```
import numpy as np
import matplotlib as mpl, matplotlib.pyplot, Image
%matplotlib inline
Image.fromarray(np.ones(3,dtype=int)*255)
```

Question 1(a)

Consider the function $f(x)$ defined below:

The gradient of $f(x)$

$$f(x) = x^T A x + b^T x + c$$

$$\nabla f(x) = \frac{\partial f(x)}{\partial x} = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial (x^T A x)}{\partial x_1} + \frac{\partial (b^T x)}{\partial x_1} + \frac{\partial c}{\partial x_1} \\ \frac{\partial (x^T A x)}{\partial x_2} + \frac{\partial (b^T x)}{\partial x_2} + \frac{\partial c}{\partial x_2} \\ \vdots \\ \frac{\partial (x^T A x)}{\partial x_n} + \frac{\partial (b^T x)}{\partial x_n} + \frac{\partial c}{\partial x_n} \end{bmatrix}$$

Update the rules in Eq. subtracting a vector that points in direction of the steepest descent.

```
A = [[ 3.0 ,-0.5 ,0.0 ],[-0.5 ,2.0 ,-1.0 ],[ 0.0 ,-1.0 ,4.0 ]]  # Storage Matrix
b = [[ 1.0 ],[ 2.0 ],[ 3.0 ]]
c = [[ 4.0 ]]
x = [[ 0.0 ],[ 0.0 ],[ 0.0 ]]  # Initialization point
alpha = 0.1                    # Step/Learning rate
iter = 10                      # Iterations
for i in range(iter):
    x = x - alpha * (A.dot(x) + A.T.dot(x) + b)
print x
```

Question 1(b)

```
# Gradient Descent
def gradient_descent(A, b, x, alpha, iter):
    for i in range(iter):
        x = x - alpha * (A.dot(x) + A.T.dot(x) + b)
        print 'x = ', x, '\n'
    return x
print gradient_descent(A, b, x, alpha, iter)
```

```
x =  [[ 0.0 ]
 [ 0.0 ]
 [ 0.0 ]]

x =  [[ 0.1 ]
 [ 0.2 ]
 [ 0.3 ]]

x =  [[ 0.17 ]
 [ 0.34 ]
 [ 0.51 ]]

x =  [[ 0.233 ]
 [ 0.466 ]
 [ 0.699 ]]

x =  [[ 0.2911 ]
 [ 0.5822 ]
 [ 0.8733 ]]

x =  [[ 0.34733 ]
 [ 0.69466 ]
 [ 1.04199 ]]

x =  [[ 0.39933 ]
 [ 0.79866 ]
 [ 1.19799 ]]

x =  [[ 0.44933 ]
 [ 0.89866 ]
 [ 1.34799 ]]

x =  [[ 0.49933 ]
 [ 0.99866 ]
 [ 1.49799 ]]

x =  [[ 0.54933 ]
 [ 1.09866 ]
 [ 1.64799 ]]

[[ 0.54933 ]
 [ 1.09866 ]
 [ 1.64799 ]]
```

Question 1(c)

The next gradient 1-D(b) condition attempts to reduce f(x) by updating x as it tries to find a minimum point in the function f(x) by moving in the direction of the negative gradient. Although it does reduce f(x) from 4.0 to 0.0 in 10 iterations, it does not find the global minimum. The global minimum occurs at x = [0.0, 0.0, 0.0] and is 0.0 for this function. Although, not for all functions, and as we discuss later, it can be shown that the gradient descent will converge to the global minimum for convex functions.

```
# Gradient Descent
def gradient_descent(A, b, x, alpha, iter):
    for i in range(iter):
        x = x - alpha * (A.dot(x) + A.T.dot(x) + b)
        print 'x = ', x, '\n'
    return x
print gradient_descent(A, b, x, alpha, iter)
```

```
x =  [[ 0.0 ]
 [ 0.0 ]
 [ 0.0 ]]

x =  [[ 0.1 ]
 [ 0.2 ]
 [ 0.3 ]]

x =  [[ 0.17 ]
 [ 0.34 ]
 [ 0.51 ]]

x =  [[ 0.233 ]
 [ 0.466 ]
 [ 0.699 ]]

x =  [[ 0.2911 ]
 [ 0.5822 ]
 [ 0.8733 ]]

x =  [[ 0.34733 ]
 [ 0.69466 ]
 [ 1.04199 ]]

x =  [[ 0.39933 ]
 [ 0.79866 ]
 [ 1.19799 ]]

x =  [[ 0.44933 ]
 [ 0.89866 ]
 [ 1.34799 ]]

x =  [[ 0.49933 ]
 [ 0.99866 ]
 [ 1.49799 ]]

x =  [[ 0.54933 ]
 [ 1.09866 ]
 [ 1.64799 ]]

[[ 0.54933 ]
 [ 1.09866 ]
 [ 1.64799 ]]
```

