Machine Learning: Programming Exercise 1

Linear Regression

In this exercise, you will implement linear regression and get to see it work on data.

Files needed for this exercise

- `ex1.mlx` - MATLAB Live Script that steps you through the exercise
- ex1data1.txt - Dataset for linear regression with one variable
- ex1data2.txt - Dataset for linear regression with multiple variables
- `submit.m` - Submission script that sends your solutions to our servers
- *`warmUpExercise.m` - Simple example function in MATLAB
- *`plotData.m` - Function to display the dataset
- *`computeCost.m` - Function to compute the cost of linear regression
- *`gradientDescent.m` - Function to run gradient descent
- **`computeCostMulti.m` - Cost function for multiple variables
- **`gradientDescentMulti.m` - Gradient descent for multiple variables
- **`featureNormalize.m` - Function to normalize features
- **`normalEqn.m` - Function to compute the normal equations

indicates files you will need to complete

****indicates optional exercises***

Confirm that your Current Folder is set correctly

Click into this section, then click the 'Run section' button above. This will execute the `dir` command below to list the files in your Current Folder. The output should contain all of the files listed above and the 'lib' folder. If it does not, right-click the 'ex1' folder and select 'Open' before proceding or see the instructions in `README.mlx` for more details.

```
dir
```

```
.                       computeCostMulti.m      ex1data1.txt
gradientDescent.m       normalEqn.m             token.mat

..                      ex1.mlx                 ex1data2.txt
gradientDescentMulti.m  plotData.m              warmUpExercise.m

computeCost.m           ex1_companion.mlx       featureNormalize.m      lib
submit.m
```

Table of Contents

Linear Regression

Files needed for this exercise

Confirm that your Current Folder is set correctly

Before you begin

1. A simple MATLAB function

1.1 Submitting Solutions

2. Linear regression with one variable

2.1 Plotting the data

2.2 Gradient Descent

2.2.1 Update Equations

2.2.2 Implementation

2.2.3 Computing the cost

2.2.4 Gradient descent

2.3 Debugging

2.4 Visualizing

Optional Exercises:

3. Linear regression with multiple variables

3.1 Feature Normalization

3.2 Gradient Descent

3.2.1 Optional (ungraded) exercise: Selecting learning rates

3.3 Normal Equations

Submission and Grading

1. A simple MATLAB function

The first part of this script gives you practice with MATLAB syntax and the homework submission process. In the file `warmUpExercise.m`, you will find the outline of a MATLAB function. Modify it to return a 5 x 5 identity matrix by filling in the following code:

```
A = eye(5);
```

When you are finished, save `warmUpExercise.m`, then run the code contained in this section to call `warmUpExercise()`.

5x5 Identity Matrix:

```
warmUpExercise()
```

```
ans = 5×5

     1     0     0     0     0

     0     1     0     0     0

     0     0     1     0     0

     0     0     0     1     0

     0     0     0     0     1
```

You should see output similar to the following:

```
ans =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

You can toggle between right-hand-side output and in-line output for printing results and displaying figures inside a Live Script by selecting the appropriate box in the upper right of the Live Editor window.

2. Linear regression with one variable

In this part of this exercise, you will implement linear regression with one variable to predict profits for a food truck. Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet. The chain already has trucks in various cities and you have data for profits and populations from the cities. You would like to use this data to help you select which city to expand to next.

The file `ex1data1.txt` contains the dataset for our linear regression problem. The first column is the population of a city and the second column is the profit of a food truck in that city. A negative value for profit indicates a loss. This script has already been set up to load this data for you.

2.1 Plotting the data

Before starting on any task, it is often useful to understand the data by visualizing it. For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population). Many other problems that you will encounter in real life are multi-dimensional and can't be plotted on a 2-d plot.

Run the code below to load the dataset from the data file into the variables X and y:

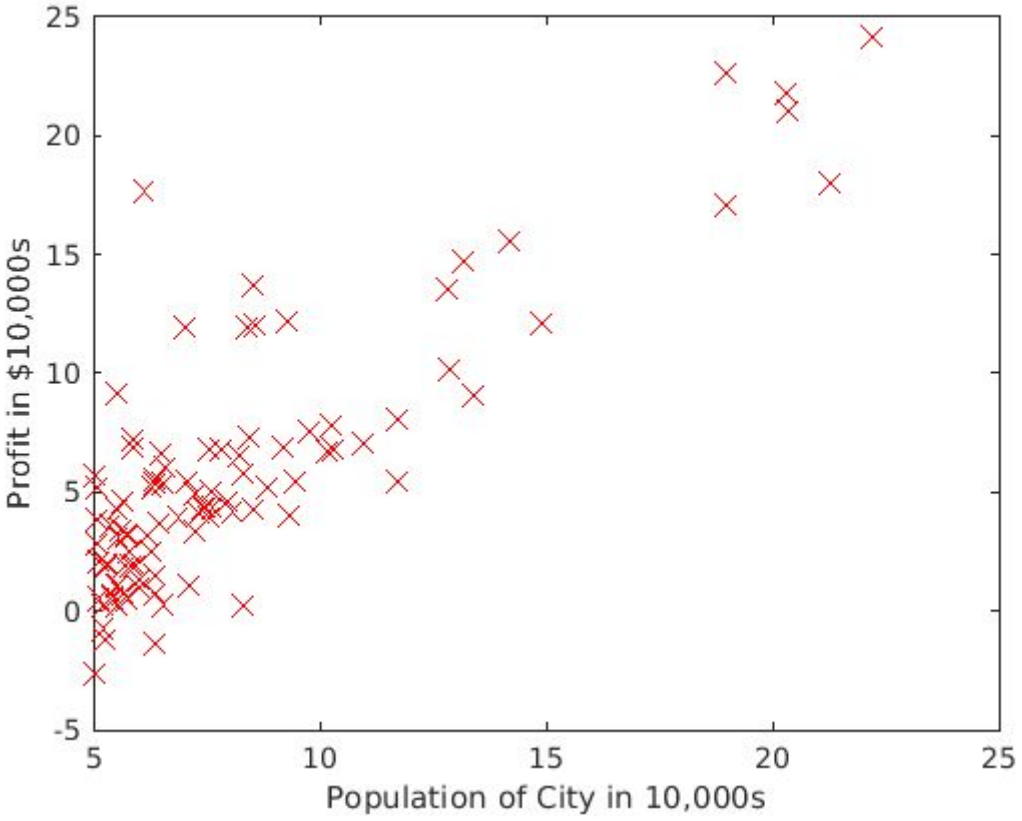
```
data = load('ex1data1.txt'); % read comma separated data
X = data(:, 1); y = data(:, 2);
```

Your job is to complete **`plotData.m`** to draw the plot; modify the file and fill in the following code:

