MAE 5930 - Optimization

Fall 2019

Homework 2

Jared Hansen

Due: 11:00 PM, Thursday September 26, 2019

A-number: `A01439768`

e-mail: `jrdhansen@gmail.com`

Purpose: the problems assigned help develop your ability to

- implement numerical algorithms for unconstrained optimization
- solve unconstrained problems using MATLAB's `pinv`, `quadprog`, and `fminunc`.
- solve unconstrained problems using gradient descent and Newton's method.
- calculate derivatives of vector functions.
- solve linear programming problems graphically and with MATLAB's `linprog`.

NOTE: you are welcome to program in MATLAB or Python.

1. Program the backtracking algorithm as described by Boyd on page 464 of *Convex Optimization*. (Note that Boyd considers gradient vectors to be column vectors instead of row vectors.)

   (a) Inputs to the function are f (the objective function), $\nabla f(x)$ (the gradient of f evaluated at the point x), and the descent direction v.

   (b) Within the function set $\alpha = 0.20$ and $beta = 0.50$.

   - For code see **CODE APPENDIX** at end of homework printout (Python code is in black, after the MATLAB code).
   - I have a large comment around PROBLEM 1 so that it's easy to find. The function's name is "**backtrack**."
   - I'll also include a screenshot of it here for your convenience.

```
#--- PART A ---------------------------------------------------------------
# Inputs to the function are the objective function, gradient function
# evaluated at x, the point x, and the descent direction v.
#--- PART B ---------------------------------------------------------------
# Within the function set alpha = 0.2 and beta = 0.5.
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
```

Figure 1: My implementation of the backtracking algorithm (in Python).

2. Program the gradient descent method as described by Boyd on page 466 of *Convex Optimization*. (Note that Boyd considers gradient vectors to be column vectors instead of row vectors.)

   (a) Inputs to the function are f (the objective function), $\nabla f(x)$ (the gradient of f evaluated at the point x), and x_0 (the starting value of x).

   (b) Within the function set ($\eta = 1 \times 10^{-6}$)

   - For code see **CODE APPENDIX** at end of homework printout (Python code is in black, after the MATLAB code).
   - I have a large comment around PROBLEM 2 so that it's easy to find. The function's name is "**grad_desc**."
   - I'll also include a screenshot of it here for your convenience.

```python
def grad_desc(f, grd, x0):
    """
    Parameters
    ----------
    f   : a mathematical function (the objective function)
    grd : a mathematical function (the gradient of f)
    x0  : a starting pt in the domain of f (either a scalar or an array/list)

    Returns
    -------
    grad_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 = 10000
    # Define our tolerance, the constant ETA
    ETA = 1e-6
    # Initialize the point we're going to update, x, as the starting point x0
    x = x0
    while(np.linalg.norm(grd(x)) > ETA):
        # Find step size using backtracking
        t = backtrack(f, grd, x, (-1*grd(x)).T.flatten())
        # Update our "more minimizing" point x
        x = (x - (t * grd(x)).reshape(len(x),))
        #x = x - (t*grd(x))
        # Print statements to see what is going on
        print("f(x)  : ", f(x))
        print("iters : ", iters)
        print()
        # Update the 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 ETA or reaching MAX_ITERS return a tuple
    # containing the final_x, f_final, and iters
    grad_output = (x, f(x), iters)
    return(grad_output)
```

Figure 2: My implementation of the gradient descent method (in Python).

3. Program Newton's Method as described by Boyd on page 487 of *Convex Optimization*. (Note that Boyd considers gradient vectors to be column vectors instead of row vectors.)

   (a) Inputs to the function are f (the objective function), $\nabla f(x)$ (the gradient of f evaluated at the point x), $\nabla^2 f(x)$ (the Hessian of f evaluated at the point x), and x_0 (the starting value of x).

   (b) Within the function set ($\epsilon = 1 \times 10^{-6}$)

      - For code see **CODE APPENDIX** at end of homework printout (Python code is in black, after the MATLAB code).
      - I have a large comment around PROBLEM 3 so that it's easy to find. The function's name is "**newtons_method**."
      - I'll also include a screenshot of it here for your convenience.

```python
def newtons_method(f, grd, hessian, x0):
    """
    Parameters
    ----------
    f   : a mathematical function (the objective function)
    grd : a mathematical function (the gradient of f)
    x0  : a starting pt in the domain of f (either a 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, MAX_TERATIONS
    iters = 1
    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(x))) ,
                                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(hessian(x)) , grd(x))
        # Line search for t (descent direction)
        t = backtrack(f, grd, x, (np.array(dlt_x).flatten()))
        # Update our "more minimizing" x
        x = np.array(x + np.dot(t, dlt_x).reshape(len(x), )).flatten()
        # 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(x))) ,
                                grd(x)))
        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
    # Return the output as a tuple (x*, f(x*), and iterations to converge)
    newt_output = (x, f(x), iters)
    return(newt_output)
```

Figure 3: My implementation of Newton's Method (in Python).

4. Consider the least squares esimation problem wherein we are trying to find the $x \in R^n$ that minimizes the error in the linear equation $Ax = b$. The matrix $A \in R^{m \times n}$ and the vector $b \in \mathbb{R}^m$. Assume that $m > n$ and that $rank(A) = n$.

After defining the errors as $e = Ax - b$, we can write the optimization problem as:

$$\min_x f(x) = \frac{1}{2}e^T e = \frac{1}{2}(Ax - b)^T(Ax - b)$$

(a) Expand the objective function so that it has three terms. Identify the quadratic term, linear term, and constant term.

$$\begin{aligned}
f(x) &= \left[\frac{1}{2}e^T e = \frac{1}{2}(Ax - b)^T(Ax - b)\right] \\
&= \left[x^T A^T Ax - x^T A^T b - b^T Ax - b^T b\right] \\
&= \left[\frac{1}{2}x^T A^T Ax - \left(\frac{1}{2}\right)\left(\frac{2}{1}\right)x^T A^T b - \frac{1}{2}b^T b\right] \\
&= \left[\left(\frac{1}{2}x^T A^T Ax\right) - \left(x^T A^T b\right) - \left(\frac{1}{2}b^T b\right)\right] \\
&= \left[\left(\text{quadratic term}\right) - \left(\text{linear term}\right) - \left(\text{constant term}\right)\right]
\end{aligned}$$

(b) Calculate the gradient vector with respect to x.

- I simply applied rules of derivatives for multiplied vectors/matrices. They can be found here: `https://atmos.washington.edu/ dennis/MatrixCalculus.pdf`.
- $\nabla f = \left[\left(\frac{1}{2}\right)\left(\frac{2}{1}\right)(A^T Ax) - A^T b\right] = \left[A^T Ax - A^T b\right]$

(c) Calculate the Hessian matrix with respect to x.

- Again, I applied the rules at `https://atmos.washington.edu/ dennis/MatrixCalculus.pdf`. to get:
- $\nabla^2 f = \left[A^T A\right]$

(d) Use the first-order necessary condition to find all candidates for a local minimum. Check if the second-order necessary condition is satisfied. Explain.

