```python
'''
Assignment:     hw6, MAE 5930 (Optimization)
File name:      optzn_hw6_JHansen.py
Author:         Jared Hansen
Date created:   11/06/2019
Python version: 3.7.3

DESCRIPTION:
    This Python script is used for answering Problem 5 from hw6 in
    MAE 5930 (Optimization).
'''



#------------------------------------------------------------------------------
#--- IMPORT STATEMENTS --------------------------------------------------------
#------------------------------------------------------------------------------
import numpy as np
import numpy.linalg as npla
import random
# Set a seed to make results reproducible
random.seed(1776)


#------------------------------------------------------------------------------
#--- FUNCTION IMPLEMENTATIONS -------------------------------------------------
#------------------------------------------------------------------------------
def obj_fctn(x):
    """ Takes the point x (R^10 vec) and returns the value of the squared
    (objective) function (this is easier/nicer to minimize).
    """
    return(np.asscalar(np.dot(x.T,x)))

def obj_grad(x):
    """ Takes the point x (R^10 vec) and returns the gradient of the objective
    function. Here this is just the vector scaled by 2.
    """
    return(2*x)

def obj_hess():
    """ Doesn't need to take any arguments: for the objective function ||x||^2
    the Hessian will always be the identity matrix scaled by 2
    with dimensions numRows=numCols=dim(x)
    """
    return( 2 * np.identity(len(x0)) )

def backtrack(f, grd, x, v):
    """
    This function implements the backtracking algorithm.

    Parameters
    ----------
    f     : a mathematical function (the objective function)
    grd   : a mathematical function (the gradient of f)
    x     : a point in the domain of f (either a scalar or a vector/list/array)
    v     : descent direction (a floating point number)
    t     : step size parameter
    alpha : adjustadble floating point number (between 0 and 0.5)
    beta  : float, btwn 0-1, granularity of search (large=fine, small=coarse)

    Returns
    -------
    t : descent direction (floating point number between 0 and 1)
    """
    # Define the starting value of t, and values of alpha and beta
    t = 1.0
    alpha = 0.2
```

```
    beta = 0.5
    # Implement the condition found in the algorithm defintion
    while(f(x + t*v) >
          f(x) + alpha*t*np.asscalar(np.dot(grd(x).T, v)) ):
        t *= beta
        print("t = ", t)
    return(t)

def newtons_method(f, grd, hessian, x0):
    """
    This function implements Newton's method. Set epsiolon = 1e-6.

    Parameters
    ----------
    f       : a mathematical function (the objective function)
    grd     : a mathematical function (the gradient of f)
    hessian : a matrix (numpy matrix; the hessian of f)
    x0      : a feasible starting pt (scalar or an array/list).

    Returns
    -------
    newtons_output: a tuple containing (final_x, f_final, iters)
    final_x : the point in the domain of f that minimizes f (to eta tolerance)
    f_final : the function f evaluated at final_x
    iters   : the number of iterations that it took to achieve the minimum
    """
    # Initialize a variable to count the number of iterations
    iters = 1
    # Declare a variable for the max number of iterations
    MAX_ITERS = 20000
    # Define our tolerance, the constant EPSILON
    EPSILON = 1e-6
    # Initialize the point we're going to update, x, as the starting point x0
    x = x0
    # Calculate Newton's decrement value
    lmb_sq = np.asscalar(np.dot(np.dot(grd(x).T , npla.pinv(hessian())) ,
                             grd(x)))
    # Stay in this while loop until the have sufficiently small lmb_sq
    while((lmb_sq/2.0) > EPSILON):
        # Compute delta_x and Newton's decrement
        dlt_x = np.dot(-npla.pinv(A) , (np.dot(A,x) - b))
        # Line search for t (descent direction)
        t = backtrack(f, grd, x, dlt_x)
        # Update our "more minimizing" x
        x = np.array(x + (t * dlt_x))
        # Print statements to see how the algorithm is doing
        print("f(x)  : ", f(x))
        print("iters : ", iters)
        print()
        # Update lmb_sq
        lmb_sq = np.asscalar(np.dot(np.dot(grd(x).T , npla.pinv(hessian())) ,
                             grd(x)))
        # Update number of iterations
        iters += 1
        # Set up the function to quit after reaching MAX_ITERS
        if(iters > MAX_ITERS):
            print("Reached MAX_ITERS (", MAX_ITERS, ") without converging.")
            break
    # After sufficiently approach EPSILON or reaching MAX_ITERS return a tuple
    # containing the final_x, f_final, and iters
    newt_output = (x, f(x), iters)
    return(newt_output)


#-----------------------------------------------------------------------------
#--- PROCEDURAL CODE ---------------------------------------------------------
#-----------------------------------------------------------------------------
```

```
# Define matrix A, vector b, and starting point x0
A = np.matrix([[ 4,  4,  4,  9, 10, 10,  3,  6,  0,  1],
               [ 7,  5,  6,  9,  5,  7,  1,  1,  5,  8],
               [ 0,  4,  1,  1,  7,  3,  0,  6,  7,  4],
               [ 3,  7,  2,  0,  3,  8,  7,  7,  5,  2],
               [ 1,  2,  8,  2,  7,  1,  2,  1,  9,  9],
               [ 1,  9, 10,  9,  8,  4,  3,  4,  6,  3],
               [ 2,  0,  3,  1,  0,  9,  5,  7,  9,  8],
               [ 3,  7,  7,  4,  8,  3,  1,  4,  1,  7]])
b = np.array([9,6,8,3,3,9,4,10]).reshape(8,1)
# A point x is feasible (per the equality constraint) if [Ax=b].
# Therefore we can simply do [x = A^(-1)*b] to find x, where A^(-1) will be
# the pseudo-inverse of A since A may not be invertible.
x0 = np.dot(npla.pinv(A), b)
# Find the solution using Newton's Method with equality constraint
newt_output = newtons_method(obj_fctn, obj_grad, obj_hess, x0)
# Output the results to the console
print("\n---------------------------------")
print("---- Newton's Method solution ---")
print("---------------------------------")
print("x* Newton's mtd   : ")
print( newt_output[0])
print("f(x*)             : ", newt_output[1])
print("iters to converge : ", newt_output[2])
```