AE 6310-A: Assignment #3

Karl Roush

Due April 07, 2020; Submission up to April 14, 2020 for no penalty

Contents

Problem 1

Part A: Finding constrained optimum and Lagrange multipliers

The function is defined as follows

$$f:\ (x_1+2)^2 + 10(x_2+3)^2$$

$$c:\ {x_1}^2 + {x_2}^2\ \leq\ 2$$

Before proceeding further, it is important to point out that the unconstrained minimum of the objective function is [-2, -3]. This point does not satisfy the constraint function, so we know that the constraint must be active.

Note that scipy.minimize requires the gradient as one of the inputs.

$$\nabla f:\ \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 + 4 \\ 20x_2 + 60 \end{bmatrix}$$

Additionally, scipy.minimize assumes the constraints are of the form C(x)>=0 and requires a Jacobian for each constraint. Therefore, the constraint is restructured as

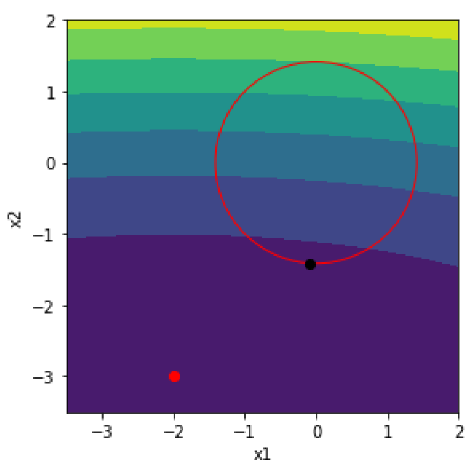
$$c:\ -{x_1}^2 - {x_2}^2 + 2\ \geq\ 0$$

$$A(x) = \begin{bmatrix} \frac{\partial c}{\partial x_1} & \frac{\partial c}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -2x_1 & -2x_2 \end{bmatrix}$$

Additionally, scipy.minimize only returns the constrained optimum (x*). In order to calculate the Lagrange multipliers, we can use the following, derived from the KKT conditions:

$$\nabla f(x*) = -A(x*)^T \lambda \ \Rightarrow\ \lambda = \frac{\mathrm{A}(\mathrm{x}*)\nabla f(x*)}{-A(x*)A(x*)^T}$$

Note that if using A(x) from the scipy definition, lamda will be negative so be sure to multiply by -1. Using this method, we find the **constrained min to be [-0.1619, -1.4049], Lagrange multiplier= 11.3536**



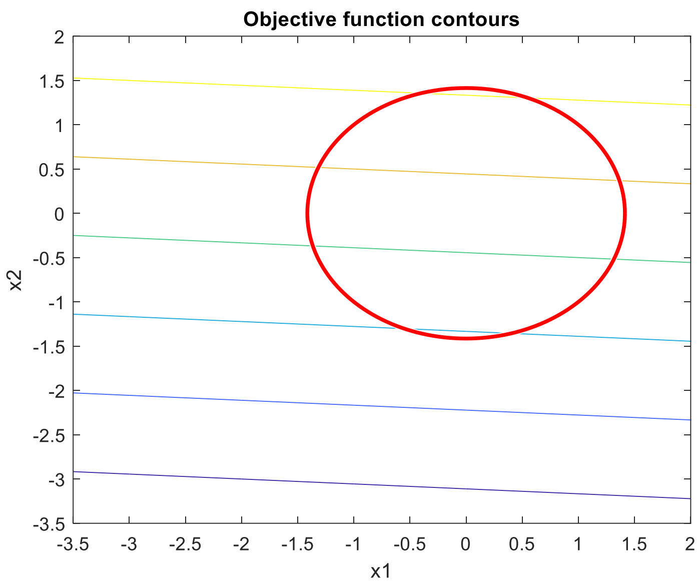
Constraint boundary =red circle, Unconstrained min= red point , Constrained min= black point

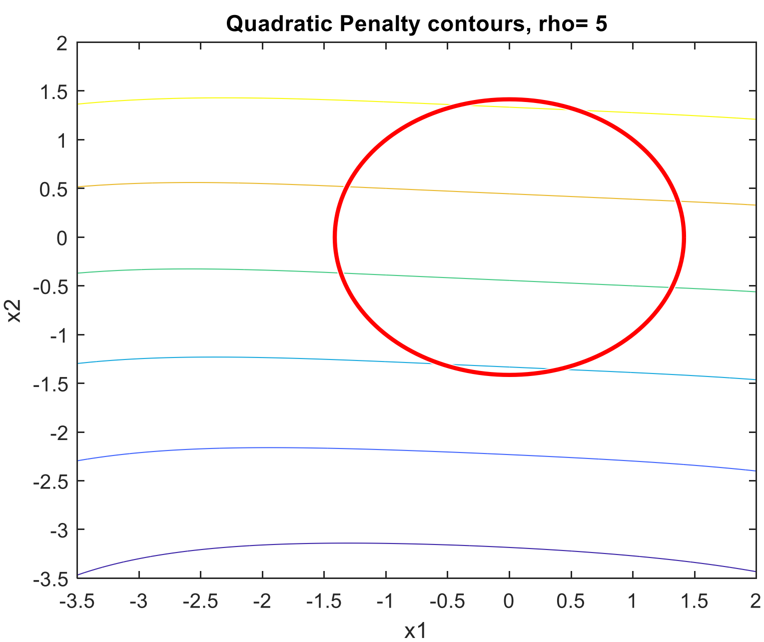
Part B: Quadratic penalty function plots

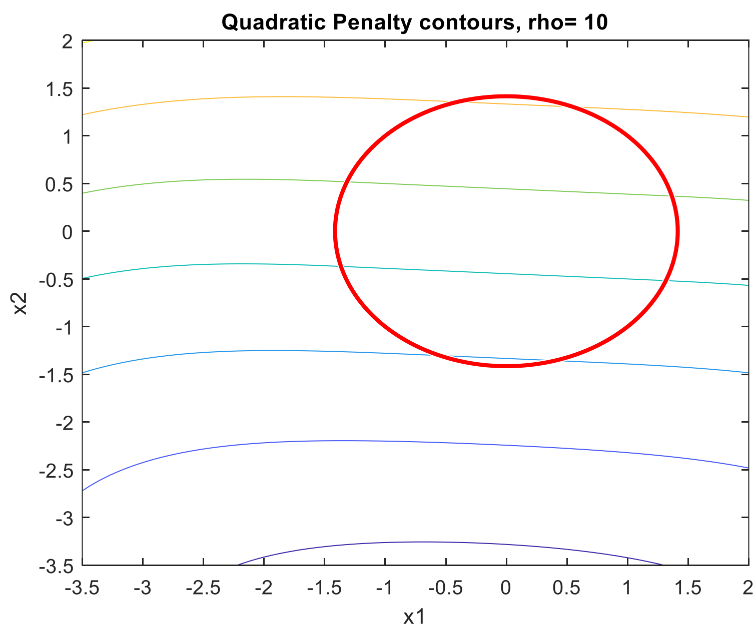
The quadratic penalty is an exterior penalty method, defined everywhere. It modifies the problem to