- First-order necessary condition is: any x^* that minimizes f will satisfy $\nabla f(x^*) = 0$
- Set $[\nabla f = \mathbf{0}] \longrightarrow \left[A^T Ax - A^T b = \mathbf{0}\right] \longrightarrow \left[A^T Ax = A^T b\right] \longrightarrow$ $\left[(A^T A)^{-1}(A^T A)x = (A^T A)^{-1}(A^T b)\right] \longrightarrow \left[x = (A^T A)^{-1}(A^T b)\right]$
- So long as $\left(A^T A\right)$ is invertible (or pseudo-invertible) then $\exists$ an $\left[x = (A^T A)^{-1}(A^T b)\right]$ such that $\nabla f(x) = 0$. (Thus there is only one candidate minimum.)

- Second-order necessary condition is: $\nabla^2 f(x^*) \geq 0$, in this case meaning that the Hessian is PSD (positve semi-definite).

- We are given that A has $\left[rank(A) = n\right] \implies \left[A \text{ is full column rank}\right] \implies$
$\left[\text{ columns of } A \text{ are linearly independent }\right] \implies \left[Av \neq \mathbf{0} \text{ so long as } v \neq \mathbf{0} \text{ itself }\right] \implies$
if $\left[v \neq \mathbf{0} \text{ we have } v^T A^T Av = (Av)^T(Av) = (z^T)(z) = \sum_{i=0}^{n}({z_i}^2) > 0\right]$
since at least one element of $\left[z \neq 0\right]$. This is the definition of a PD (positive definite matrix)
$\implies \left[A^T A = \nabla^2 f\right] \implies \nabla^2 f$ is PD.

- Since all PD matrices are also PSD – $\left[\nabla^2 > 0\right] \implies \left[\nabla^2 f \geq 0\right]$ – we know that the Hessian of f is PSD, and the second-order necessary condition is satisfied.

(e) Use the first and second-order sufficient condition to check if the candidate is indeed a local minimum. Explain.

- Refer to the work above from part D. We've already shown that the first-order condition is satisfied for the candidate minimum $\left[x = (A^T A)^{-1}(A^T b)\right]$, e.g. $\nabla f\Big((A^T A)^{-1}(A^T b)\Big) = 0$.
- Since we've shown that $\nabla^2 f$ is PD (PSD = necessary, PD = sufficient) in part D, we know that the candidate minimum $\left[x = (A^T A)^{-1}(A^T b)\right]$ is indeed the minimum of the function f since we've satisfied the second-order sufficient condition that the Hessian be PD.

5. In MATLAB create a random A and b for the problem above.

This creates a 100×3 matrix A and a 100×1 vector b populated with values coming from a uniform distribution, $\sim U[0.0, 1.0]$

```
>> A = rand(100,3);
>> b = rand(100,1);
```

Since I do parts (e), (f), and (g) in Python, I create objects of the same size also populated with values coming from a uniform distribution, $\sim U[0.0, 1.0]$ as follows:

```
>> A = np.random.uniform(low=0.0, high=1.0, size=(100,3))
>> b = np.random.uniform(low=0.0, high=1.0, size=(100,1))
```

(a) Solve the problem using your analytical solution from Problem 4.

- For code see **CODE APPENDIX** at end of homework printout (Python code is in black, after the MATLAB code).
- I have a large comment around PROBLEM 5 so that it's easy to find. Here is part (a) output:

```
--------------------------------------------------
--- 5A: Analytical Solution ----------------------
--------------------------------------------------
x* analytical   :  [0.39832205 0.30460217 0.18999574]
f(x*)           :  4.365045447332738
```

Figure 4: Analytical solution

(b) Solve the problem using MATLAB's `pinv` command.

- For code see **CODE APPENDIX** at end of homework printout (Python code is in black, after the MATLAB code).
- **NOTE!!!:** since I used Python for parts A, E, F, and G, the answers are different than for parts B, C, and D where I used MATLAB. This is due to different random numbers being generated. BUT, there is consistency within the Python answers and there is consistency within the MATLAB answers.
- I have a large comment around PROBLEM 5 so that it's easy to find. Here is part (b) output:

```
pinv_soln =

    0.3673
    0.3322
    0.2414


Minimum found that satisfies the constraints.
```

Figure 5: Solution using MATLAB's `pinv` command.

(c) Solve the problem using MATLAB's `quadprog` command.

- For code see **CODE APPENDIX** at end of homework printout (Python code is in black, after the MATLAB code).
- **NOTE!!!:** since I used Python for parts A, E, F, and G, the answers are different than for parts B, C, and D where I used MATLAB. This is due to different random numbers being generated. BUT, there is consistency within the Python answers and there is consistency within the MATLAB answers.
- I have a large comment around PROBLEM 5 so that it's easy to find. Here is part (c) output:

```
qprog_soln =

    0.3673
    0.3322
    0.2414


Local minimum found.
```

Figure 6: Solution using MATLAB's `quadprog` command.

(d) Solve the problem using MATLAB's `fminunc` command.

- For code see **CODE APPENDIX** at end of homework printout (Python code is in black, after the MATLAB code).
- **NOTE!!!:** since I used Python for parts A, E, F, and G, the answers are different than for parts B, C, and D where I used MATLAB. This is due to different random numbers being generated. BUT, there is consistency within the Python answers and there is consistency within the MATLAB answers.
- I have a large comment around PROBLEM 5 so that it's easy to find. Here is part (d) output:

```
fminunc_soln =

    0.3673    0.3322    0.2414
```

Figure 7: Solution using MATLAB's `fminunc` command.

(e) Solve the problem using your gradient descent program from Problem 2.

- For code see **CODE APPENDIX** at end of homework printout (Python code is in black, after the MATLAB code).
- I have a large comment around PROBLEM 5 so that it's easy to find. Here is part (e) output:

```
----------------------------------------------------------------
--- 5E: Gradient Descent for random A, b, x0=[0.4, 0.4, 0.4] ---
----------------------------------------------------------------
x* analytical     :  [0.39832205 0.30460217 0.18999574]
x* grad desc      :  [0.39832206 0.3046021  0.1899958 ]
f(x*)             :  4.365045447332767
iters to converge :  64
```

Figure 8: Solution using gradient descent. Shows analytical solution x, gradient descent x, optimal f, and the number of iterations it took to converge to within η of the true optimum.

(f) Solve the problem using your Newton's method program from Problem 3.

- For code see **CODE APPENDIX** at end of homework printout (Python code is in black, after the MATLAB code).
- I have a large comment around PROBLEM 5 so that it's easy to find. Here is part (f) output:

```
----------------------------------------------------------------
---- 5F: Newton's Method for random A, b, x0=[0.4, 0.4, 0.4] ---
----------------------------------------------------------------
x* analytical     :  [0.39832205 0.30460217 0.18999574]
x* Newton's mtd   :  [0.39832205 0.30460217 0.18999574]
f(x*)             :  4.365045447332738
iters to converge :  2
```

Figure 9: Solution using Newton's Method. Shows analytical solution x, Newton's method x, optimal f, and the number of iterations it took to converge to within η of the true optimum.

(g) Try different initial guesses and document your observations/thoughts.

