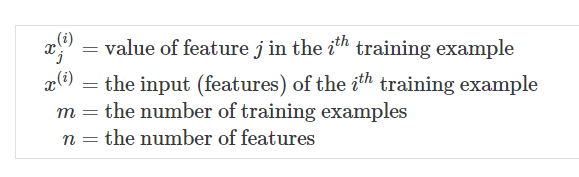
Part 1 Multivariate Linear Regression

Multiple Features

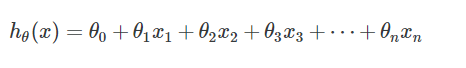
**Note:** [7:25 - *θT* is a 1 by (n+1) matrix and not an (n+1) by 1 matrix]

Linear regression with multiple variables is also known as "multivariate linear regression".

We now introduce notation for equations where we can have any number of input variables.

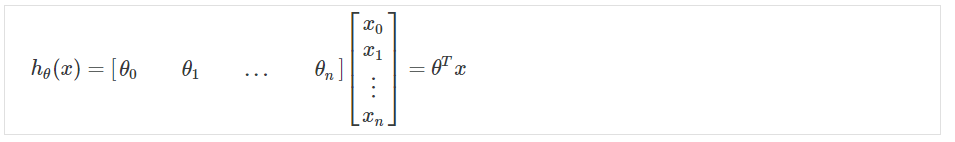


The multivariable form of the hypothesis function accommodating these multiple features is as follows:

**

In order to develop intuition about this function, we can think about *θ*0 as the basic price of a house, *θ*1 as the price per square meter, *θ*2 as the price per floor, etc. *x*1 will be the number of square meters in the house, *x*2 the number of floors, etc.

Using the definition of matrix multiplication, our multivariable hypothesis function can be concisely represented as:



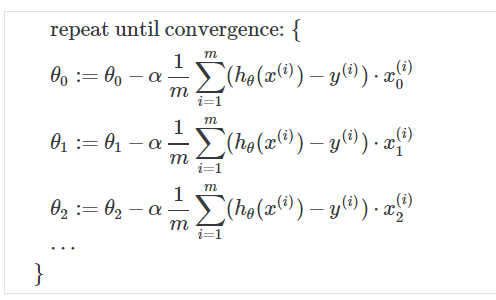
This is a vectorization of our hypothesis function for one training example; see the lessons on vectorization to learn more.

Remark: Note that for convenience reasons in this course we assume *C:\Users\Alava\AppData\Local\Temp\1501422082(1).png*. This allows us to do matrix operations with theta and x. Hence making the two vectors '*θ*' and *x*(*i*) match each other element-wise (that is, have the same number of elements: n+1).]

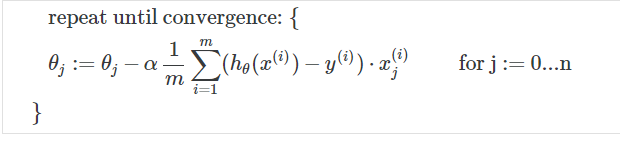
# Gradient Descent For Multiple Variables

## ****Gradient Descent for Multiple Variables****

The gradient descent equation itself is generally the same form; we just have to repeat it for our 'n' features:



In other words:



The following image compares gradient descent with one variable to gradient descent with multiple variables:



# Gradient Descent in Practice I - Feature Scaling

**Note:** [6:20 - The average size of a house is 1000 but 100 is accidentally written instead]

We can speed up gradient descent by having each of our input values in roughly the same range. This is because θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven.

The way to prevent this is to modify the ranges of our input variables so that they are all roughly the same. Ideally:

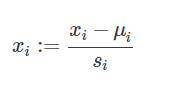
−1 ≤ *x*(*i*) ≤ 1

or

−0.5 ≤ *x*(*i*) ≤ 0.5

These aren't exact requirements; we are only trying to speed things up. The goal is to get all input variables into roughly one of these ranges, give or take a few.

Two techniques to help with this are **feature scaling** and **mean normalization**. Feature scaling involves dividing the input values by the range (i.e. the maximum value minus the minimum value) of the input variable, resulting in a new range of just 1. Mean normalization involves subtracting the average value for an input variable from the values for that input variable resulting in a new average value for the input variable of just zero. To implement both of these techniques, adjust your input values as shown in this formula:

**

Where *μi* is the **average** of all the values for feature (i) and *si* is the range of values (max - min), or *si* is the standard deviation.

Note that dividing by the range, or dividing by the standard deviation, give different results. The quizzes in this course use range - the programming exercises use standard deviation.