$$\min_x \left\{ f(x) + \frac{\rho}{2} \sum_i (c_i(x), 0)^2 \right\}$$

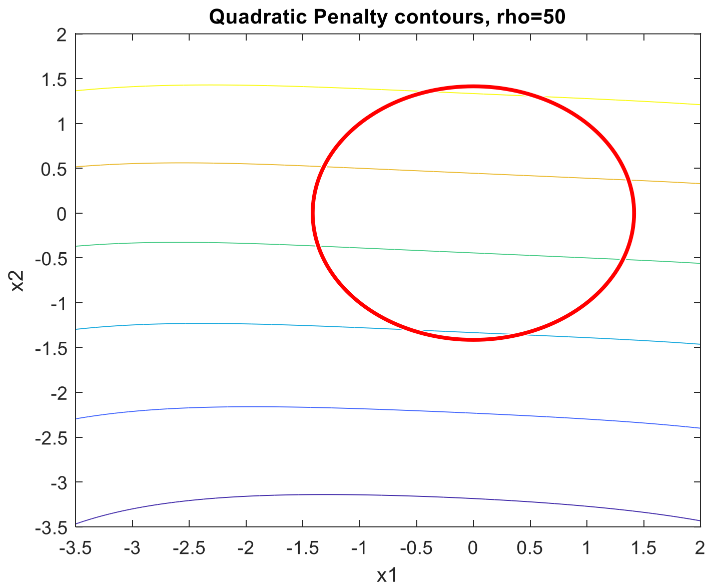
In both of the following plots, the original constraint boundary is marked by a red circle. As rho increases, the minimizer approaches the constrained minimizer from the infeasible space.

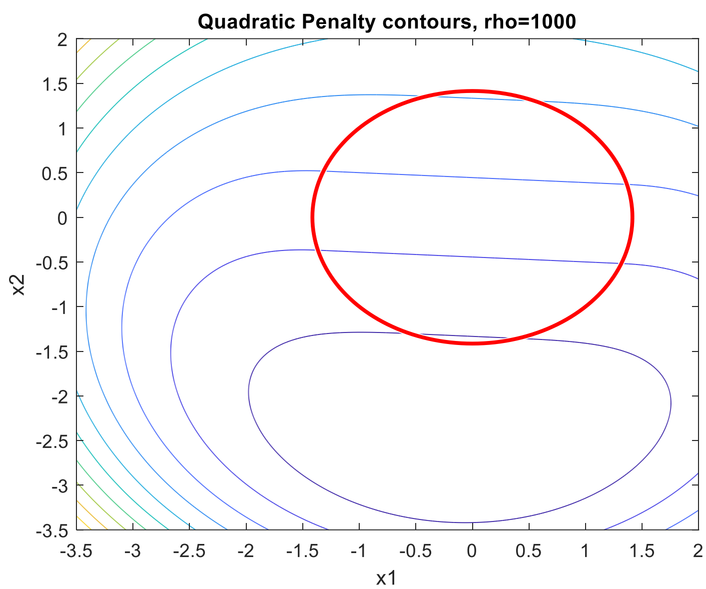






If the contours are hard to view, consider two extreme cases where the difference between the rho is increased significantly. In this case, it is much easier to see as rho increases, the minimizer approaches the constrained minimizer from the *infeasible* space.



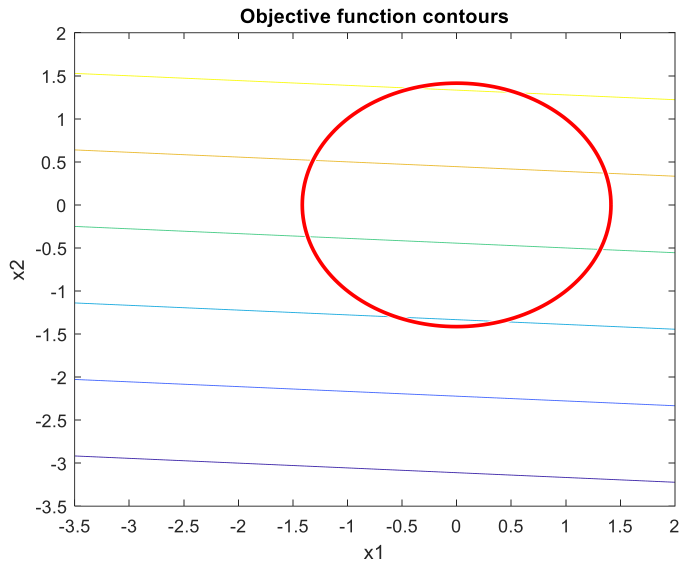


Part C: Log-barrier penalty function plots

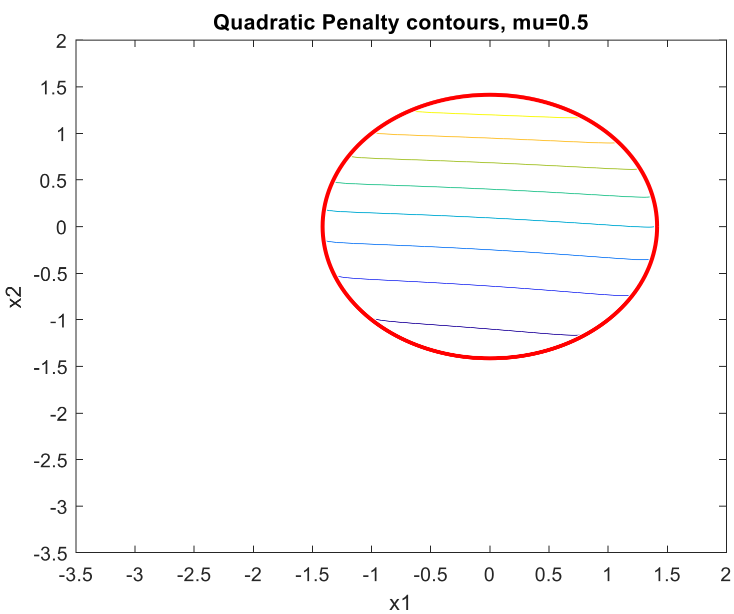
The log-barrier penalty is an interior penalty method, defined only within the feasible space. It modifies the problem to