- For code see **CODE APPENDIX** at end of homework printout (Python code is in black, after the MATLAB code).
- To summarize what the output below shows: Newton's Method converges far faster than gradient descent does (in terms of number of iterations needed). Even when given an initial x_0 far from the true value x^* that optimizes f, Newton's Method typically only took 2 iterations to converge.
- Also, Newton's Method came up with more precise approximations of x^* than gradient descent did. My guess is that this has to do with the fact that Newton's Method incorporates the additional information provided by the Hessian very well, whereas gradient descent doesn't have this same information (the Hessian).
- I have a large comment around PROBLEM 5 so that it's easy to find. Here is part (g) output:

```
-------------------------------------------------------------------
--- 5G: Gradient Descent (x0_close=anltc_sln-[0.01, 0.01, 0.01]) ---
-------------------------------------------------------------------
x* analytical     :  [0.39832205 0.30460217 0.18999574]
x* grad desc      :  [0.39832205 0.30460225 0.18999568]
f(x*)             :  4.365045447332774
iters to converge :  38
```

Figure 10: Solution using gradient descent with an initial guess, x_0 close to the true optimum. Shows analytical solution x, gradient descent x, optimal f, and the number of iterations it took to converge to within η of the true optimum.

```
-------------------------------------------------------
--- 5G: Gradient Descent (x0_far=[300.0, -400.0, 120.0]) ---
-------------------------------------------------------
x* analytical     :  [0.39832205 0.30460217 0.18999574]
x* grad desc      :  [0.39832207 0.30460209 0.18999582]
f(x*)             :  4.3650454473327756
iters to converge :  103
```

Figure 11: Solution using gradient descent with an initial guess, x_0 far from the true optimum. Shows analytical solution x, gradient descent x, optimal f, and the number of iterations it took to converge to within η of the true optimum. Notice that this took many more iterations than the "close" x_0 version did.

```
--------------------------------------------------------------------
--- 5G: Newton's Method (x0_close=anltc_sln-[0.01, 0.01, 0.01]) ----
--------------------------------------------------------------------
x* analytical     :  [0.39832205 0.30460217 0.18999574]
x* Newton's mtd   :  [0.39832205 0.30460217 0.18999574]
f(x*)             :  4.365045447332738
iters to converge :  2
```

Figure 12: Solution using Newton's Method with an initial guess, x_0 close to the true optimum. Shows analytical solution x, Newton's method x, optimal f, and the number of iterations it took to converge to within ϵ of the true optimum. Only took 2 iterations! Far fewer than gradient descent. Also notice how precise the estimate of x^* is.

```
--------------------------------------------------------------
---- 5G: Newton's Method (x0_far=[300.0, -400.0, 120.0]) ----
--------------------------------------------------------------
x* analytical     :  [0.39832205 0.30460217 0.18999574]
x* Newton's mtd   :  [0.39832205 0.30460217 0.18999574]
f(x*)             :  4.365045447332738
iters to converge :  2
```

Figure 13: Solution using Newton's Method with an initial guess, x_0 far from the true optimum. Shows analytical solution x, Newton's method x, optimal f, and the number of iterations it took to converge to within ϵ of the true optimum. Only took 2 iterations again! Also notice how precise the estimate of x^* is again.

6. Consider the following function: $f(x, y) = (\alpha - x)^2 + \beta(y - x^2)^2$ with $\alpha = 1$ and $\beta = 100$.

(a) Calculate the gradient vector.

- $f'(x, y) = \nabla_f = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \\ \dfrac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} -2\alpha + 2x - 4\beta xy + 4\beta x^3 \\ \\ 2\beta y - 2\beta x^2 \end{bmatrix} = \begin{bmatrix} -2 + 2x - 400xy + 400x^3 \\ \\ 200y - 200x^2 \end{bmatrix}$

(b) Calculate the Hessian matrix.

- The Hessian $\nabla^2{}_f = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x^2} & \dfrac{\partial^2 f}{\partial x \partial y} \\ \\ \dfrac{\partial^2 f}{\partial y \partial x} & \dfrac{\partial^2 f}{\partial y^2} \end{bmatrix} = \begin{bmatrix} 2 - 400y + 1200x^2 & -400x \\ \\ -400x & 200 \end{bmatrix}$

(c) Use the first-order necessary condition to find all candidates for a local minimum.

- $f'(x, y) = \nabla_f = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \\ \dfrac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} -2\alpha + 2x - 4\beta xy + 4\beta x^3 \\ \\ 2\beta y - 2\beta x^2 \end{bmatrix}$ set $= \begin{bmatrix} 0 \\ \\ 0 \end{bmatrix}$ giving
  $\begin{bmatrix} -2\alpha + 2x - 4\beta xy + 4\beta x^3 \\ \\ 2\beta y - 2\beta x^2 \end{bmatrix} = \begin{bmatrix} 0 \\ \\ 0 \end{bmatrix}$
- Now multiply both sides by $\dfrac{1}{2}$, giving:

  $\begin{bmatrix} -\alpha + x - 2\beta xy + 2\beta x^3 \\ \\ \beta y - \beta x^2 \end{bmatrix} = \begin{bmatrix} 0 \\ \\ 0 \end{bmatrix}$ Multiply both sides of bottom equation by $(2x)$ giving:
  $\begin{bmatrix} -\alpha + x - 2\beta xy + 2\beta x^3 \\ \\ 2\beta xy - 2\beta x^3 \end{bmatrix} = \begin{bmatrix} 0 \\ \\ 0 \end{bmatrix}$
- Now add the two equations together (method of elimination) to get:
  $\Big[\big(-\alpha + x - 2\beta xy + 2\beta x^3\big) + \big(2\beta xy - 2\beta x^3\big) = (0) + (0)\Big] =$
  $\Big[-\alpha + x = 0\Big] \implies \Big[x = \alpha\Big]$ **But what is y?**
- We know from the bottom equation that $\beta y - \beta x^2 = 0$. Substituting our solved-for $x = \alpha$ we have $\Big[\beta y - \beta(\alpha)^2 = 0\Big] \to \Big[\dfrac{\beta y}{\beta} = \dfrac{\beta \alpha^2}{\beta}\Big] \to \Big[y = \alpha^2\Big]$
- Therefore, the critical point, and assumed minimum w/out checking the 2^{nd}-order condition, is the vector $\begin{bmatrix} x \\ \\ y \end{bmatrix} = \begin{bmatrix} \alpha \\ \\ \alpha^2 \end{bmatrix} = \begin{bmatrix} 1 \\ \\ 1 \end{bmatrix}$
- This gives a general candidate for minima, but the prompt specifies that $\alpha = 1$ so the specific vector (point) $[x, y]^T = [1, 1]^T$ is our only candidate for the location of a minimum of f.

(d) Compute the eigenvalues of the Hessian matrix evaluated at the candidate points using MATLAB's `eig` command. Is the matrix PSD or PD or something else?