Question 1(d)

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

df = pd.read_csv('Portland.csv')
# Feature Scaling and Mean Normalization
features = df.drop('Price(Dollars)', axis=1)
features = (features - features.mean()) / features.std()
# Prepending a column of ones
features = pd.concat([pd.DataFrame(np.ones(len(features)), columns=['Ones']),
features], axis=1)
# Making X and y variables
X = features.values
y = df['Price(Dollars)'].values
print df.head(10)
print features.head(10)
```

```
   Area(Sq. Ft.)  Bedrooms  Price(Dollars)
0           2104         3         399900
1           1600         3         329900
2           2400         3         369000
3           1416         2         232000
4           3000         4         539900
5           1985         4         299900
6           1534         3         314900
7           1427         3         198999
8           1380         3         212000
9           1494         3         242500

   Ones  Area(Sq. Ft.)  Bedrooms
0   0.0       0.130427 -0.226093
1   0.0       0.504190 -0.226093
2   0.0       0.502476 -0.226093
3   0.0      -0.735723 -1.554392
4   0.0       3.000851  1.102206
5   0.0       0.244731  1.102206
6   0.0       0.267959 -0.226093
7   0.0      -0.771659 -0.226093
8   0.0      -0.830989 -0.226093
9   0.0      -0.733775 -0.226093
```

Question 1(e)

```
# Cost Function
def compute_cost(X, y, theta):
    '''
    Take in a matrix of feature values and a vector
    of actual output values.
    Output the mean-squared error.
    '''
    m = len(y)
    return sum(sum((X.dot(theta) - y)**2))/(2*m)
print compute_cost(X, y, zeros(shape=(X.shape[1], 1)))
```

```
0.48936170213
```

Question 1(f)

```
# Gradient Descent
def gradient_descent(X, y, theta, alpha, iter):
    '''
    Take in a matrix of feature values, a vector
    of actual output values, a vector of initial
    parameter values, a learning rate, and the
    number of iterations you want to run.
    Output the final parameter values.
    '''
    m = len(y)
    for i in range(iter):
        theta = theta - (alpha/m) * (X.T.dot(X.dot(theta) - y))
    return theta
print gradient_descent(X, y, zeros(shape=(X.shape[1], 1)), 0.01, 1500)
```

```
[[ 340412.659574 ]
 [ 109447.796703 ]
 [  -6578.354854 ]]
```

Question 1(g)

```
# Normal Equation
def normal_eq(X, y):
    '''
    Take in a matrix of feature values and a vector
    of actual output values.
    Output the parameter values from the normal
    equation.
    '''
    return (inv(X.T.dot(X)).dot(X.T).dot(y))
print normal_eq(X, y)
```

```
[[ 340412.659574 ]
 [ 109447.796703 ]
 [  -6578.354854 ]]
```

Question 1(h)

```
import numpy as np
%matplotlib inline
Image.fromarray(np.ones(3,dtype=int)*255)
```

```
array([[255, 255, 255],
       [255, 255, 255],
       [255, 255, 255]])
```

```
# Feature Normalization
df = pd.read_csv('Portland.csv')
features = df.drop('Price(Dollars)', axis=1)
features = (features - features.mean()) / features.std()
features.head(10)
```

	Area(Sq. Ft.)	Bedrooms
0	0.130427	-0.226093
1	0.504190	-0.226093
2	0.502476	-0.226093
3	-0.735723	-1.554392
4	3.000851	1.102206
5	0.244731	1.102206
6	0.267959	-0.226093
7	-0.771659	-0.226093
8	-0.830989	-0.226093
9	-0.733775	-0.226093

```
# Prepending a column of ones
features = pd.concat([pd.DataFrame(np.ones(len(features)), columns=['Ones'],
dtype=int), features], axis=1)
features.head(10)
```

	Ones	Area(Sq. Ft.)	Bedrooms
0	1	0.130427	-0.226093
1	1	0.504190	-0.226093
2	1	0.502476	-0.226093
3	1	-0.735723	-1.554392
4	1	3.000851	1.102206
5	1	0.244731	1.102206
6	1	0.267959	-0.226093
7	1	-0.771659	-0.226093
8	1	-0.830989	-0.226093
9	1	-0.733775	-0.226093

```
# Making X and y variables
X = features.values
y = df['Price(Dollars)'].values
print X[:10]
print y[:10]
```

```
[[ 1.          0.130427 -0.226093]
 [ 1.          0.504190 -0.226093]
 [ 1.          0.502476 -0.226093]
 [ 1.         -0.735723 -1.554392]
 [ 1.          3.000851  1.102206]
 [ 1.          0.244731  1.102206]
 [ 1.          0.267959 -0.226093]
 [ 1.         -0.771659 -0.226093]
 [ 1.         -0.830989 -0.226093]
 [ 1.         -0.733775 -0.226093]]
[399900 329900 369000 232000 539900 299900 314900 198999 212000 242500]
```

Question 1(i)