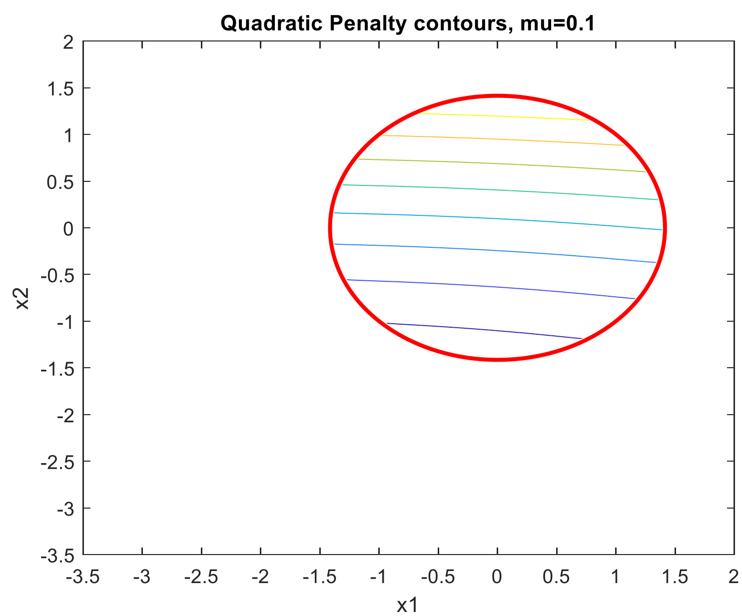
$$\min_{x}\left\{f(x)-\mu\sum_{i}\ln(-c_i(x))\right\}$$

In both of the following plots, the original constraint boundary is marked by a red circle. As mu decreases, the minimizer approaches the constrained minimizer from the feasible space.

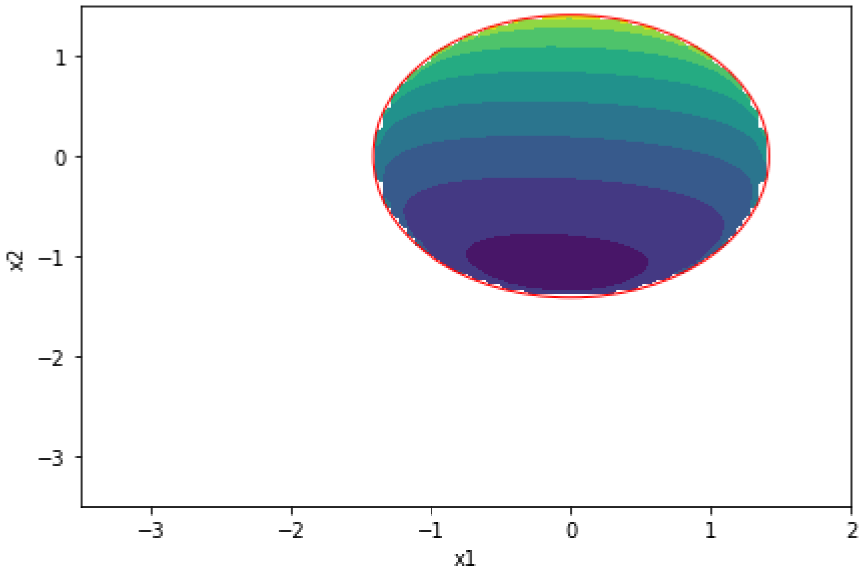


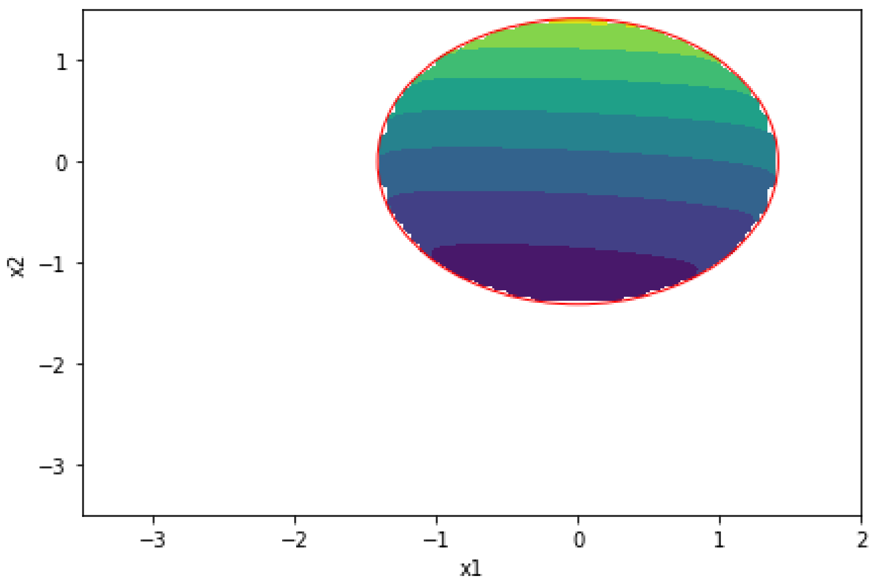
There is a small difference between the mu values, but the contour change is there.





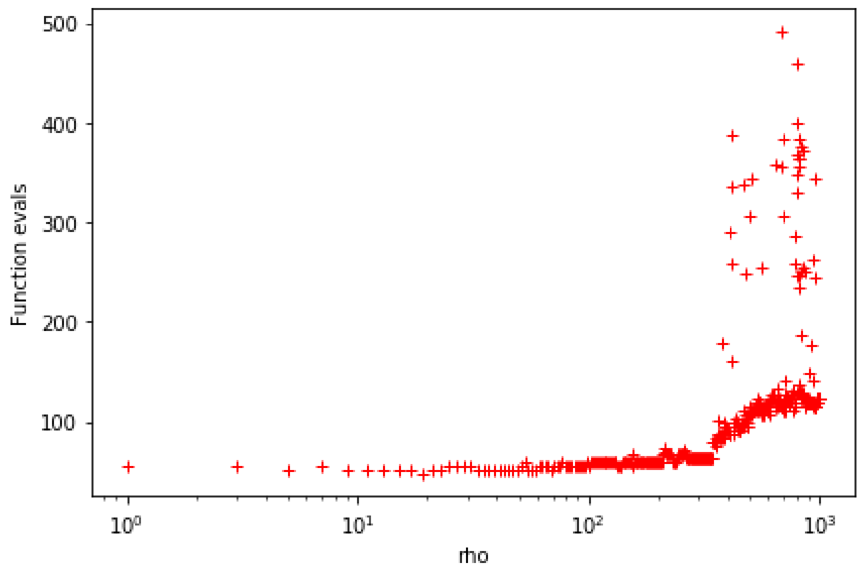
If the contours are hard to view, consider two extreme cases where the difference between the mu is increased significantly. In this case, it is much easier to see as mu decreases, the minimizer approaches the constrained minimizer from the *feasible* space.