- See **CODE APPENDIX** for my implementation and output. I've also included the code here.

```
%== Define the [x,y] vector = [x(1), x(2)] = [x(1), (x(1))^2]
x(1) = 1.0;
x(2) = (x(1))^2;
%== Define the Hessian of the Rosenbrock function
ros_hess(1,:) = [2-400*x(2)+1200*(x(1)^2)  -400*x(1)];
ros_hess(2,:) = [-400*x(1)  200];
%== Output eigenvalues of the Hessian of the Rosenbrock function
eig(ros_hess)
```

Figure 14: This shows my specification of the Hessian of the Rosenbrock function, and numerical calcuation of the Hessian for different values of $[x, y]^T$, here shown as $[x(1), x(2)]^T$.

- For the candidate specified by the prompt setting $\alpha = 1$, the vector $[1, 1]^T$, we get that the eigenvalues are 0.0004 and 1.0016.
- Since $\Big[\{\text{evals for } [1,1]^T\} = \{0.0004, 1.0016\} > 0\Big] \implies \Big[\nabla^2 f([1,1]^T) \text{ is PD}\Big]$.

(e) Use the first and second-order sufficient condition to check if the candidate is indeed a local minimum. Explain.

- <u>First-order sufficient condition is</u>: any x^* that minimizes f will satisfy $\nabla f(x^*) = 0$
- In part C we have shown that, by construction, the candidate $[1, 1]^T$ for being a minimum location has $\nabla f([1, 1]^T) = [0, 0]^T$. However, we can explicitly show it here.
- Let $x^c = [1, 1]^T$ (standing for x candidate)

$$\nabla f = \begin{bmatrix} -2 + 2x - 400xy + 400x^3 \\ \\ 200y - 200x^2 \end{bmatrix} \longrightarrow \nabla f(x^c) = \begin{bmatrix} -2 + 2(1) - 400(1)(1) + 4001^3 \\ \\ 2001 - 2001^2 \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} -2 + 2 - 400 + 400 \\ \\ 200 - 200 \end{bmatrix} \longrightarrow \nabla f(x^c) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

- This shows that at candidate $\Big[x = [1, 1]^T,\ \nabla f = 0\Big]$, satisfying the first-order sufficient condition.
- <u>Second-order sufficient condition is</u>: any x^* that minimizes f will satisfy $\nabla^2 f(x^*) > 0$, meaning that the Hessian will be PD.
- Refer to the last two bullet points from part D. To reiterate here, since $\Big[\{\text{evals for } [1,1]^T\} = \{0.0004, 1.0016\} > 0\Big] \implies \Big[\nabla^2 f([1,1]^T) \text{ is PD}\Big]$ $\implies \Big[\nabla^2(f([1,1]^T) \text{ is indeed a minimum}\Big]$.
- In summary, since $\Big[\nabla f([1,1]^T) = [0, 0]\Big]$ and $\Big[\nabla^2 f([1,1]^T) \text{ is PD}\Big]$ we know that $x = [1, 1]^T$ minimizes f.

7. For the optimization problem described in Problem 6:

   (a) Solve the problem using MATLAB's `fminunc` command.
   - For code see **CODE APPENDIX** at end of homework printout (MATLAB code is at the top, white background).

```
fminunc_rosen_soln =

    1.0000    1.0000
```

Figure 15: Solution output to minimizing the Rosenbrock ($\alpha = 1$ and $\beta = 100$) function with `fminunc`.

   (b) Solve the problem using your gradient descent program from Problem 2.
   - For code see **CODE APPENDIX** at end of homework printout (Python code is toward the bottom, black background).

```
----------------------------------------------------------------
---- Gradient Descent to minimize Rosenbrock (x0=[1.3,1.3]) ----
----------------------------------------------------------------
x* true           :  [1. 1.]
x* grad desc      :  [1.00000111 1.00000222]
f(x*)             :  1.234326908705323e-12
iters to converge :  1628
```

Figure 16: Solution output to minimizing the Rosenbrock ($\alpha = 1$ and $\beta = 100$) function with gradient descent and starting point of $x_0 = [1.3, 1.3]^T$ that I arbitrarily picked (problem didn't specify x_0 for this part.)

   (c) Solve the problem using your Newton's Method program from Problem 3.
   - For code see **CODE APPENDIX** at end of homework printout (Python code is toward the bottom, black background).

```
----------------------------------------------------------------
---- Newton's Method to minimize Rosenbrock (x0=[1.3,1.3]) ----
----------------------------------------------------------------
x* true           :  [1. 1.]
x* Newton's mtd   :  [1.00048475 1.00092863]
f(x*)             :  4.039592512867547e-07
iters to converge :  8
```

Figure 17: Solution output to minimizing the Rosenbrock ($\alpha = 1$ and $\beta = 100$) function with Newton's Method and starting point of $x_0 = [1.3, 1.3]^T$ that I arbitrarily picked (problem didn't specify x_0 for this part.)

(d) Try initial guesses $[1, -1]^T$ and $[100, -1]^T$. Document your observations.

- For code see **CODE APPENDIX** at end of homework printout (Python code is toward the bottom, black background).

```
------------------------------------------------
---- Results of Gradient Descent (x0=[1,-1]) ----
------------------------------------------------
x* true            :  [1. 1.]
x* grad desc       :  [0.99999905 0.99999809]
f(x*)              :  9.10906532172002e-13
iters to converge :  2599
```

Figure 18: Solution output to minimizing the Rosenbrock ($\alpha = 1$ and $\beta = 100$) function with gradient descent and starting point of $x_0 = [1.0, -1.0]^T$.

```
------------------------------------------------
---- Results of Newton's Method (x0=[1,-1]) ----
------------------------------------------------
x* analytical      :  [1. 1.]
x* Newton's mtd    :  [1. 1.]
f(x*)              :  1.232595164407831e-28
iters to converge :  2
```

Figure 19: Solution output to minimizing the Rosenbrock ($\alpha = 1$ and $\beta = 100$) function with Newton's Method and starting point of $x_0 = [1.0, -1.0]^T$.

```
------------------------------------------------
---- Results of Gradient Descent (x0=[100,-1]) ----
------------------------------------------------
x* true            :  [1. 1.]
x* grad desc       :  [0.99999891 0.99999781]
f(x*)              :  1.194328307854695e-12
iters to converge :  3125
```

Figure 20: Solution output to minimizing the Rosenbrock ($\alpha = 1$ and $\beta = 100$) function with gradient descent and starting point of $x_0 = [100.0, -1.0]^T$.

```
------------------------------------------------
---- Results of Newton's Method (x0=[100,-1]) ----
------------------------------------------------
x* analytical      :  [1. 1.]
x* Newton's mtd    :  [1.00001704 1.00003189]
f(x*)              :  7.687080724480512e-10
iters to converge :  210
```

Figure 21: Solution output to minimizing the Rosenbrock ($\alpha = 1$ and $\beta = 100$) function with Newton's Method and starting point of $x_0 = [100.0, -1.0]^T$.

- Newton's Method converges in fewer iterations for both x_0:
  $\left[2_{NM} < 2599_{GD}\right]$ for $x_0 = [1, -1]^T$ and $\left[210_{NM} < 3125_{GD}\right]$ for $x_0 = [100, -1]^T$.
- Also, for $x_0 = [1, -1]^T$ Newton's Method achieves (essentially) the exact answer, while gradient descent is just a very close approximation.
- As I've said above in 5G, my guess is that the main reason Newton's Method outperforms gradient descent is because Newton's Method incorporates the additional information provided by the Hessian, whereas gradient descent only makes use of the gradient information.