```
# Cost Function
def compute_cost(X, y, theta):
    '''
    Take in a matrix of feature values and a vector
    of actual output values.
    Output the mean-squared error.
    '''
    m = len(y)
    return sum(sum((X.dot(theta) - y)**2))/(2*m)
print compute_cost(X, y, zeros(shape=(X.shape[1], 1)))
```

```
0.48936170213
```

Question 1(j)

```
# Gradient Descent
def gradient_descent(X, y, theta, alpha, iter):
    '''
    Take in a matrix of feature values, a vector
    of actual output values, a vector of initial
    parameter values, a learning rate, and the
    number of iterations you want to run.
    Output the final parameter values.
    '''
    m = len(y)
    for i in range(iter):
        theta = theta - (alpha/m) * (X.T.dot(X.dot(theta) - y))
    return theta
print gradient_descent(X, y, zeros(shape=(X.shape[1], 1)), 0.01, 1500)
```

```
[[ 340412.659574 ]
 [ 109447.796703 ]
 [  -6578.354854 ]]
```

Question 1(k)

```
# Normal Equation
def normal_eq(X, y):
    '''
    Take in a matrix of feature values and a vector
    of actual output values.
    Output the parameter values from the normal
    equation.
    '''
    return (inv(X.T.dot(X)).dot(X.T).dot(y))
print normal_eq(X, y)
```

```
[[ 340412.659574 ]
 [ 109447.796703 ]
 [  -6578.354854 ]]
```

Question 1(l)

```
import numpy as np
%matplotlib inline
Image.fromarray(np.ones(3,dtype=int)*255)
```

```
array([[255, 255, 255],
       [255, 255, 255],
       [255, 255, 255]])
```

```
# Feature Normalization
df = pd.read_csv('Portland.csv')
features = df.drop('Price(Dollars)', axis=1)
features = (features - features.mean()) / features.std()
features.head(10)
```

	Area(Sq. Ft.)	Bedrooms
0	0.130427	-0.226093
1	0.504190	-0.226093
2	0.502476	-0.226093
3	-0.735723	-1.554392
4	3.000851	1.102206
5	0.244731	1.102206
6	0.267959	-0.226093
7	-0.771659	-0.226093
8	-0.830989	-0.226093
9	-0.733775	-0.226093

```
# Prepending a column of ones
features = pd.concat([pd.DataFrame(np.ones(len(features)), columns=['Ones'],
dtype=int), features], axis=1)
features.head(10)
```

	Ones	Area(Sq. Ft.)	Bedrooms
0	1	0.130427	-0.226093
1	1	0.504190	-0.226093
2	1	0.502476	-0.226093
3	1	-0.735723	-1.554392
4	1	3.000851	1.102206
5	1	0.244731	1.102206
6	1	0.267959	-0.226093
7	1	-0.771659	-0.226093
8	1	-0.830989	-0.226093
9	1	-0.733775	-0.226093

```
# Making X and y variables
X = features.values
y = df['Price(Dollars)'].values
print X[:10]
print y[:10]
```

```
[[ 1.          0.130427 -0.226093]
 [ 1.          0.504190 -0.226093]
 [ 1.          0.502476 -0.226093]
 [ 1.         -0.735723 -1.554392]
 [ 1.          3.000851  1.102206]
 [ 1.          0.244731  1.102206]
 [ 1.          0.267959 -0.226093]
 [ 1.         -0.771659 -0.226093]
 [ 1.         -0.830989 -0.226093]
 [ 1.         -0.733775 -0.226093]]
[399900 329900 369000 232000 539900 299900 314900 198999 212000 242500]
```

Question 1(m)

```
import numpy as np
%matplotlib inline
Image.fromarray(np.ones(3,dtype=int)*255)
```

```
array([[255, 255, 255],
       [255, 255, 255],
       [255, 255, 255]])
```

```
# Feature Normalization
df = pd.read_csv('Portland.csv')
features = df.drop('Price(Dollars)', axis=1)
features = (features - features.mean()) / features.std()
features.head(10)
```

	Area(Sq. Ft.)	Bedrooms
0	0.130427	-0.226093
1	0.504190	-0.226093
2	0.502476	-0.226093
3	-0.735723	-1.554392
4	3.000851	1.102206
5	0.244731	1.102206
6	0.267959	-0.226093
7	-0.771659	-0.226093
8	-0.830989	-0.226093
9	-0.733775	-0.226093