```
plot(x, y, 'rx', 'MarkerSize', 10); % Plot the data
ylabel('Profit in $10,000s'); % Set the y-axis label
xlabel('Population of City in 10,000s'); % Set the x-axis label
```

Once you are finished, save `plotData.m`, and execute the code in this section which will call `plotData`.

```
plotData(X,y)
```



The resulting plot should appear as in Figure 1 below:

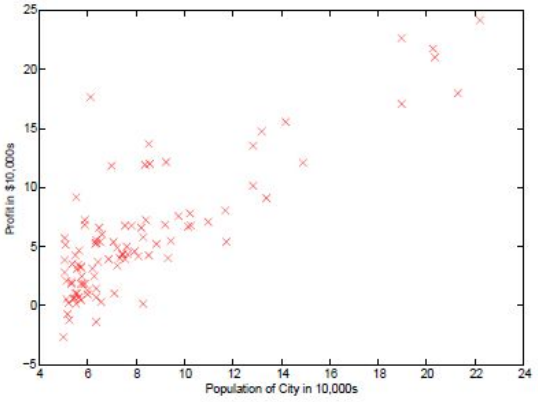


Figure 1: Scatter plot of training data

To learn more about the `plot` command, you can run the command **`help plot`** at the command prompt, type `plot()` inside the MATLAB Live Editor and click on the "(?)" tooltip, or you can search the MATLAB documentation for "plot". Note that to change the markers to red x's in the plot, we used the option: **`'rx'`** together with the **`plot`** command, i.e.,

```
plot(..,[your options here],..,'rx');
```

2.2 Gradient Descent

In this section, you will fit the linear regression parameters to our dataset using gradient descent.

2.2.1 Update Equations

The objective of linear regression is to minimize the cost function

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

where the hypothesis $h_\theta(x)$ is given by the linear model

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

Recall that the parameters of your model are the θ values. These are the values you will adjust to minimize cost $J(\theta)$. One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs the update

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \qquad (\text{simultaneously update } \theta_j \text{ for all } j)$$

With each step of gradient descent, your parameters j come closer to the optimal values that will achieve the lowest cost $J(\theta)$.

Implementation Note: We store each example as a row in the the `X` matrix in MATLAB. To take into account the intercept term (θ_0), we add an additional first column to **x** and set it to all ones. This allows us to treat θ_0 as simply another 'feature'.

2.2.2 Implementation

In this script, we have already set up the data for linear regression. In the following lines, we add another dimension to our data to accommodate the θ_0 intercept term. Run the code below to initialize the parameters to 0 and the learning rate `alpha` to 0.01.

```
m = length(X) % number of training examples
```

```
m = 97
```

```
X = [ones(m, 1), data(:,1)]; % Add a column of ones to x
```

```
theta = zeros(2, 1); % initialize fitting parameters

iterations = 1500;

alpha = 0.01;
```

2.2.3 Computing the cost $J(\theta)$

As you perform gradient descent to minimize the cost function $J(\theta)$, it is helpful to monitor the convergence by computing the cost. In this section, you will implement a function to calculate $J(\theta)$ so you can check the convergence of your gradient descent implementation.

Your next task is to complete the code in the file `computeCost.m`, which is a function that computes $J(\theta)$. As you are doing this, remember that the variables `X` and `y` are not scalar values, but matrices whose rows represent the examples from the training set.

Once you have completed the function definition, run this section. The code below will call `computeCost` once using θ initialized to zeros, and you will see the cost printed to the screen. You should expect to see a cost of 32.07 for the first output below:

```
% Compute and display initial cost with theta all zeros

computeCost(X, y, theta)
```

```
ans = 32.0727
```

Next we call `computeCost` again, this time with non-zero `theta` values as an additional test. You should expect to see an output of 54.24 below:

