, 20]])
    return hx
```

2. Newton's Method with Backtracking Line Search. Write a routine for a backtracking line search method using the Newton direction instead of the steepest direction of problem 3 of the last assignment.

```
def backtrack(fun,grad, p,xk,alpha,beta):

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

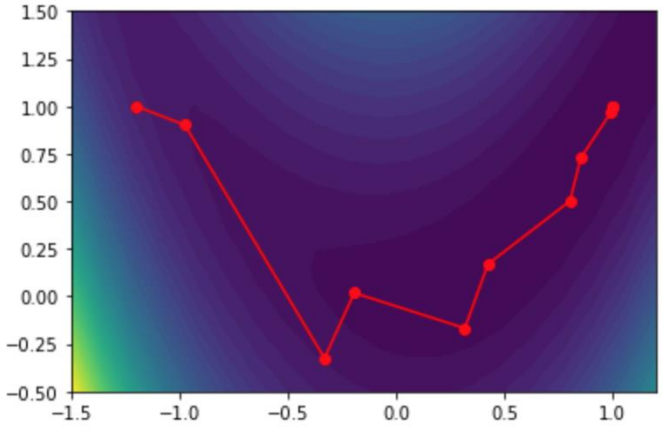
```
def newtont_bt(f, grad, hess, x0, tol = 1e-5):
    x=x0
    Xs=np.array([x0])
    while(la.norm(grad(x))> tol):
        p= la.solve(hess(x),-1*grad(x))
        t = backtrack(f,grad(x),p,x,alpha=0.1,beta=0.9)
        x+=t*p
        Xs=np.vstack((Xs,x))
    return Xs
```

Zainab Al-Suwaykit
Daniel González Esparza

3. Performance of Newton's Method.

Test the performance of the algorithm on Rosenbrock's function $f(x) = 10(x_2 - x^2_1)^2 + (1 - x_1)^2$ starting at the point $[-1.2\ 1.0]^t$ by finding the number of iterations till convergence to a gradient norm of 10^{-5}.

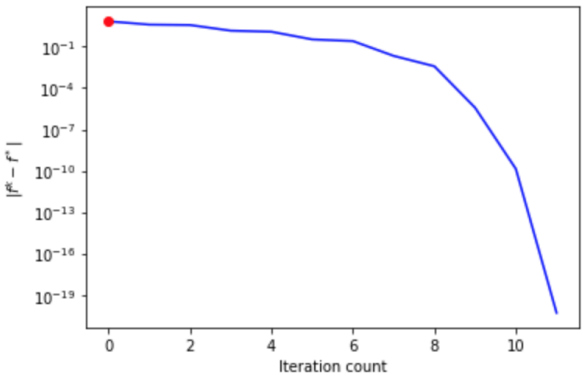
```
x0= np.transpose(np.array([-1.2,1.0]))

Xs=newtont_bt(fun, rosen_grad,rosen_hessian,x0,tol = 1e-5)
n =len(Xs)
plt.plot(Xs[:,0],Xs[:,1],'r-o')
ax=plt.contourf(x1,x2,f,50,linestyles='solid')
plt.show()
```



• Generate a semilog plot of $|f^k - f^*|$ vs. iteration count.

```
f=np.zeros(n)
for i in range(n):
    f[i]=fun(Xs[i])
plt.figure('convergence')
plt.semilogy(np.arange(0, n), np.absolute(f),'b')
plt.semilogy(0,f[0],'ro')
plt.xlabel('Iteration count')
plt.ylabel(r'$|f^k - f^*|$')
plt.show()
```

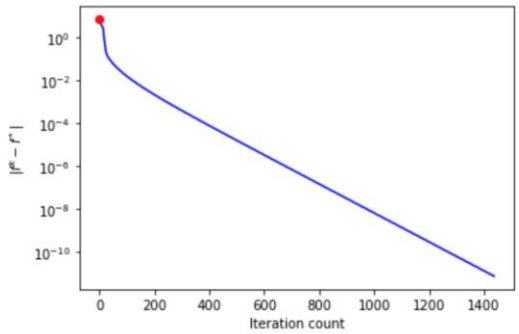
Zainab Al-Suwaykit
Daniel González Esparza



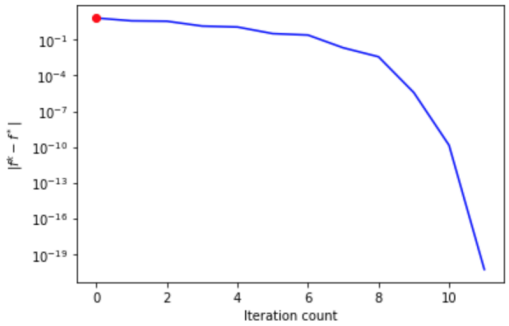
• Write an expression for the expected convergence behavior of the algorithm near the minimum

$|fk\ +1 - f^*| < C\ |fk - f^*|^2$

• Comment on the convergence plot, and compare it to that of the steepest descent direction



Steepest descent



Newton method

The convergence rate of the newton method is much faster than the steepest descent. It takes around 12 iterations for the function to converge in the newton method, compared to the 1500 iterations in the steepest descent and this is because the newton method uses the quadratic approximation.