8. Consider the linear programming (LP) problem with objective:

minimize:
$f = x_1 - x_2$

subject to:

$$\begin{aligned} x_1 + x_2 &\leq 1 \\ -x_1 + 2x_2 &\leq 2 \\ x_1 &\geq -1 \\ -x_1 + 3x_2 &\geq -3 \end{aligned}$$

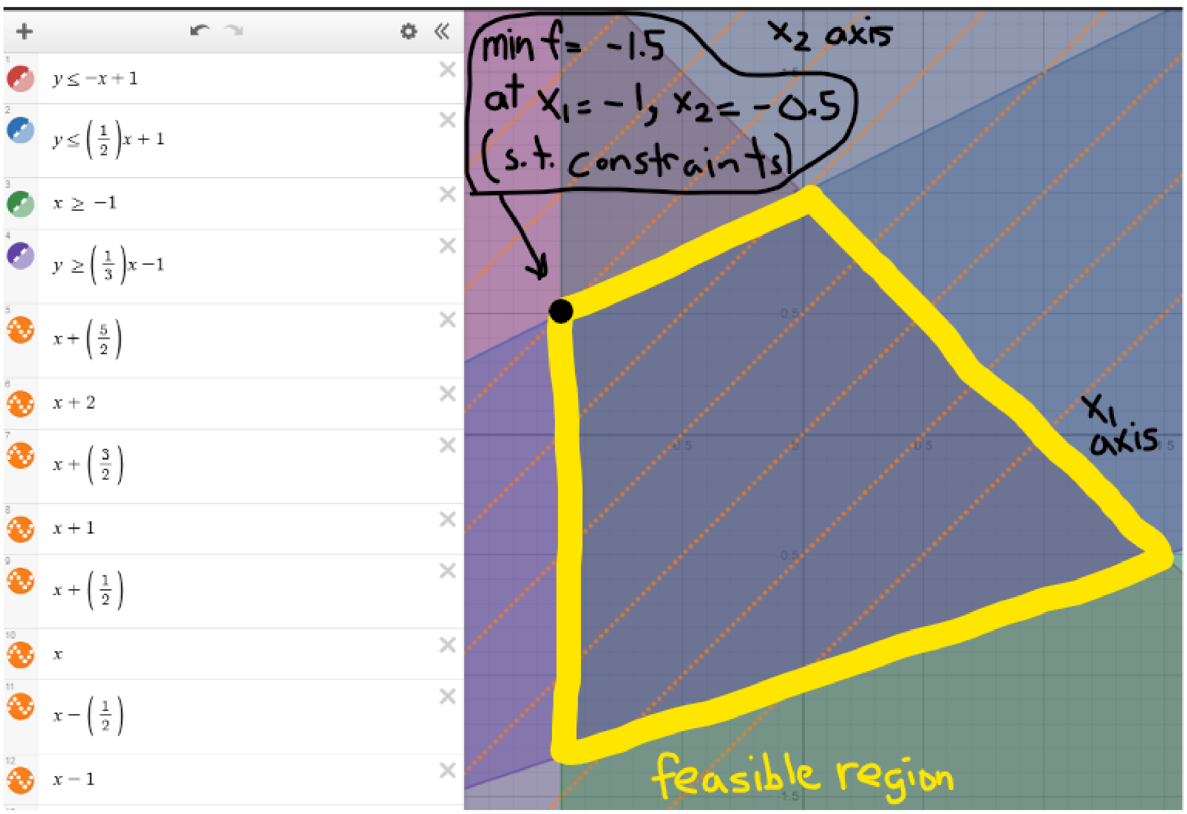

Figure 22: **Refer to this figure for parts A, B, and C.**

(a) Plot the level curves (lines) of the objective function on an $x_1 - x_2$ graph. Identify the direction of decreasing objective value.

- Refer to Figure 22 for the plot of level curves of the objective function. They are drawn as orange dotted lines.
- As we can see, the value of the objective function ($f = x_1 - x_2$) decreases as we move to level sets closer to the $(-\infty, \infty)$ "far reaches" of Quadrant II in the (x_1, x_2) Cartesian plane.

(b) Identify the feasible region by plotting the constraints on the same graph.

- Refer to Figure 22 for the plot of the constraints and feasible region.
- The constraints are shaded in red, blue, green, and purple. The area where they overlap is denoted by the closed-in yellow boundary. The feasible region is the area enclosed in yellow, and includes the boundary edges since all constraints are either $\leq$ or $\geq$.

(c) Graphically identify the minimum point using your graph.

- The way we graphically identify the minimum point is to find the level set with the lowest objective function value f that intersects with the feasible region. Also, we know that this will occur on an edge of the feasible region (not in the middle).
- Examining Figure 22 we can see that the level set of $x_1 - x_2 = -\frac{3}{2}$ $\left(\text{graphed as } y = x + \frac{3}{2}\right)$ intersects the feasible region at the point $(x_1 = -1.0, x_2 = 0.5)$, yielding an objective function value of $f = x_1 - x_2 = -1.0 - 0.5 = -1.5$ at that point.
- Therefore, the minimum of $f(x^*) = -1.5$ where $x^* = \begin{bmatrix} {x_1}^* \\ \\ {x_2}^* \end{bmatrix} = \begin{bmatrix} -1.0 \\ \\ 0.5 \end{bmatrix}$

(d) Convert the problem to standard (general) LP form and solve using MATLAB's `linprog`.

- First, let's recall what standard (general) form for an LP looks like:
  i. $\left[\min_x c^T x\right]$, subject to ii, iii, and iv:
  ii. $Ax \leq b$
  iii. $\bar{A}x = \bar{b}$
  iv. $l \leq x \leq u$
- Let's rewrite $\left[-x_1 + 3x_2 \geq -3\right] \longrightarrow \left[(-1)(-x_1 + 3x_2) \leq (-1)(-3)\right] \longrightarrow \left[x_1 - 3x_2 \leq 3\right]$
- We can now write this LP in standard form:
  i. $\left[\min_x c^T x\right] \longrightarrow \min_x \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ \\ x_2 \end{bmatrix}$ subject to:
  ii. $\left[Ax \leq b\right] \longrightarrow \begin{bmatrix} 1 & 1 \\ -1 & 2 \\ 1 & -3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$
  iii. No equality constraints for this LP
  iv. $\left[l \leq x \leq u\right] \longrightarrow \begin{bmatrix} -1 \\ -\infty \end{bmatrix} \leq \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} \infty \\ \infty \end{bmatrix}$

- Now we can solve the problem in MATLAB. For code see **CODE APPENDIX** at end of homework printout (MATLAB code is at the top, white background).
  I've also included the code here for your convenience.

```
%=========================================================================
%== 8(D): Convert the problem to standard form and solve using linprog
%=========================================================================

% Coefficients for the objective function f = x(1) - x(2)
f = [1, -1];
% The matrix for inequality constraints
A(1,:) = [1, 1];
A(2,:) = [-1, 2];
A(3,:) = [1, -3];
% The column vector for inequality constraints
b = [1; 2; 3];
% The vector bounding x on the low side
lb = [-1; -inf];
% The vector bounding x on the high side
ub = [inf; inf];

% Solve the problem using linprog
linprog_soln = linprog(f, A, b, [], [], lb, ub);
linprog_soln
```

Figure 23: MATLAB code for solving the general form LP formulated above.