```
% Compute and display initial cost with non-zero theta

computeCost(X, y,[-1; 2])
```

```
ans = 54.2425
```

If the outputs above match the expected values, you can submit your solution for assessment. If the outputs do not match or you receive an error, check your cost function for mistakes, then rerun this section once you have addressed them.

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2.2.4 Gradient descent

Next, you will implement gradient descent in the file `gradientDescent.m`. The loop structure has been written for you, and you only need to supply the updates to θ within each iteration.

As you program, make sure you understand what you are trying to optimize and what is being updated. Keep in mind that the cost $J(\theta)$ is parameterized by the vector θ, not X and y. That is, we minimize the value of $J(\theta)$ by changing the values of the vector θ, not by changing X or y. Refer to the equations given earlier and to the video lectures if you are uncertain.

A good way to verify that gradient descent is working correctly is to look at the value of `J` and check that it is decreasing with each step. The starter code for `gradientDescent.m` calls `computeCost` on every iteration and prints the cost. Assuming you have implemented gradient descent and `computeCost` correctly, your value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm.

After you are finished, run this execute this section. The code below will use your final parameters to plot the linear fit. The result should look something like Figure 2 below:

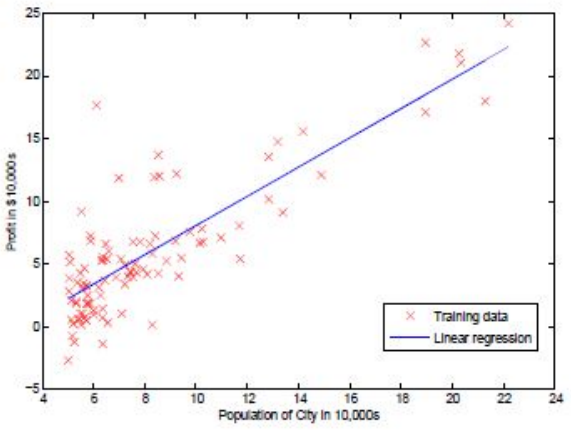


Figure 2: Training data with linear regression fit

Your final values for θ will also be used to make predictions on profits in areas of 35,000 and 70,000 people.

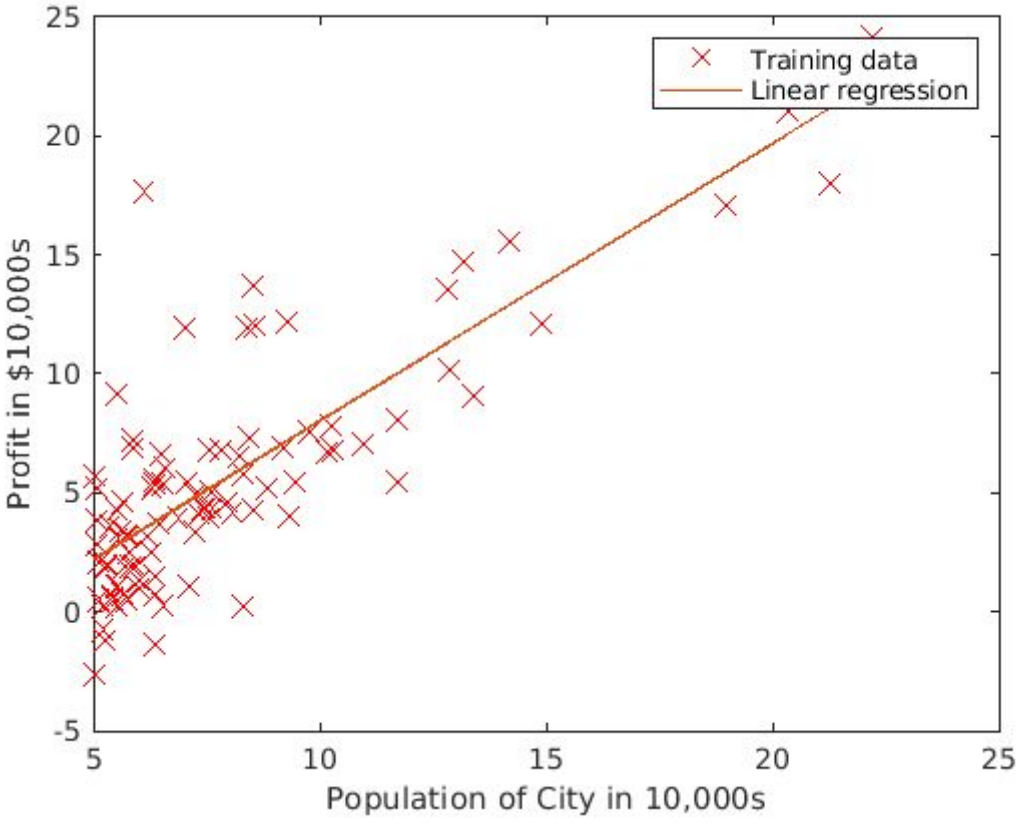
```
% Run gradient descent:
% Compute theta
theta = gradientDescent(X, y, theta, alpha, iterations);
```

```
% Print theta to screen
% Display gradient descent's result
```

```
fprintf('Theta computed from gradient
descent:\n%f,\n%f',theta(1),theta(2))
```