```
# Prepending a column of ones
features = pd.concat([pd.DataFrame(np.ones(len(features)), columns=['Ones'],
dtype=int), features], axis=1)
features.head(10)
```

	Ones	Area(Sq. Ft.)	Bedrooms
0	1	0.130427	-0.226093
1	1	0.504190	-0.226093
2	1	0.502476	-0.226093
3	1	-0.735723	-1.554392
4	1	3.000851	1.102206
5	1	0.244731	1.102206
6	1	0.267959	-0.226093
7	1	-0.771659	-0.226093
8	1	-0.830989	-0.226093
9	1	-0.733775	-0.226093

```
# Making X and y variables
X = features.values
y = df['Price(Dollars)'].values
print X[:10]
print y[:10]
```

```
[[ 1.          0.130427 -0.226093]
 [ 1.          0.504190 -0.226093]
 [ 1.          0.502476 -0.226093]
 [ 1.         -0.735723 -1.554392]
 [ 1.          3.000851  1.102206]
 [ 1.          0.244731  1.102206]
 [ 1.          0.267959 -0.226093]
 [ 1.         -0.771659 -0.226093]
 [ 1.         -0.830989 -0.226093]
 [ 1.         -0.733775 -0.226093]]
[399900 329900 369000 232000 539900 299900 314900 198999 212000 242500]
```

Question 1(n)

```
# Cost Function
def compute_cost(X, y, theta):
    '''
    Take in a matrix of feature values and a vector
    of actual output values.
    Output the mean-squared error.
    '''
    m = len(y)
    return sum(sum((X.dot(theta) - y)**2))/(2*m)
print compute_cost(X, y, zeros(shape=(X.shape[1], 1)))
```

```
0.48936170213
```

Question 1(o)

```
# Gradient Descent
def gradient_descent(X, y, theta, alpha, iter):
    '''
    Take in a matrix of feature values, a vector
    of actual output values, a vector of initial
    parameter values, a learning rate, and the
    number of iterations you want to run.
    Output the final parameter values.
    '''
    m = len(y)
    for i in range(iter):
        theta = theta - (alpha/m) * (X.T.dot(X.dot(theta) - y))
    return theta
print gradient_descent(X, y, zeros(shape=(X.shape[1], 1)), 0.01, 1500)
```

```
[[ 340412.659574 ]
 [ 109447.796703 ]
 [  -6578.354854 ]]
```

Question 1(p)

```
# Normal Equation
def normal_eq(X, y):
    '''
    Take in a matrix of feature values and a vector
    of actual output values.
    Output the parameter values from the normal
    equation.
    '''
    return (inv(X.T.dot(X)).dot(X.T).dot(y))
print normal_eq(X, y)
```

```
[[ 340412.659574 ]
 [ 109447.796703 ]
 [  -6578.354854 ]]
```

Question 1(q) - Train & Test MSE

Let us plot n = 0:50 and experiment to the maximum the lowest and the minimum MSE losses.

```
# Plotting Cost Vs Iterations
# Cost Function
def compute_cost(X, y, theta):
    '''
    Take in a matrix of feature values and a vector
    of actual output values.
    Output the mean-squared error.
    '''
    m = len(y)
    return sum(sum((X.dot(theta) - y)**2))/(2*m)
# Gradient Descent
def gradient_descent(X, y, theta, alpha, iter):
    '''
    Take in a matrix of feature values, a vector
    of actual output values, a vector of initial
    parameter values, a learning rate, and the
    number of iterations you want to run.
    Output the final parameter values.
    '''
    m = len(y)
    J_history = zeros(shape=(iter, 1))
    for i in range(iter):
        theta = theta - (alpha/m) * (X.T.dot(X.dot(theta) - y))
        J_history[i, 0] = compute_cost(X, y, theta)
    return J_history
# Testing the gradient descent
alpha = 0.01
num_iters = 50
J_h = gradient_descent(X, y, zeros(shape=(X.shape[1], 1)), alpha, num_iters)
# Plot the convergence graph
plot(J_h, '-')
xlabel('Cost J')
xlabel('Iterations')
title('Figure 1: Cost Function')
show()
```

```
array([[ 1.        ,  0.130427, -0.226093],
       [ 1.        ,  0.504190, -0.226093],
       [ 1.        ,  0.502476, -0.226093],
       [ 1.        , -0.735723, -1.554392],
       [ 1.        ,  3.000851,  1.102206],
       [ 1.        ,  0.244731,  1.102206],
       [ 1.        ,  0.267959, -0.226093],
       [ 1.        , -0.771659, -0.226093],
       [ 1.        , -0.830989, -0.226093],
       [ 1.        , -0.733775, -0.226093]])
array([399900, 329900, 369000, 232000, 539900, 299900, 314900, 198999,
       212000, 242500])
array([[ 340412.659574,  340412.659574,  340412.659574,  340412.659574,
         340412.659574,  340412.659574,  340412.659574,  340412.659574,
         340412.659574,  340412.659574,  340412.659574,  340412.659574,
         
```