- Here is the MATLAB output of the solution. As we can see, this matches our graphical solution, giving that the x that minimizes $f = x_1 - x_2$ is $x^* = \begin{bmatrix} {x_1}^* \\ {x_2}^* \end{bmatrix} = \begin{bmatrix} -1.0 \\ 0.5 \end{bmatrix}$

```
linprog_soln =

   -1.0000
    0.5000
```

Figure 24: MATLAB output for the solution $[x_1, x_2]^T$.

CODE APPENDIX, problems in order (Matlab=white, Python=black)

```
%===========================================================================
%== ASSIGNMENT : hw2, Optimization (MAE 5930)
%== AUTHOR     : Jared Hansen
%== DUE        : Thursday, 09/26/2019
%===========================================================================
rng(1776)
clear all; close all; clc;








%===========================================================================
%===========================================================================
%== PROBLEM 5
%===========================================================================
%===========================================================================

%===========================================================================
%== Create the matrix A and the vector b, filled with ~U[0,1] values
%===========================================================================
A = rand(100,3);
b = rand(100,1);

%===========================================================================
%== 5(B): solve using pinv command
%===========================================================================
%== The analytical solution is: [(A.T * A)^(-1)] * [(A.T) * b]
pinv_soln = pinv((A.')*(A)) * ((A.')*(b));
pinv_soln

%===========================================================================
%== 5(C): solve using quadprog command
%===========================================================================
%== Define matrix H and row vector f
H = ((A.')*(A));
f = (-1.0*(b.')*(A));
qprog_soln = quadprog(H, f);
qprog_soln

%===========================================================================
%== 5(D): solve using fminunc command
%===========================================================================
%== Define the function and an initial guess
fun = @(x) (1/2.0)*(x)*(A.')*(A)*(x.') - (b.')*(A)*(x.');
x0 = [1.0, 1.0, 1.0];
fminunc_soln = fminunc(fun, x0);
fminunc_soln
```

```
clear all; close all; clc;
%=========================================================================
%=========================================================================
%== PROBLEM 6
%=========================================================================
%=========================================================================

%== Define the Rosenbrock function
%rosen = @(x) (1-x(1))^2 + 100*(x(2) - x(1)^2)^2;
%== Define the [x,y] vector = [x(1), x(2)] = [x(1), (x(1))^2]
x(1) = 3.0;
x(2) = (x(1))^2;
%== Define the Hessian of the Rosenbrock function
ros_hess(1,:) = [2-400*x(2)+1200*(x(1)^2)  -400*x(1)];
ros_hess(2,:) = [-400*x(1)  200];
%== Output eigenvalues of the Hessian of the Rosenbrock function
eig(ros_hess)
```

```
clear all; close all; clc;
%=========================================================================
%=========================================================================
%== PROBLEM 7
%=========================================================================
%== For the optimization problem described in Problem 4:
%=========================================================================

%=========================================================================
%== 7(A): solve using fminunc command
%=========================================================================

%== Define the Rosenbrock function
rosen = @(x) (1-x(1))^2 + 100*(x(2) - x(1)^2)^2;
x0 = [2.0, 2.0];
fminunc_rosen_soln = fminunc(rosen, x0);
fminunc_rosen_soln
```

```
clear all; close all; clc;
%=======================================================================
%=======================================================================
%== PROBLEM 8
%=======================================================================
%=======================================================================

%=======================================================================
%== 8(D): Convert the problem to standard form and solve using linprog
%=======================================================================

% Coefficients for the objective function f = x(1) - x(2)
f = [1, -1];
% The matrix for inequality constraints
A(1,:) = [1, 1];
A(2,:) = [-1, 2];
A(3,:) = [1, -3];
% The column vector for inequality constraints
b = [1; 2; 3];
% The vector bounding x on the low side
lb = [-1; -inf];
% The vector bounding x on the high side
ub = [inf; inf];

% Solve the problem using linprog
linprog_soln = linprog(f, A, b, [], [], lb, ub);
linprog_soln
```

```
%=========================================================================
%=========================================================================
%== MATLAB AUTOMATIC OUTPUT BELOW THIS POINT
%=========================================================================
%=========================================================================


pinv_soln =

    0.3673
    0.3322
    0.2414


Minimum found that satisfies the constraints.

Optimization completed because the objective function is non-
decreasing in
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint
 tolerance.


qprog_soln =

    0.3673
    0.3322
    0.2414


Local minimum found.

Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.


fminunc_soln =

    0.3673    0.3322    0.2414


ans =

   1.0e+03 *

    0.0001
    7.4019


Local minimum found.

Optimization completed because the size of the gradient is less than
```

```
the value of the optimality tolerance.


fminunc_rosen_soln =

    1.0000    1.0000


Optimal solution found.


linprog_soln =

   -1.0000
    0.5000
```