```
Theta computed from gradient descent:

-3.630291,

1.166362
```

```
% Plot the linear fit

hold on; % keep previous plot visible

plot(X(:,2), X*theta, '-')

legend('Training data', 'Linear regression')

hold off % don't overlay any more plots on this figure
```




```
% Predict values for population sizes of 35,000 and 70,000

predict1 = [1, 3.5] *theta;

fprintf('For population = 35,000, we predict a profit of %f\n',
predict1*10000);
```

```
For population = 35,000, we predict a profit of 4519.767868
```

```
predict2 = [1, 7] * theta;

fprintf('For population = 70,000, we predict a profit of %f\n',
predict2*10000);
```

```
For population = 70,000, we predict a profit of 45342.450129
```

Note the way that the lines above use matrix multiplication, rather than explicit summation or looping, to calculate the predictions. This is an example of *code vectorization* in MATLAB.

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2.3 Debugging

Here are some things to keep in mind as you implement gradient descent:

- MATLAB array indices start from one, not zero. If you're storing θ_0 and θ_1 in a vector called **`theta`**, the values will be **`theta(1)`** and **`theta(2)`**.
- If you are seeing many errors at runtime, inspect your matrix operations to make sure that you're adding and multiplying matrices of compatible dimensions. Printing the dimensions of variables with the `size` command will help you debug.
- By default, MATLAB interprets math operators to be matrix operators. This is a common source of size incompatibility errors. If you don't want matrix multiplication, you need to add the "dot" notation to specify this to MATLAB. For example, `A*B` does a matrix multiply, while `A.*B` does an element-wise multiplication.

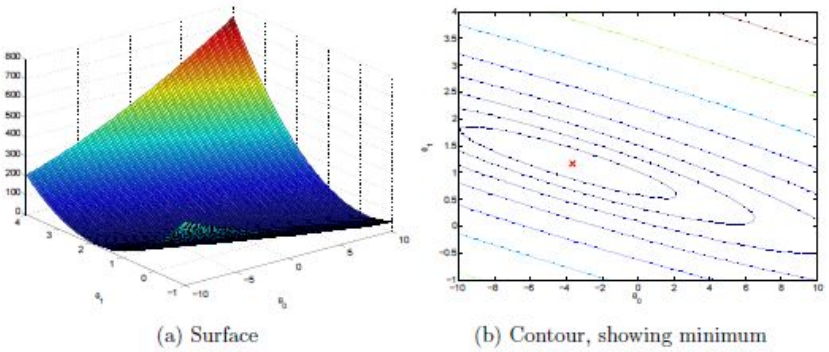
2.4 Visualizing $J(\theta)$

To understand the cost function $J(\theta)$ better, you will now plot the cost over a 2-dimensional grid of θ_0 and θ_1 values. You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images.

In the next step, there is code set up to calculate $J(\theta)$ over a grid of values using the `computeCost` function that you wrote.

```
% Visualizing J(theta_0, theta_1):
% Grid over which we will calculate J
theta0_vals = linspace(-10, 10, 100);
theta1_vals = linspace(-1, 4, 100);

% initialize J_vals to a matrix of 0's
J_vals = zeros(length(theta0_vals), length(theta1_vals));

% Fill out J_vals
for i = 1:length(theta0_vals)
   for j = 1:length(theta1_vals)
      t = [theta0_vals(i); theta1_vals(j)];
      J_vals(i,j) = computeCost(X, y, t);
   end
end
```

After the code above is executed, you will have a 2-D array of $J(\theta)$ values. The code below will then use these values to produce surface and contour plots of $J(\theta)$ using the **`surf`** and **`contour`** commands. Run the code in this section now. The resulting plots should look something like the figure below.



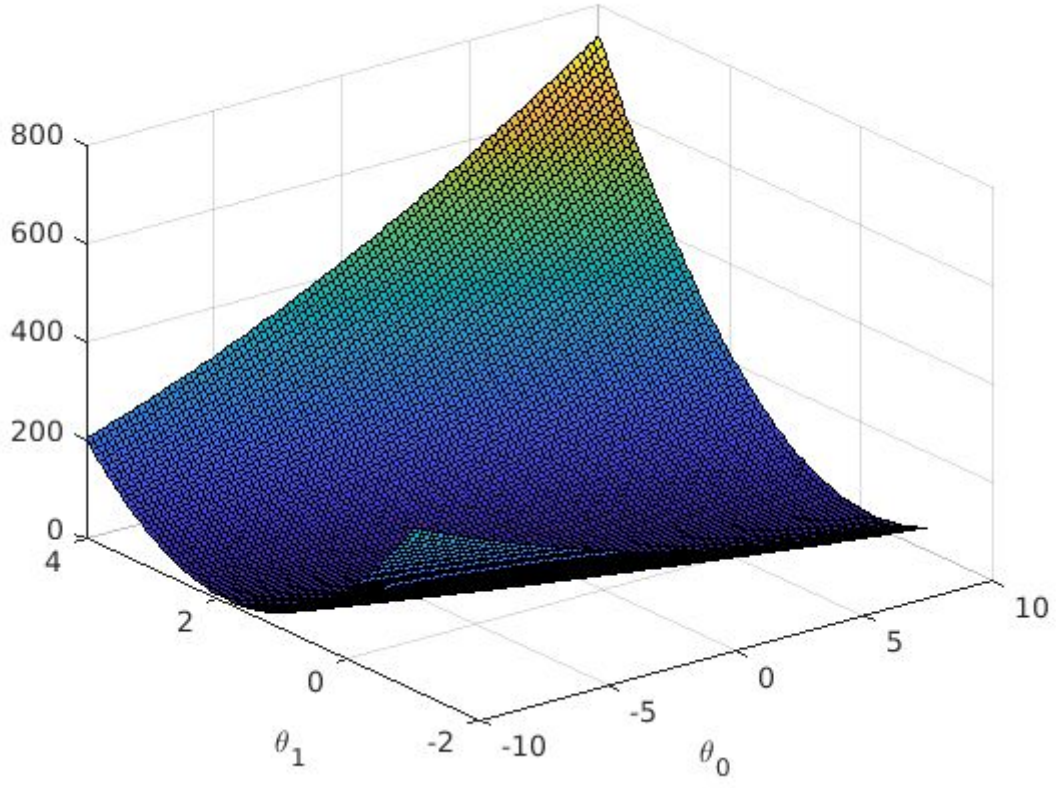
(a) Surface (b) Contour, showing minimum

Figure 3: Cost function $J(\theta)$