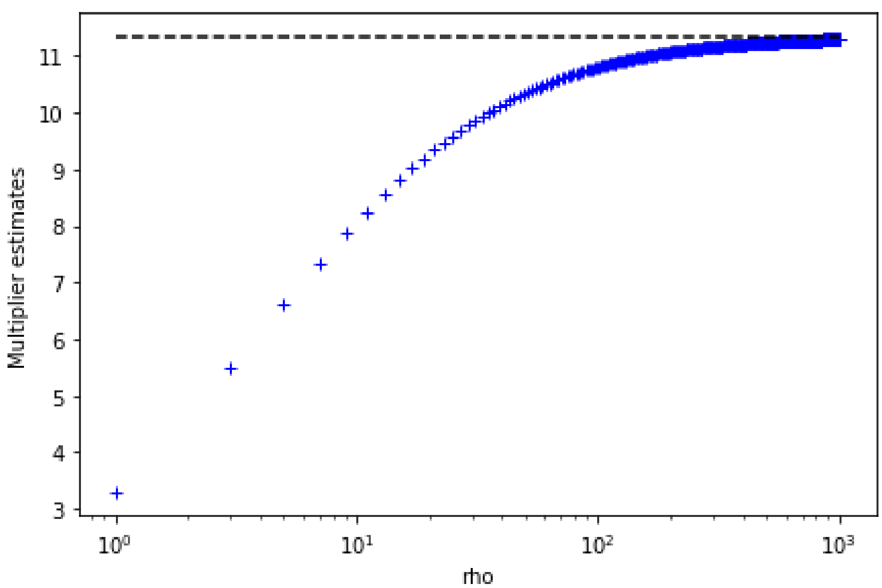




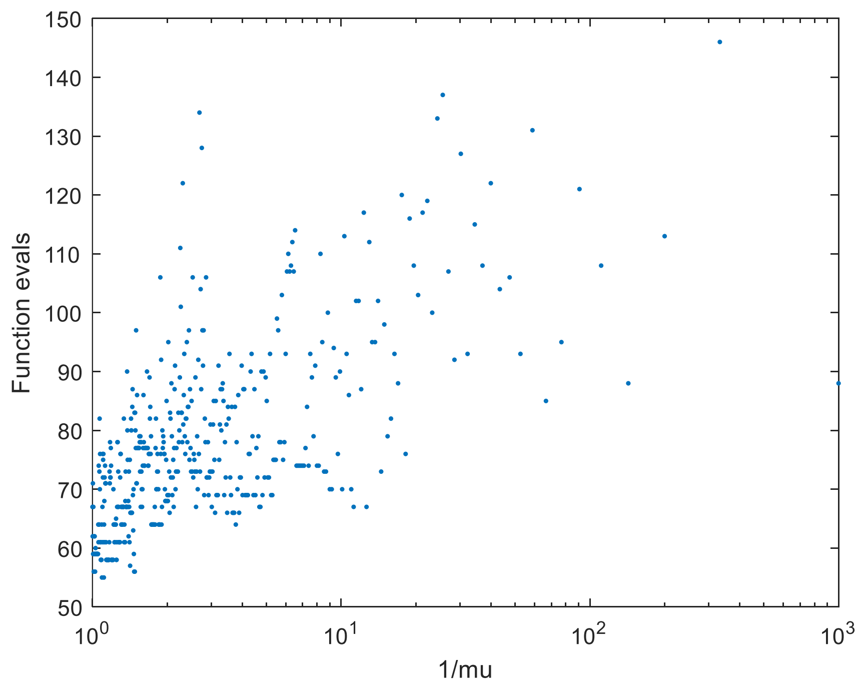
Part D: Plots for varying rho and mu

Varying rho between [1,1000] shows that increasing rho increases the number of function evaluations, but provides a better estimate of the multiplier estimates.

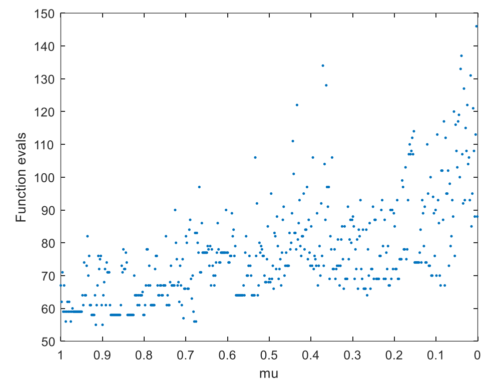


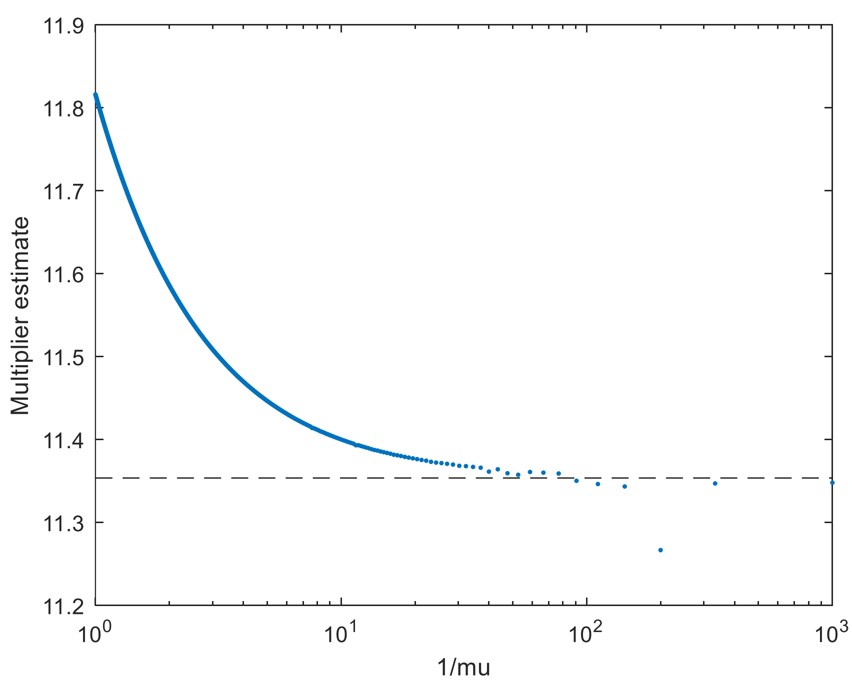


Varying mu between [1,0.001] shows that decreasing mu (i.e. increasing 1/mu) increases the number of function evaluations, but provides a better estimate of the multiplier estimates. Note that MATLAB handles the log-barrier method a bit better than Python.