Published with MATLAB® R2019a

```
'''
Assignment:     hw2, MAE 5930 (Optimization)
File name:      optzn_hw2__JHansen.py
Author:         Jared Hansen
Date created:   09/23/2019
Python version: 3.7.3

DESCRIPTION:
    This Python script is used for answering the following questions from
    MAE 5930 (Optimization) hw2:
        = Problem 1
        = Problem 2
        = Problem 3
        = Problem 5: parts E and F
        = Problem 7: parts B, C, and D
'''










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












#-----------------------------------------------------------------------------
#-----------------------------------------------------------------------------
#--- PROBLEM 1 ---------------------------------------------------------------
#-----------------------------------------------------------------------------
# Program the backtracking algorithm as described by Boyd on page 464.
#-----------------------------------------------------------------------------
#-----------------------------------------------------------------------------

#--- PART A ------------------------------------------------------------------
# Inputs to the function are the objective function, gradient function
# evaluated at x, the point x, and the descent direction v.
#--- PART B ------------------------------------------------------------------
# Within the function set alpha = 0.2 and beta = 0.5.
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


# Testing out the backtracking algorithm function using the Rosenbrock function
'''
x_val = np.array([1.0,2.0])
backtrack(rosen_f, rosen_grad, x_val, np.array([-1.0,1.0]))
'''

















#-----------------------------------------------------------------------------
#-----------------------------------------------------------------------------
#--- PROBLEM 2 ---------------------------------------------------------------
#-----------------------------------------------------------------------------
# Program the gradient descent method as described by Boyd on page 466.
#-----------------------------------------------------------------------------
#-----------------------------------------------------------------------------
#--- PART A ------------------------------------------------------------------
# Inputs to the function are the objective, gradient, and starting point.
#--- PART B ------------------------------------------------------------------
# Within the function set eta = 1e-6

def grad_desc(f, grd, x0):
    """
    This function implements the gradient descent algorithm.

    Parameters
    ----------
    f   : a mathematical function (the objective function)
    grd : a mathematical function (the gradient of f)
    x0  : a starting pt in the domain of f (either a scalar or an array/list)

    Returns
    -------
    grad_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 = 10000
    # Define our tolerance, the constant ETA
    ETA = 1e-6
    # Initialize the point we're going to update, x, as the starting point x0
    x = x0
    while(np.linalg.norm(grd(x)) > ETA):
        # Find step size using backtracking
        t = backtrack(f, grd, x, (-1*grd(x)).T.flatten())
        # Update our "more minimizing" point x
        x = (x - (t * grd(x)).reshape(len(x),))
        #x = x - (t*grd(x))
        # Print statements to see what is going on
        print("f(x)  : ", f(x))
        print("iters : ", iters)
        print()
        # Update the 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 ETA or reaching MAX_ITERS return a tuple
    # containing the final_x, f_final, and iters
    grad_output = (x, f(x), iters)
    return(grad_output)









#------------------------------------------------------------------------------
#------------------------------------------------------------------------------
#--- PROBLEM 3 ----------------------------------------------------------------
#------------------------------------------------------------------------------
# Program Newton's Method as described by Boyd on page 487.
#------------------------------------------------------------------------------
#------------------------------------------------------------------------------
#--- PART A -------------------------------------------------------------------
# Inputs to the function are the objective, gradient, Hessian, and starting pt.
#--- PART B -------------------------------------------------------------------
# Within the function set epsilon = 1e-6

def newtons_method(f, grd, hessian, x0):
    """
    This function implements Newton's method.

    Parameters
```

```
    ----------
    f   : a mathematical function (the objective function)
    grd : a mathematical function (the gradient of f)
    x0  : a starting pt in the domain of f (either a 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(x))) ,
                         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(hessian(x)) , grd(x))
        # Line search for t (descent direction)
        t = backtrack(f, grd, x, (np.array(dlt_x).flatten()))
        # Update our "more minimizing" x
        x = np.array(x + np.dot(t, dlt_x).reshape(len(x), )).flatten()
        # 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(x))) ,
                         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)

#--------------------------------------------------------------------------
```

```
#-----------------------------------------------------------------------------
#--- PROBLEM 5 ----------------------------------------------------------------
#-----------------------------------------------------------------------------
#-----------------------------------------------------------------------------


# Create a 100x3 matrix A and a 100x1 vector b, each containing random values
# uniformly distributed on [0,1].
#-----------------------------------------------------------------------------
A = np.random.uniform(low=0.0, high=1.0, size=(100,3))
b = np.random.uniform(low=0.0, high=1.0, size=(100,1))



#--- PART A -------------------------------------------------------------------
# Solve the problem using your analytical solution from Problem 4.
# Our solution from problem 4 is x = (A.T * A)^(-1) * (A.T) * (b)
analytical_soln = np.dot( (np.dot( np.linalg.pinv(np.dot(A.T, A)), A.T) ), b)
analytical_soln

regr_obj(analytical_soln)



#--- FOR PARTS e, f, AND g ----------------------------------------------------
#-----------------------------------------------------------------------------
#--- DEFINE FUNCTIONS FOR: OBJECTIVE FCTN, GRADIENT, and HESSIAN -------------
def regr_obj(x):
    """ Takes the point x (R^3 vec) and returns the value of the regression
    (objective) function
    """
    return(np.asscalar((1/2.0) * np.dot((np.dot(A, x).reshape(100,1) - b).T,
                                        (np.dot(A, x).reshape(100,1) - b))))

def regr_grad(x):
    """ Takes the point x (R^3 vec) and returns the gradient of the regression
    (objective) function
    """
    return(np.dot(np.dot(A.T, A), x).reshape(3,1) -np.dot(A.T, b).reshape(3,1))

def regr_hess(x):
    """ Takes the matrix A (dim(A) = 100x3) and returns the Hessian of the
    regression (objective) function
    """
    return(np.dot(A.T, A))


#--- PART E -------------------------------------------------------------------
# Solve the problem using your gradient descent program from Problem 2.
#-----------------------------------------------------------------------------
# Let's make an initial guess. I'm going to start with all values of x = 0.4
x0 = np.array([0.4, 0.4, 0.4])
# Find the solution using gradient descent
grad_output = grad_desc(regr_obj, regr_grad, x0)
# Output the results to the console
print("\n----------------------------------------------------------------")
print("--- 5E: Gradient Descent for random A, b, x0=[0.4, 0.4, 0.4] ---")
print("----------------------------------------------------------------")
print("x* analytical     : ", analytical_soln.flatten())
print("x* grad desc      : ", grad_output[0])
print("f(x*)             : ", grad_output[1])
print("iters to converge : ", grad_output[2])


#--- PART F -------------------------------------------------------------------
# Solve the problem using your Newton's method program from Problem 3.
#-----------------------------------------------------------------------------
```

```python
# Let's make an initial guess. I'm going to start with all values of x = 0.4
x0 = np.array([0.4, 0.4, 0.4])
# Find the solution using gradient descent
newt_output = newtons_method(regr_obj, regr_grad, regr_hess, x0)
# Output the results to the console
print("\n------------------------------------------------------------")
print("---- 5F: Newton's Method for random A, b, x0=[0.4, 0.4, 0.4] ---")