```
% Because of the way meshgrids work in the surf command, we need to
```

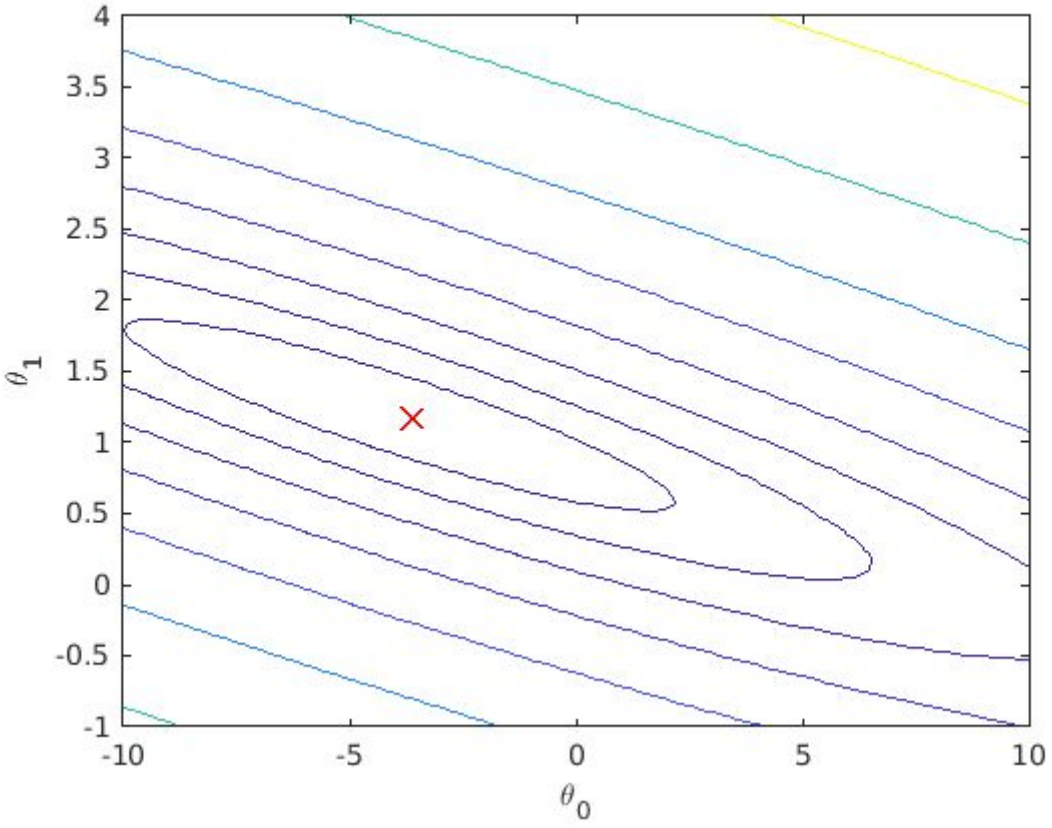
```
% transpose J_vals before calling surf, or else the axes will be
flipped

J_vals = J_vals';
```

```
% Surface plot

figure;

surf(theta0_vals, theta1_vals, J_vals)

xlabel('\theta_0'); ylabel('\theta_1');
```




```
% Contour plot

figure;

% Plot J_vals as 15 contours spaced logarithmically between 0.01 and
100

contour(theta0_vals, theta1_vals, J_vals, logspace(-2, 3, 20))

xlabel('\theta_0'); ylabel('\theta_1');

hold on;
```

```
plot(theta(1), theta(2), 'rx', 'MarkerSize', 10, 'LineWidth', 2);
hold off;
```



The purpose of these graphs is to show you that how $J(\theta)$ varies with changes in θ_0 and θ_1. The cost function $J(\theta)$ is bowl-shaped and has a global mininum. (This is easier to see in the contour plot than in the 3D surface plot). This minimum is the optimal point for θ_0 and θ_1, and each step of gradient descent moves closer to this point.

Optional Exercises:

If you have successfully completed the material above, congratulations! You now understand linear regression and should able to start using it on your own datasets. For the rest of this programming exercise, we have included the following optional exercises. These exercises will help you gain a deeper understanding of the material, and if you are able to do so, we encourage you to complete them as well.

3. Linear regression with multiple variables

In this part, you will implement linear regression with multiple variables to predict the prices of houses. Suppose you are selling your house and you want to know what a good market price would be. One way to do this is to first collect information on recent houses sold and make a model of housing prices.

The file `ex1data2.txt` contains a training set of housing prices in Portland, Oregon. The first column is the size of the house (in square feet), the second column is the number of bedrooms, and the third column is the price of the house. Run this section now to preview the data.