Linear version of left plot





Problem 2

For this problem, consider the design variables x1, x2, x3 and the state variables u1, u2. The design variables must be positive. The governing equations are given as

$$R(x,u) = \begin{bmatrix} (x_1 + x_2)u_1 + (x_3 - x_2)u_2 - 1 \\ (x_3 - x_2)u_1 + (x_1 + x_2 - x_3)u_2 - 1 \end{bmatrix} = 0$$

$$R(x,u) = \begin{bmatrix} (x_1 + x_2) & (x_3 - x_2) \\ (x_3 - x_2) & (x_1 + x_2 - x_3) \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 0$$

The state variables must have the same sign. The output function of interest is given as

$$f(x,u) = x_1 x_2 + \sqrt{u_1 u_2}$$

The adjoint method makes use of several derivatives which are detailed below.

$$\frac{\partial R}{\partial u} = \begin{bmatrix} (x_1 + x_2) & (x_3 - x_2) \\ (x_3 - x_2) & (x_1 + x_2 - x_3) \end{bmatrix}$$

$$\frac{\partial R}{\partial x} = \begin{bmatrix} \frac{\partial R_1}{\partial x_1} & \frac{\partial R_1}{\partial x_2} & \frac{\partial R_1}{\partial x_3} \\ \frac{\partial R_2}{\partial x_1} & \frac{\partial R_2}{\partial x_2} & \frac{\partial R_2}{\partial x_3} \end{bmatrix} = \begin{bmatrix} u_1 & u_1 - u_2 & u_2 \\ u_2 & u_2 - u_1 & u_1 - u_2 \end{bmatrix}$$

$$\frac{\partial f}{\partial u} = \begin{bmatrix} \frac{u_2}{2\sqrt{u_1 u_2}} & \frac{u_1}{2\sqrt{u_1 u_2}} \end{bmatrix}$$

$$\frac{\partial f}{\partial x} = \begin{bmatrix} x_2 & x_1 & 0 \end{bmatrix}$$

Part A: Forward difference and complex step

Both methods work by evaluating the function at a point then comparing it to another point after taking a step. Calculation of the output function value is done through the following process:

1. Pick a point for the design variables [x1, x2, x3]
2. Solve the governing equations for the state variables [u1, u2]
3. Evaluate the output function of interest

After stepping to the next point and finding the value of the function of interest at that new point, we can then create an estimate for the derivate.

For the forward-difference method, the derivative is estimated as

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h)$$

For the complex step method, the derivative is estimated as

$$\frac{df}{dx} \approx \frac{Im\{f(x + i\,h)\}}{h} + \mathcal{O}(h^2)$$

Note that relative error is defined as

$$\text{relative error} = \frac{|\text{method- adjoint}|}{|\text{adjoint}|}$$

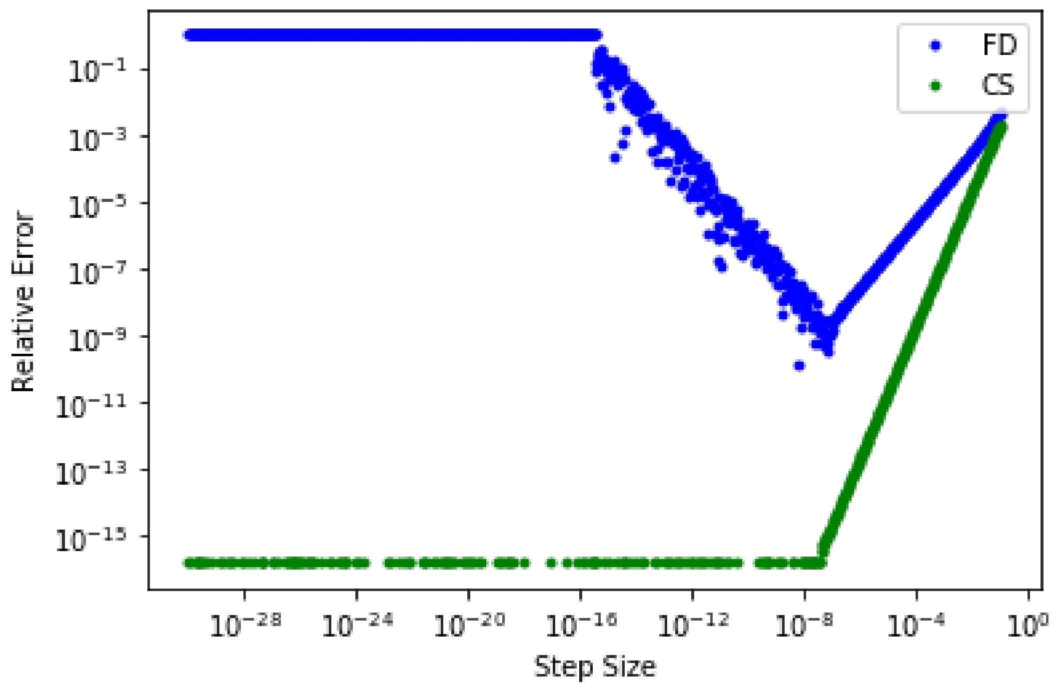
Consider a design point of [1.1, 1.2, 1.3] looking at two different step sizes:

```
Design x0 point: [1.1 1.2 1.3]
Function value:  1.9344649467017594

Step size:  1e-30
Forward-difference approximation:  [0. 0. 0.]
Complex-step approximation:        [ 0.79554704  1.12290251 -0.15157669]
Adjoint based total derivative:    [ 0.79554704  1.12290251 -0.15157669]
Magnitude of relative FD error:    1.0
Magnitude of relative CS error:    1.6038140697726189e-16
------------------------------------------------------------------------
Design x0 point: [1.1 1.2 1.3]
Function value:  1.9344649467017594

Step size:  1e-06
Forward-difference approximation:  [ 0.79554732  1.12290235 -0.15157683]
Complex-step approximation:        [ 0.79554704  1.12290251 -0.15157669]
Adjoint based total derivative:    [ 0.79554704  1.12290251 -0.15157669]
Magnitude of relative FD error:    2.9361459771228685e-08
Magnitude of relative CS error:    2.2966617479143904e-13
```

Now consider a range of different step sizes, keeping the same design point:



Part B: Adjoint and complex step comparison

The adjoint equations are defined as follows

$$(\frac{\partial R}{\partial u})^T \Psi = -(\frac{\partial f}{\partial u})^T$$

$$\begin{bmatrix} (x_1 + x_2) & (x_3 - x_2) \\ (x_3 - x_2) & (x_1 + x_2 - x_3) \end{bmatrix}^T \begin{bmatrix} \Psi_1 \\ \Psi_2 \end{bmatrix} = -\begin{bmatrix} \frac{u_2}{2\sqrt{u_1 u_2}} & \frac{u_1}{2\sqrt{u_1 u_2}} \end{bmatrix}^T$$