print("------------------------------------------------------------")
print("x* analytical     : ", analytical_soln.flatten())
print("x* Newton's mtd   : ", newt_output[0])
print("f(x*)             : ", newt_output[1])
print("iters to converge : ", newt_output[2])


#--- PART G --------------------------------------------------------------
# Try different initial guesses and document your observations/thoughts.
#-------------------------------------------------------------------------

# Let's try a starting guess that is far from the correct answer (x0_far)
x0_far = np.array([300.0, -400.0, 120.0])
# With gradient descent
grad_output_FAR = grad_desc(regr_obj, regr_grad, x0_far)
print("\n---------------------------------------------------------")
print("--- 5G: Gradient Descent (x0_far=[300.0, -400.0, 120.0]) ---")
print("---------------------------------------------------------")
print("x* analytical     : ", analytical_soln.flatten())
print("x* grad desc      : ", grad_output_FAR[0])
print("f(x*)             : ", grad_output_FAR[1])
print("iters to converge : ", grad_output_FAR[2])
# With Newton's method
newt_output_FAR = newtons_method(regr_obj, regr_grad, regr_hess, x0_far)
print("\n---------------------------------------------------------")
print("---- 5G: Newton's Method (x0_far=[300.0, -400.0, 120.0]) ----")
print("----------------------------------------------------------")
print("x* analytical     : ", analytical_soln.flatten())
print("x* Newton's mtd   : ", newt_output_FAR[0])
print("f(x*)             : ", newt_output_FAR[1])
print("iters to converge : ", newt_output_FAR[2])

# Let's try a starting guess that is close to the correct answer (x0_close)
x0_close = (analytical_soln -
        np.array([0.01, 0.01, 0.01]).reshape(3,1)).flatten()
# With gradient descent
grad_output_CLOSE = grad_desc(regr_obj, regr_grad, x0_close)
print("\n----------------------------------------------------------------")
print("--- 5G: Gradient Descent (x0_close=anltc_sln-[0.01, 0.01, 0.01]) ---")
print("----------------------------------------------------------------")
print("x* analytical     : ", analytical_soln.flatten())
print("x* grad desc      : ", grad_output_CLOSE[0])
print("f(x*)             : ", grad_output_CLOSE[1])
print("iters to converge : ", grad_output_CLOSE[2])
# With Newton's method
newt_output_CLOSE = newtons_method(regr_obj, regr_grad, regr_hess, x0_close)
print("\n----------------------------------------------------------------")
print("--- 5G: Newton's Method (x0_close=anltc_sln-[0.01, 0.01, 0.01]) ----")
print("----------------------------------------------------------------")
print("x* analytical     : ", analytical_soln.flatten())
print("x* Newton's mtd   : ", newt_output_CLOSE[0])
print("f(x*)             : ", newt_output_CLOSE[1])
print("iters to converge : ", newt_output_CLOSE[2])
```

```python
#-----------------------------------------------------------------------------
#-----------------------------------------------------------------------------
#--- PROBLEM 7 ---------------------------------------------------------------
#-----------------------------------------------------------------------------
# For the optimization problem described in Problem 6:
#-----------------------------------------------------------------------------


#-----------------------------------------------------------------------------
#--- DEFINING ROSENBROCK FUNCTIONS (OBJECTIVE, GRADIENT, HESSIAN) FOR
#--- MINIMIZING WITH GRADIENT DESCENT AND NEWTON'S METHOD
#-----------------------------------------------------------------------------


#--- ROSENBROCK FUNCTION ---------------------------------------------------
def rosen_f(x):
    """ Takes the point (R^2 vector) x and returns the Rosenbrock function at x
    """
    return((1 - x[0])**2+ 100*((x[1]-x[0]**2)**2))

#--- ROSENBROCK FUNCTION'S GRADIENT -----------------------------------------
def rosen_grad(x):
    """ Takes the point x and returns the gradient of the Rosenbrock fctn at x
    """
    # The partial derivative of f w.r.t. x[0]
    df1 = -2*(1 - x[0]) - (400*x[0])*(x[1] - (x[0]**2))
    # The partial derivative of f w.r.t. x[1]
    df2 = 200*(x[1] - (x[0]**2))
    # Returns the gradient as a NumPy array
    return(np.array([df1, df2]))

#--- ROSENBROCK FUNCTION'S HESSIAN ------------------------------------------
def rosen_hess(x):
    """ Takes the point x and returns the Hessian of the Rosenbrock fctn at x
    """
    x0 = np.asscalar(x[0])
    x1 = np.asscalar(x[1])
    # The second partial derivative of f w.r.t. x[0]
    d2f_dx2 = 2 - 400*(x1) + 1200 * (x0**2)
    # The off-diagonal entry in the Hessian
    d2f_dydx = -400*x0
    # The second partial derivative of f w.r.t. x[1]
    d2f_dy2 = 200
    # Arrange these partial derivatives in a matrix, return that matrix
    hess_matrix = np.matrix([[d2f_dx2, d2f_dydx], [d2f_dydx, d2f_dy2]])
    return(hess_matrix)


#--- PART B ------------------------------------------------------------------
# Solve the problem using your gradient descent program from Problem 2.
#-----------------------------------------------------------------------------
# For x0=[1.3,1.3] grad desc converges to local min [1,1] in 1628 iterations
gd_rosen = grad_desc(rosen_f, rosen_grad, np.array([1.3, 1.3]))
print("\n-------------------------------------------------------------")
print("---- Gradient Descent to minimize Rosenbrock (x0=[1.3,1.3]) ----")
print("-------------------------------------------------------------")
print("x* true            : ", np.array([1.0,1.0]))
```

```python
print("x* grad desc      : ", gd_rosen[0])
print("f(x*)             : ", gd_rosen[1])
print("iters to converge : ", gd_rosen[2])


#--- PART C ---------------------------------------------------------------
# Solve the problem using your Newton's method program from Problem 2.
#--------------------------------------------------------------------------
# For x0=[1.3,1.3] Newton's method converges to local min [1,1] in 8 iterations
nm_rosen = newtons_method(rosen_f, rosen_grad, rosen_hess,
                          np.array([1.3, 1.3]))
print("\n----------------------------------------------------------------")
print("---- Newton's Method to minimize Rosenbrock (x0=[1.3,1.3]) ----")
print("----------------------------------------------------------------")
print("x* true           : ", np.array([1.0,1.0]))
print("x* Newton's mtd   : ", nm_rosen[0])
print("f(x*)             : ", nm_rosen[1])
print("iters to converge : ", nm_rosen[2])


#--- PART D ---------------------------------------------------------------
# Try initial guesses [1.0, -1.0] and [100.0, -1.0]. Document observations
#--------------------------------------------------------------------------
# Gradient descent converges to local min [1,1] in 2599 iterations
gd_1_neg1 = grad_desc(rosen_f, rosen_grad, np.array([1.0, -1.0]))
print("\n-------------------------------------------------")
print("---- Results of Gradient Descent (x0=[1,-1]) ----")
print("-------------------------------------------------")
print("x* true           : ", np.array([1.0,1.0]))
print("x* grad desc      : ", gd_1_neg1[0])
print("f(x*)             : ", gd_1_neg1[1])
print("iters to converge : ", gd_1_neg1[2])
# Newton's method converges to local min [1,1] in 2 iterations
nm_1_neg1 = newtons_method(rosen_f, rosen_grad, rosen_hess,
                           np.array([1.0, -1.0]))
print("\n-------------------------------------------------")
print("---- Results of Newton's Method (x0=[1,-1]) ----")
print("-------------------------------------------------")
print("x* analytical     : ", np.array([1.0,1.0]))
print("x* Newton's mtd   : ", nm_1_neg1[0])
print("f(x*)             : ", nm_1_neg1[1])
print("iters to converge : ", nm_1_neg1[2])
# Gradient descent converges to local min [1,1] in 3125 iterations
gd_100_neg1 = grad_desc(rosen_f, rosen_grad, np.array([100.0, -1.0]))
print("\n---------------------------------------------------")
print("---- Results of Gradient Descent (x0=[100,-1]) ----")
print("---------------------------------------------------")
print("x* true           : ", np.array([1.0,1.0]))
print("x* grad desc      : ", gd_100_neg1[0])
print("f(x*)             : ", gd_100_neg1[1])
print("iters to converge : ", gd_100_neg1[2])
# Newton's method converges to local min [1,1] in 210 iterations
nm_100_neg1 = newtons_method(rosen_f, rosen_grad, rosen_hess,
                             np.array([100.0, -1.0]))
print("\n---------------------------------------------------")
print("---- Results of Newton's Method (x0=[100,-1]) ----")
print("---------------------------------------------------")
print("x* analytical     : ", np.array([1.0,1.0]))
print("x* Newton's mtd   : ", nm_100_neg1[0])
print("f(x*)             : ", nm_100_neg1[1])
print("iters to converge : ", nm_100_neg1[2])
```