```
% Load Data
data = load('ex1data2.txt');
X = data(:, 1:2);
y = data(:, 3);
m = length(y);
```

```
% Print out some data points
% First 10 examples from the dataset
fprintf(' x = [%.0f %.0f], y = %.0f \n', [X(1:10,:) y(1:10,:)]');
```

```
x = [2104 3], y = 399900
x = [1600 3], y = 329900
x = [2400 3], y = 369000
x = [1416 2], y = 232000
x = [3000 4], y = 539900
x = [1985 4], y = 299900
x = [1534 3], y = 314900
x = [1427 3], y = 198999
x = [1380 3], y = 212000
x = [1494 3], y = 242500
```

The remainder of this script has been set up to help you step through this exercise.

3.1 Feature Normalization

This section of the script will start by loading and displaying some values from this dataset. By looking at the values, note that house sizes are about 1000 times the number of bedrooms.

When features differ by orders of magnitude, first performing feature scaling can make gradient descent converge much more quickly.

Your task here is to complete the code in `featureNormalize.m` to:

- Subtract the mean value of each feature from the dataset.
- After subtracting the mean, additionally scale (divide) the feature values by their respective "standard deviations".

The standard deviation is a way of measuring how much variation there is in the range of values of a particular feature (most data points will lie within ± 2 standard deviations of the mean); this is an alternative to taking the range of values (*max* - *min*). In MATLAB, you can use the `std` function to compute the standard deviation. For example, inside `featureNormalize.m`, the quantity `X(:,1)` contains all the values of x_1 (house sizes) in the training set, so `std(X(:,1))` computes the standard deviation of the house sizes. At the time that `featureNormalize.m` is called, the extra column of 1's corresponding to $x_0 = 1$ has not yet been added to `X` (see the code below for details).

You will do this for all the features and your code should work with datasets of all sizes (any number of features / examples). Note that each column of the matrix **X** corresponds to one feature. When you are finished with `featureNormailize.m`, run this section to normailze the features of the housing dataset.

```
% Scale features and set them to zero mean
[X, mu, sigma] = featureNormalize(X);
```

Implementation Note: When normalizing the features, it is important to store the values used for normalization - the mean value and the standard deviation used for the computations. After learning the parameters from the model, we often want to predict the prices of houses we have not seen before. Given a new `x` value (living room area and number of bedrooms), we must first normalize `x` using the mean and standard deviation that we had previously computed from the training set.

Now that we have normailzed the features, we again add a column of ones corresponding to θ_0 to the data matrix `X`.

```
% Add intercept term to X
```

```
X = [ones(m, 1) X];
```

3.2 Gradient Descent

Previously, you implemented gradient descent on a univariate regression problem. The only difference now is that there is one more feature in the matrix `X`. The hypothesis function and the batch gradient descent update rule remain unchanged.

You should complete the code in `computeCostMulti.m` and `gradientDescentMulti.m` to implement the cost function and gradient descent for linear regression *with multiple variables*. If your code in the previous part (single variable) already supports multiple variables, you can use it here too.

Make sure your code supports any number of features and is well-vectorized. You can use the command `size(X,2)` to find out how many features are present in the dataset.

We have provided you with the following starter code below that runs gradient descent with a particular learning rate (`alpha`). Your task is to first make sure that your functions `computeCost` and `gradientDescent` already work with this starter code and support multiple variables.

Implementation Note: In the multivariate case, the cost function can also be written in the following vectorized form:

$$J(\theta) = \frac{1}{2m}\left(X\theta - \vec{y}\right)^T\left(X\theta - \vec{y}\right)$$

where

$$X = \begin{bmatrix} -\left(x^{(1)}\right)^T- \\ -\left(x^{(2)}\right)^T- \\ \vdots \\ -\left(x^{(m)}\right)^T- \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

The vectorized version is efficient when you're working with numerical computing tools like MATLAB. If you are an expert with matrix operations, you can prove to yourself that the two forms are equivalent.

```
% Run gradient descent

% Choose some alpha value

alpha = 0.1;
```

```
num_iters = 400;
```

```
% Init Theta and Run Gradient Descent

theta = zeros(3, 1);

[theta, ~] = gradientDescentMulti(X, y, theta, alpha, num_iters);
```

```
% Display gradient descent's result

fprintf('Theta computed from gradient
descent:\n%f,\n%f',theta(1),theta(2))
```

```
Theta computed from gradient descent:

340412.659574,

110631.048958
```

Finally, you should complete and run the code below to predict the price of a 1650 sq-ft, 3 br house using the value of `theta` obtained above.

Hint: At prediction, make sure you do the same feature normalization. Recall that the first column of `X` is all ones. Thus, it does not need to be normalized.