The total derivative is then given by

$$\nabla f = \frac{\partial f}{\partial x} + \Psi^T \frac{\partial R}{\partial x}$$

$$\nabla f = [x_2 \quad x_1 \quad 0] + [\Psi_1 \quad \Psi_2] \begin{bmatrix} u_1 & u_1 - u_2 & u_2 \\ u_2 & u_2 - u_1 & u_1 - u_2 \end{bmatrix}$$

Therefore, the function to compute the total derivative using the adjoint method goes through the following set of steps:

1. Pick a point for the design variables [x1, x2, x3]
2. Solve the governing equations for the state variables [u1, u2]
3. Solve for the adjoint variables
4. Take derivative of the interest function with respect to x (design variables)
5. Add the result of step 4 to the transpose of the adjoint variables times the derivative of the governing equations with respect to x (design variables)

Once the total derivative has been computed, we estimate the value at the new point by simply taking the dot product of the perturbation and the total derivative. The relative error between the complex step and adjoint method is simply (adj - cs)/cs where adj and cs are the estimations at the x1 point using the adjoint and complex step methods respectively.

Using a step of 1e-30, the relative error is very small, approximately machine precision. Consider the design point of [1.1, 1.2, 1.3] below:

```
Adjoint-based derivative @x0:     [ 0.79554704  1.12290251 -0.15157669]
Complex-step approximation:       [ 0.79554704  1.12290251 -0.15157669]
Rel error between CS & Adjoint:   [1.39554668e-16 1.97741658e-16 1.83112423e-16]
```

Code snippets

Problem 1A

```python
def p1_ptA():
    # Plot the function with constraint and unconstrained min

    # find unconstrained min
    res = sp.minimize(func, [0,0], method='BFGS', options={'disp':False})
    uncon_min= res.x

    # find unconstrained min
    x0=[0,0]
    cons= ({'type':'ineq',
            'fun': con,
            'jac': con_deriv})
    res = sp.minimize(func, x0, jac=func_deriv, constraints=cons,
                      method='SLSQP', options={'disp':False})
    x_star=res.x
    delta_fStar= res.jac
    l_mult=lagrange_mult(x_star,delta_fStar)
    cplot(func,False,uncon_min,x_star)

    print('Constrained min= ',x_star)
    print('Langrange multiplier= ',l_mult)
```

```python
def func(x):
    #x=[x1,x2]
    return (x[0]+2)**2 +10*(x[1]+3)**2

def quad_penalty(x, *args):

def log_barrier(x, *args):

def func_deriv(x):
    res= np.zeros((1, 2))
    dfdx1= 2*x[0]+4
    dfdx2= 20*x[1]+60
    return np.array([dfdx1,dfdx2])

def func_jac(x):
    return np.array([[df2dx12,df2dx1x2],[df2dx2x1,df2dx22]])

def con(x):
    #x1^2 +x2^2 <= 2
    # scipy wants the form of c(x) >= 0
    #-x1^2 -x2^2 +2 >= 0
    return -x[0]**2 -x[1]**2 +2

def con_deriv(x):
    res= np.zeros((1, 2))
    dcdx1= -2*x[0]
    dcdx2= -2*x[1]
    return np.array([dcdx1,dcdx2])

def lagrange_mult(x_star,delta_fstar):
    A_x= con_deriv(x_star)
    l_mult= np.dot(A_x,delta_fstar)/(-np.dot(A_x,np.transpose(A_x)))*-1
    return l_mult
```

Problem 1B and 1C

```python
def cplot(fobj,line,*args):
    n = 100
    x1 = np.linspace(-3.5, 2, n)
    x2 = np.linspace(-3.5, 1.5, n)
    X1, X2 = np.meshgrid(x1, x2)
    f = np.zeros((n, n))

    if line:
        for j in range(n):
            for i in range(n):
                f[i, j] = fobj([X1[i, j], X2[i, j]])
        fig, ax = plt.subplots(1, 1)
        ax.contour(X1, X2, f)

    else:
    # Query the function at the specified locations
        for i in range(n):
            for j in range(n):
                f[i, j] = fobj([X1[i, j], X2[i, j]])

        fig, ax = plt.subplots(1, 1)
        ax.contourf(X1, X2, f)

    #this is the constraint
    circ = plt.Circle((0, 0), 2**0.5, color='r',fill=False,linestyle='-')
    ax.add_artist(circ)

    if len(args) !=0:
        uncon_min=args[0]
        x_star=args[1]
        ax.plot(uncon_min[0],uncon_min[1],'ro')
        ax.plot(x_star[0],x_star[1],'ko')

    # ax.set_aspect('equal', 'box')
    fig.tight_layout()
    plt.xlabel('x1')
    plt.ylabel('x2')
```

```python
def quad_penalty(x, *args):
    #defined everywhere
    if len(args) !=0:
        rho= args[0]
    else:
        rho= 5
    f_val= func(x) #function value
    c_val= con(x)*-1 #constraint value, have to *-1 to undo scipy
    return f_val+ (0.5*rho)*(max(c_val,0)**2)

def log_barrier(x, *args):
    #only defined in feasible space
    if len(args) !=0:
        mu= args[0]
    else:
        mu= 0.1
    f_val= func(x)
    c_val= con(x)*-1

    if c_val < 0:
        return f_val -(mu)*np.log(-c_val)
    else:
        return float('NaN')
```

```
function quad_graphs(X1,X2)
figure()
rho= 50;
f2= quadPenalty(X1,X2,rho);
contour(X1,X2,f2)
hold on
viscircles([0,0],sqrt(2));
hold off
title_text= strcat('Quadratic Penalty contours, rho= ', num2str(rho));
title(title_text)
xlabel('x1'); ylabel('x2');

figure()
rho= 1000;
f2= quadPenalty(X1,X2,rho);
contour(X1,X2,f2)
hold on
viscircles([0,0],sqrt(2));
hold off
title_text= strcat('Quadratic Penalty contours, rho= ', num2str(rho));
title(title_text)
xlabel('x1'); ylabel('x2');
end
```

```
function log_graphs(X1,X2)
figure()
mu= 0.5;
f2= logBarrier_penalty(X1,X2,mu);
contour(X1,X2,f2)
hold on
viscircles([0,0],sqrt(2));
hold off
title_text= strcat('Quadratic Penalty contours, mu= ', num2str(mu));
title(title_text)
xlabel('x1'); ylabel('x2');

figure()
mu= 0.1;
f2= logBarrier_penalty(X1,X2,mu);
contour(X1,X2,f2)
hold on
viscircles([0,0],sqrt(2));
hold off
title_text= strcat('Quadratic Penalty contours, mu= ', num2str(mu));
title(title_text)
xlabel('x1'); ylabel('x2');
end
```

```
function f= quadPenalty(x1,x2,rho)
f=(x1+2)^2 +10.*(x2+3)^2;
con_val= x1.^2+x2.^2 -2;
dim= size(x1);
zero= zeros(dim(1));
f=f+ (rho/2)*(max(con_val,zero).^2);
end

function f= logBarrier_penalty(x1,x2,mu)
dim= size(x1);
f= ones(dim(1));
for i=1:1:dim(1) %iterate through x1 (cols)
    for j=1:1:dim(1) %iterate through x2 (rows)
        x1_curr= x1(1,i);
        x2_curr= x2(j,1);

        c= x1_curr^2 + x2_curr^2 -2;
        if c<0 %feasible if c <0
           f(i,j)=(x1_curr+2)^2 +10.*(x2_curr+3)^2 +mu*log(-c);
        else
           f(i,j)= NaN;
        end
    end
end
f= f';
end
```