Zainab Al-Suwaykit
Daniel González Esparza

4. Problem in R^{100}.

Consider the problem of minimizing the following function with $x \in R^{100}$ and a_i given vectors.

- Derive expressions for the gradient and Hessian that would be needed for using Newton's method.

```
A = (np.random.randint(50, size=(100, 500)))/50
def fr(x,A):
    s1 =0
    s2=0
    for i in range (0,500):
        s1 += np.log(1-np.transpose(A[:,i])@x)
        #if(1-np.transpose(A[:,i])@x<0):
            #print(i)
    for j in range (0,100):
        #if(1-x[j]**2):
            #print(j)
        s2 += np.log(1-x[j]**2)
    f = (-1*s1) - s2
    return f

def grad_r(x,A):
    grad=np.array(x)
    for j in range(0,100):
        s1 =0
        for i in range (0,500):
            s1 += (A[j,i])/ (1-np.transpose(A[:,i]) @ x)
        s1 += (2*x[j])/ (1- x[j]**2)
        grad[j] = s1
    return grad
```

```
def hessian_r(x,A):
    hessian=np.zeros((100,100))
    j=0;
    while(j<100):
        for k in range(j,100):
            for i in range(0,500):
                if k==j:
                    hessian[j,k]=hessian[j,k]+A[j,i]**2/(1-A[:,i]@x)**2
                else:
                    hessian[j,k]=hessian[j,k]+A[j,i]*A[k,i]/(1-A[:,i]@x)**2
            if k==j:
                hessian[j,k]=hessian[j,k]+2*(x[j]**2+1)/(1-x[j]**2)**2
            else:
                hessian[k,j]=hessian[j,k]
        j+=1
    return hessian
```

Zainab Al-Suwaykit
Daniel González Esparza

- Write the system of linear equations that needs to be solved at every iteration for finding the Newton direction.

  p= la.solve(hessian_r(x,A),-1*grad_r(x,A))

```
def newtont_btr(fr, grad_r, hessian_r, x0,A, tol = 1e-5):
    print('Newton')
    x=x0
    Fs=np.array(fr(x,A))
    plt.figure('convergence 500x100')
    while(la.norm(grad_r(x,A))> tol):
        p= la.solve(hessian_r(x,A),-1*grad_r(x,A))
        t = backtrack_r(fr,grad_r(x,A),p,x,A,alpha=0.1,beta=0.9)
        print(t,'t')
        x+=t*p
        Fs=np.vstack((Fs,fr(x,A)))
        print('k= ',len(Fs),fr(x,A), la.norm(grad_r(x,A)))
    return Fs
```

- The backtracking line search routine we wrote earlier must be modified so that it does not return a step length that places x outside the region where the objective function is defined, i.e., $\min_i(1-a^t_i x)$ and $\min_i(1-x^2_i)$ must be positive. We call this condition a feasibility condition.
- Modify backtrack so that it first decreases the step length until the feasibility condition is satisfied, and then continues decreasing it further until the admissibility condition is satisfied.

```
def backtrack_r(fun,grad, p,xk,A,alpha,beta):
    t=1
    xk1=xk+t*p
    while((min(1-np.transpose(A)@xk1)<0) or (min(1-xk1*xk1)<0)):
        t *=beta
        xk1=xk+t*p
    while(((fun(xk + t*p,A)> fun(xk,A) + alpha* t * (np.transpose(grad) @ p)))):
        t *=beta
    return t
```

- Generate a random coefficient matrix A (100×500) whose columns serve as the vectors a_i. You may use, for example, the numpy function randn. Solve the problem starting from an initial (feasible) point. Plot the convergence behavior.

Zainab Al-Suwaykit
Daniel González Esparza

```
x=(np.random.randint(98, size=(100, 1))-49)/50
print(len(x))
print(min(1-np.transpose(A)@x))
print(min(1-x*x))
while((min(1-np.transpose(A)@x)<0) or (min(1-x*x)<0)):
    x=(np.random.randint(98, size=(100, 1))-49)/50
Fs=newtont_btr(fr, grad_r, hessian_r, x,A, tol = 1e-5)
```

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