```
% Estimate the price of a 1650 sq-ft, 3 br house

% ====================== YOUR CODE HERE ======================

price = [1 (1650-mu(1))/sigma(1) (3-mu(2))/sigma(2)]*theta % Enter
your price formula here
```

```
price = 2.9308e+05
```

```
% ============================================================
```

```
fprintf('Predicted price of a 1650 sq-ft, 3 br house (using gradient
descent):\n $%f', price);
```

```
Predicted price of a 1650 sq-ft, 3 br house (using gradient descent):

$293081.464622
```

3.2.1 Optional (ungraded) exercise: Selecting learning rates

In this part of the exercise, you will get to try out dierent learning rates for the dataset and find a learning rate that converges quickly. You can change the learning rate by modifying the code below and changing the part of the code that sets the learning rate.

The code below will call your `gradientDescent` function and run gradient descent for about 50 iterations at the chosen learning rate. The function should also return the history of $J(\theta)$ values in a vector `J`. After the last iteration, the code plots the J values against the number of the iterations. If you picked a learning rate within a good range, your plot should look similar Figure 4 below.

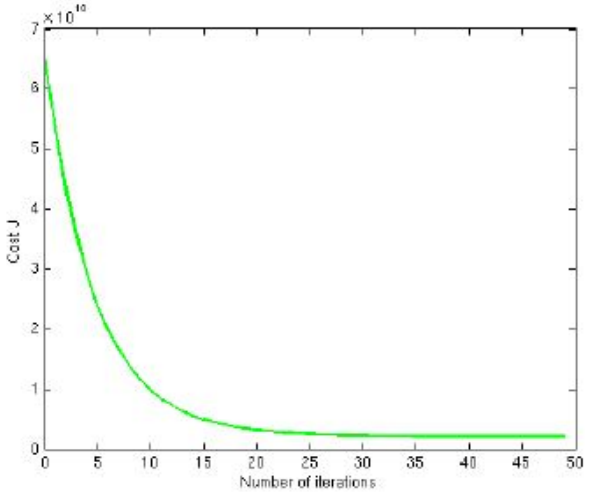


Figure 4: Convergence of gradient descent with an appropriate learning rate

If your graph looks very different, especially if your value of $J(\theta)$ increases or even blows up, use the control to adjust your learning rate and try again. We recommend trying values of the learning rate on a log-scale, at multiplicative steps of about 3 times the previous value (i.e., 0.3, 0.1, 0.03, 0.01 and so on). You may also want to adjust the number of iterations you are running if that will help you see the overall trend in the curve.

Implementation Note: If your learning rate is too large, $J(\theta)$ can diverge and 'blow up', resulting in values which are too large for computer calculations. In these situations, MATLAB will tend to return `NaNs`. `NaN` stands for 'not a number' and is often caused by undefined operations that involve $\pm\infty$.

MATLAB Tip: To compare how different learning learning rates affect convergence, it's helpful to plot `J` for several learning rates on the same figure. In MATLAB, this can be done by performing gradient descent multiple times with a `hold on` command between plots. Make

sure to use the `hold off` command when you are done plotting in that figure. Concretely, if you've tried three different values of `alpha` (you should probably try more values than this) and stored the costs in `J1`, `J2` and `J3`, you can use the following commands to plot them on the same figure:

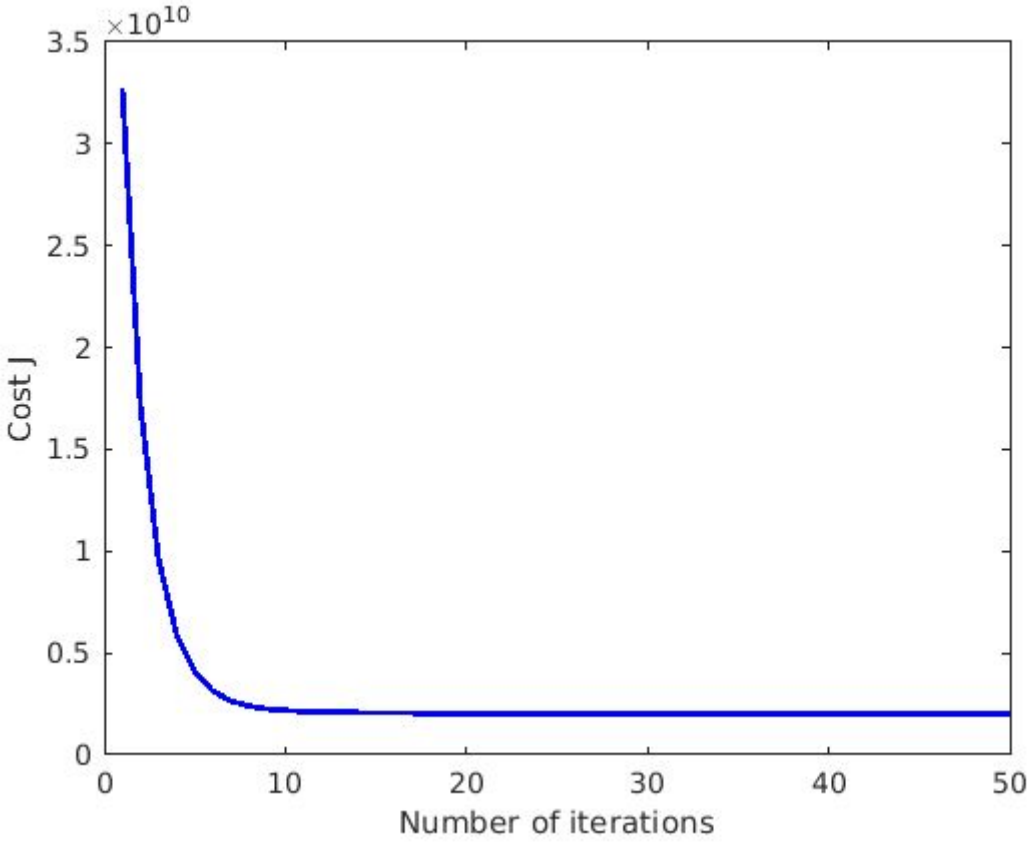
```
plot(1:50, J1(1:50), 'b');
hold on
plot(1:50, J2(1:50), 'r');
plot(1:50, J3(1:50), 'k');
hold off
```

The final arguments `'b'`,`'r'`, and `'k'` specify different colors for the plots. If desired, you can use this technique and adapt the code below to plot multiple convergence histories in the same plot.