Problem 1D

```python
def var_rho(): #use python
    n= 500
    x1 = np.linspace(-3.5, 2, n)
    x2 = np.linspace(-3.5, 2, n)
    rho= np.linspace(1,1000,n)
    f_evals= []
    lambda_est= []
    x0= [0,0]
    for current_rho in rho:
        res = sp.minimize(quad_penalty, x0, args=current_rho, method='BFGS', options={'disp':False})
        f_evals.append(res.nfev)
        current_L_est= current_rho*con(res.x)*-1
        lambda_est.append(current_L_est)

    fig, ax = plt.subplots(1, 1)
    ax.set_xscale('log')
    ax.plot(rho,f_evals,'r+')
    fig.tight_layout()
    plt.xlabel('rho')
    plt.ylabel('Function evals')

    fig, ax = plt.subplots(1, 1)
    ax.plot(rho,lambda_est,'b+')
    ax.plot(rho, 11.3536*np.ones([1,n])[0],'k--')
    ax.set_xscale('log')
    fig.tight_layout()
    plt.xlabel('rho')
    plt.ylabel('Multiplier estimates')
```

```python
def var_mu(): #use matlab
    n= 500
    x1 = np.linspace(-3.5, 2, n)
    x2 = np.linspace(-3.5, 2, n)
    mu= np.linspace(1,0.001,n)
    f_evals= []
    lambda_est= []
    x0= [0,0]
    for current_mu in mu:
        res = sp.minimize(quad_penalty, x0, args=current_mu, method='BFGS', options={'disp':False})
        f_evals.append(res.nfev)
        current_L_est= -current_mu/(con(res.x)*-1)
        lambda_est.append(current_L_est)

    fig, ax = plt.subplots(1, 1)
    ax.plot(1/mu,f_evals,'r+')
    fig.tight_layout()
    plt.xlabel('1/mu')
    plt.ylabel('Function evals')

    fig, ax = plt.subplots(1, 1)
    ax.plot(1/mu,lambda_est,'b+')
    fig.tight_layout()
    plt.xlabel('1/mu')
    plt.ylabel('Multiplier estimates')
```

```matlab
function quadPenalty_vary() %Use Python
funcEvals= [];
lambda_est= [];
n= 500;

for rho= linspace(1,1000,n)
    fun_quad= @(x) (x(1)+2)^2 +10*(x(2)+3)^2 +...
        (rho/2)*(max(x(1)^2+ x(2)^2 -2,0).^2);
    nonlcon = @cons;
    x0= [-1,-1];
    A= []; b= [];
    Aeq= []; beq= [];
    lb= []; ub= [];
    [x,fval,exitflag,output] = fmincon(fun_quad,x0,A,b,Aeq,beq,lb,ub,nonlcon);
    funcEvals= [funcEvals, output.funcCount];

    c= x(1)^2+ x(2)^2 -2;
    lambda_est= [lambda_est, rho*c];
end
rho= linspace(1,1000,n);
figure()
plot(rho, funcEvals,'+')
xlabel('rho')
set(gca, 'XScale', 'log')
ylabel('Function evals')

figure()
plot(rho, lambda_est,'+')
xlabel('rho')
set(gca, 'XScale', 'log')
ylabel('Multiplier estimate')
end
```

```matlab
function logBarrier_vary() %use MATLAB
funcEvals= [];
lambda_est= [];
n= 500;

for mu= linspace(1,0.001,n)
    fun_quad= @(x) (x(1)+2)^2 +10*(x(2)+3)^2 -...
        mu*log(-1*(x(1)^2+ x(2)^2 -2));
    nonlcon = @cons;
    x0= [-1,0];
    A= []; b= [];
    Aeq= []; beq= [];
    lb= []; ub= [];
    [x,fval,exitflag,output] = fmincon(fun_quad,x0,A,b,Aeq,beq,lb,ub,nonlcon);
    funcEvals= [funcEvals, output.funcCount];

    c= x(1)^2+ x(2)^2 -2;
    lambda_est= [lambda_est, -mu/c];
end
mu= linspace(1,0.001,n);
figure()
plot(1./mu, funcEvals,'.')
xlabel('mu')
% set(gca, 'XScale', 'log')
set(gca, 'xdir','reverse')
ylabel('Function evals')

figure()
plot(1./mu, lambda_est,'.')
hold on
plot(1./mu, 11.3536.*ones(1,n),'k--')
xlabel('1/mu')
set(gca, 'XScale', 'log')
ylabel('Multiplier estimate')
end
```