```
% Run gradient descent:
% Choose some alpha value
alpha = 0.3;
num_iters = 50;
```

```
% Init Theta and Run Gradient Descent
theta = zeros(3, 1);
[~, J_history] = gradientDescentMulti(X, y, theta, alpha, num_iters);
```

```
% Plot the convergence graph
plot(1:num_iters, J_history, '-b', 'LineWidth', 2);
xlabel('Number of iterations');
ylabel('Cost J');
```



Notice the changes in the convergence curves as the learning rate changes. With a small learning rate, you should find that gradient descent takes a very long time to converge to the optimal value. Conversely, with a large learning rate, gradient descent might not converge or might even diverge!

Using the best learning rate that you found, run the section of code below, which will run gradient descent until convergence to find the final values of θ. Next, use this value of θ to predict the price of a house with 1650 square feet and 3 bedrooms. You will use value later to check your implementation of the normal equations. Don't forget to normalize your features when you make this prediction!

```
% Run gradient descent
% Replace the value of alpha below best alpha value you found above
alpha = 0.3;
num_iters = 50;
```

```
% Init Theta and Run Gradient Descent
```

```
theta = zeros(3, 1);

[theta, ~] = gradientDescentMulti(X, y, theta, alpha, num_iters);
```

```
% Display gradient descent's result

fprintf('Theta computed from gradient
descent:\n%f\n%f',theta(1),theta(2))
```

```
Theta computed from gradient descent:

340412.653452

110572.961931
```

```
% Estimate the price of a 1650 sq-ft, 3 br house. You can use the
same

% code you entered ealier to predict the price

% ====================== YOUR CODE HERE ======================
```

```
price = [1 (1650-mu(1))/sigma(1) (3-mu(2))/sigma(2)]*theta; % Enter
your price formula here
```

```
% ============================================================
```

```
fprintf('Predicted price of a 1650 sq-ft, 3 br house (using gradient
descent):\n $%f', price);
```

```
Predicted price of a 1650 sq-ft, 3 br house (using gradient descent):

$293094.098122
```

3.3 Normal Equations

In the lecture videos, you learned that the closed-form solution to linear regression is

$$\theta = (X^T X)^{-1} X^T \vec{y}$$

Using this formula does not require any feature scaling, and you will get an exact solution in one calculation: there is no "loop until convergence" like in gradient descent.

Complete the code in `normalEqn.m` to use the formula above to calculate θ, then run the code in this section. Remember that while you don't need to scale your features, we still need

to add a column of 1's to the X matrix to have an intercept term (θ_0) . Note that the code below will add the column of 1's to X for you.

```
% Solve with normal equations:
% Load Data
data = csvread('ex1data2.txt');
X = data(:, 1:2);
y = data(:, 3);
m = length(y);
```

```
% Add intercept term to X
X = [ones(m, 1) X];
```

```
% Calculate the parameters from the normal equation
theta = normalEqn(X, y);
```

```
% Display normal equation's result
fprintf('Theta computed from the normal equations:\n%f\n%f',
theta(1),theta(2));
```

```
Theta computed from the normal equations:
89597.909544
139.210674
```

Optional (ungraded) exercise: Now, once you have found θ using this method, use it to make a price prediction for a 1650-square-foot house with 3 bedrooms. You should find that gives the same predicted price as the value you obtained using the model fit with gradient descent (in Section 3.2.1).

```
% Estimate the price of a 1650 sq-ft, 3 br house.
% ====================== YOUR CODE HERE ======================
```

```
price = [1 (1650-mu(1))/sigma(1) (3-mu(2))/sigma(2)]*theta;; % Enter
your price forumla here
```

```
% ================================================================
```

```
fprintf('Predicted price of a 1650 sq-ft, 3 br house (using normal
equations):\n $%f', price);
```

```
Predicted price of a 1650 sq-ft, 3 br house (using normal equations):

$91490.957664
```

Submission and Grading