Problem 2

```python
import numpy as np
import time
import matplotlib.pylab as plt

def evaluate(x):
    """
    Evaluate the function of interest
    Args: x (np.ndarray) Vector of length 3. The design variables
    Return: The value of the function of interest
    """
    # Fill in the values of the governing equation
    # R(x,u) = K(x)*u - F = 0
    K, F = evaluate_governing_eqns(x)
    # Solve the governing equations to obtain u
    u = np.linalg.solve(K, F)

    # Evaluate the function of interest
    f = x[0]*x[1] +np.sqrt(u[0]*u[1])
    return f

def evaluate_governing_eqns(x):
    #0= K(x)*u -F
    K = np.zeros((2, 2), dtype=x.dtype)
    F = np.zeros(2, dtype=x.dtype)

    #x= [x1, x2, x3]
    K[0,0] = x[0] +x[1] #x1+x2
    K[0,1] = x[2] -x[1] #x3-x2
    K[1,0] = x[2] -x[1] #x3-x2
    K[1,1] = x[0] +x[1] -x[2] #x1+x2-x3

    F[0] = 1.0
    F[1] = 1.0
    return K, F
```

```python
def adjoint_total_derivative(x):
    """ Use the adjoint method to evaluate the derivative of the function
    of interest with respect to the design variables.
    Args: x (np.ndarray) Vector of length 3. The design variables
    Return: dfdx (np.ndarray) Vector of length 3. The total derivative
    """
    # Fill in the values of the governing equation
    # R(x,u) = K(x)*u - F = 0
    K, F = evaluate_governing_eqns(x)
    # Solve the governing equations to obtain u
    u = np.linalg.solve(K, F)

    # define dR/du = K
    # define df/du
    dfdu = np.zeros((2), dtype=x.dtype)
    dfdu[0] = u[1]/(2*np.sqrt(u[0]*u[1]))
    dfdu[1] = u[0]/(2*np.sqrt(u[0]*u[1]))
    # Solve for the adjoint variables
    psi = -np.linalg.solve(K.T, dfdu.T)

    # Define dR/dx
    dRdx = np.zeros((2, 3), dtype=x.dtype)
    dRdx[0,0] = u[0] #first column
    dRdx[1,0] = u[1]
    dRdx[0,1] = u[0]- u[1] #column 2
    dRdx[1,1] = u[1]- u[0]
    dRdx[0,2] = u[1] #column 3
    dRdx[1,2] = u[0]- u[1]

    #define dF/dx
    dFdx = np.zeros((1, 3), dtype=x.dtype)
    dFdx[0,0] = x[1]
    dFdx[0,1] = x[0]
    dFdx[0,2] = 0.0

    return dFdx+ np.dot(psi.T, dRdx)
```

```python
def ptA_compare(h):
    # Set the perturbation vector. We perturb the design variables
    pert = np.array([1,1,1])

    # Compute the function of interest at the point x0
    x0 = np.array([1.1, 1.2, 1.3])
    f0 = evaluate(x0)
    print('Design x0 point:', x0)
    print('Function value: ', f0)

    # forward difference approximation
    print ('\nStep size: ',h)
    pert_x1= np.array([1,0,0])
    pert_x2= np.array([0,1,0])
    pert_x3= np.array([0,0,1])

    x1_d= x0+ h*pert_x1
    f1_d= evaluate(x1_d)
    f1d = (f1_d - f0)/h

    x2_d= x0+ h*pert_x2
    f2_d= evaluate(x2_d)
    f2d = (f2_d - f0)/h

    x3_d= x0+ h*pert_x3
    f3_d= evaluate(x3_d)
    f3d = (f3_d - f0)/h
    fd= np.array([f1d, f2d, f3d])
    print('Forward-difference approximation: ', fd)
    fd= np.linalg.norm(fd)

    # complex step approximation
    x1_c = x0 + h*1j*pert_x1
    f1_c = evaluate(x1_c)
    cs_1 = f1_c.imag/h

    x2_c = x0 + h*1j*pert_x2
    f2_c = evaluate(x2_c)
    cs_2 = f2_c.imag/h

    x3_c = x0 + h*1j*pert_x3
    f3_c = evaluate(x3_c)
    cs_3 = f3_c.imag/h

    cs= np.array([cs_1,cs_2,cs_3])
    print('Complex-step approximation:       ', cs)
    cs= np.linalg.norm(cs)

    total_der = adjoint_total_derivative(x0)[0]
    print('Adjoint based total derivative:    ', total_der)
    total_der= np.linalg.norm(total_der)

    error_fd= abs(fd- total_der)/abs(total_der)
    error_cs= abs(cs- total_der)/abs(total_der)
    print('Magnitude of relative FD error:    ', error_fd)
    print('Magnitude of relative CS error:    ', error_cs,'\n')
```

```python
def ptA_graph():
    pert = np.array([1,1,1])
    x0 = np.array([1.1, 1.2, 1.3])
    f0 = evaluate(x0)
    n= 1000
    exp= np.linspace(1,30,n)
    h_list= 1*10**(-exp)
    fd_error=[]
    cs_error= []

    pert_x1= np.array([1,0,0])
    pert_x2= np.array([0,1,0])
    pert_x3= np.array([0,0,1])

    for h in h_list:
        #forward step approx
        x1_d= x0+ h*pert_x1
        f1_d= evaluate(x1_d)
        f1d = (f1_d - f0)/h
        x2_d= x0+ h*pert_x2
        f2_d= evaluate(x2_d)
        f2d = (f2_d - f0)/h
        x3_d= x0+ h*pert_x3
        f3_d= evaluate(x3_d)
        f3d = (f3_d - f0)/h

        fd= np.array([f1d, f2d, f3d])
        fd= np.linalg.norm(fd)

        #complex step approximation
        x1_c = x0 + h*1j*pert_x1
        f1_c = evaluate(x1_c)
        cs_1 = f1_c.imag/h
        x2_c = x0 + h*1j*pert_x2
        f2_c = evaluate(x2_c)
        cs_2 = f2_c.imag/h
        x3_c = x0 + h*1j*pert_x3
        f3_c = evaluate(x3_c)
        cs_3 = f3_c.imag/h

        cs= np.array([cs_1,cs_2,cs_3])
        cs= np.linalg.norm(cs)

        total_der = adjoint_total_derivative(x0)[0]
        total_der= np.linalg.norm(total_der)
```

total_der= np.linalg.norm(total_der)

```python

        fd_error.append(abs(fd- total_der)/abs(total_der))
        cs_error.append(abs(cs- total_der)/abs(total_der))

    fig, ax = plt.subplots()
    ax.plot(h_list,fd_error,'b.')
    ax.plot(h_list,cs_error,'g.')
    ax.set_xscale('log')
    ax.set_yscale('log')

    ax.legend(['FD','CS'],loc='best')
    plt.xlabel('Step Size')
    plt.ylabel('Relative Error')
```

```python
def ptB():
    x0 = np.array([1.1, 1.2, 1.3])

    total_der = adjoint_total_derivative(x0)[0]
    print('Adjoint-based derivative @x0:     ', total_der)

    h= 1e-30
    pert_x1= np.array([1,0,0])
    x1_c = x0 + h*1j*pert_x1
    f1_c = evaluate(x1_c)
    cs_1 = f1_c.imag/h

    pert_x2= np.array([0,1,0])
    x2_c = x0 + h*1j*pert_x2
    f2_c = evaluate(x2_c)
    cs_2 = f2_c.imag/h

    pert_x3= np.array([0,0,1])
    x3_c = x0 + h*1j*pert_x3
    f3_c = evaluate(x3_c)
    cs_3 = f3_c.imag/h

    cs= np.array([cs_1,cs_2,cs_3])
    print('Complex-step approximation:       ', cs)
    diff= abs(cs- total_der)/abs(total_der)
    print('Rel error between CS & Adjoint:   ', diff)
```