For example, if *xi* represents housing prices with a range of 100 to 2000 and a mean value of 1000, then,

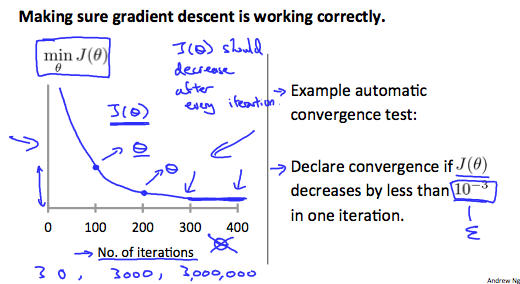
*C:\Users\Alava\AppData\Local\Temp\1501422157(1).png*

Gradient Descent in Practice II - Learning Rate

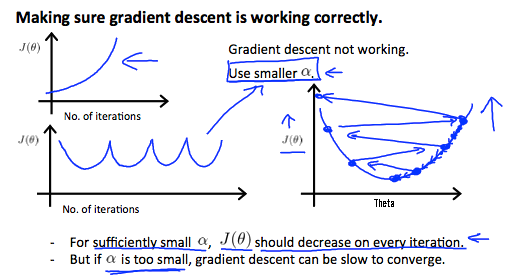
**Note:** [5:20 - the x -axis label in the right graph should be *θ* rather than No. of iterations ]

**Debugging gradient descent.** Make a plot with *number of iterations* on the x-axis. Now plot the cost function, J(θ) over the number of iterations of gradient descent. If J(θ) ever increases, then you probably need to decrease α.

**Automatic convergence test.** Declare convergence if J(θ) decreases by less than E in one iteration, where E is some small value such as C:\Users\Alava\AppData\Local\Temp\1501422253(1).png However in practice it's difficult to choose this threshold value.



It has been proven that if learning rate α is sufficiently small, then J(θ) will decrease on every iteration.



To summarize:

If *α* is too small: slow convergence.

If *α* is too large: may not decrease on every iteration and thus may not converge.

# Features and Polynomial Regression

We can improve our features and the form of our hypothesis function in a couple different ways.

We can **combine** multiple features into one. For example, we can combine *x*1 and *x*2 into a new feature *x*3 by taking *x*1⋅*x*2.

### ****Polynomial Regression****

Our hypothesis function need not be linear (a straight line) if that does not fit the data well.

We can **change the behavior or curve** of our hypothesis function by making it a quadratic, cubic or square root function (or any other form).

For example, if our hypothesis function is *hθ*(*x*)=*θ*0+*θ*1*x*1 then we can create additional features based on *x*1, to get the quadratic function *C:\Users\Alava\AppData\Local\Temp\1501422325(1).png* or the cubic function *C:\Users\Alava\AppData\Local\Temp\1501422345(1).png*

In the cubic version, we have created new features *x*2 and *x*3 where *C:\Users\Alava\AppData\Local\Temp\1501422367(1).png*

To make it a square root function, we could do: *C:\Users\Alava\AppData\Local\Temp\1501422399(1).png*

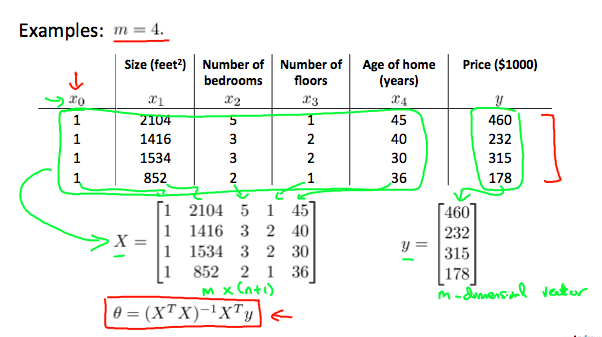
One important thing to keep in mind is, if you choose your features this way then feature scaling becomes very important.

eg. if *x*1 has range 1 - 1000 then range of *C:\Users\Alava\AppData\Local\Temp\1501422419(1).png* becomes 1 - 1000000 and that of *C:\Users\Alava\AppData\Local\Temp\1501422437(1).png* becomes 1 - 1000000000

Normal Equation

**Note:** [8:00 to 8:44 - The design matrix X (in the bottom right side of the slide) given in the example should have elements x with subscript 1 and superscripts varying from 1 to m because for all m training sets there are only 2 features *x*0 and *x*1. 12:56 - The X matrix is m by (n+1) and NOT n by n. ]

Gradient descent gives one way of minimizing J. Let’s discuss a second way of doing so, this time performing the minimization explicitly and without resorting to an iterative algorithm. In the "Normal Equation" method, we will minimize J by explicitly taking its derivatives with respect to the θj ’s, and setting them to zero. This allows us to find the optimum theta without iteration. The normal equation formula is given below:

*C:\Users\Alava\AppData\Local\Temp\1501422500(1).png*

There is **no need** to do feature scaling with the normal equation.

The following is a comparison of gradient descent and the normal equation:

|  |  |
| --- | --- |
| **Gradient Descent** | **Normal Equation** |
| Need to choose alpha | No need to choose alpha |
| Needs many iterations | No need to iterate |
| C:\Users\Alava\AppData\Local\Temp\1501422535(1).png | C:\Users\Alava\AppData\Local\Temp\1501422549(1).png |
| Works well when n is large | Slow if n is very large |

With the normal equation, computing the inversion has complexity C:\Users\Alava\AppData\Local\Temp\1501422566(1).png. So if we have a very large number of features, the normal equation will be slow. In practice, when n exceeds 10,000 it might be a good time to go from a normal solution to an iterative process.

# Normal Equation Noninvertibility

When implementing the normal equation in octave we want to use the 'pinv' function rather than 'inv.' The 'pinv' function will give you a value of *θ* even if *C:\Users\Alava\AppData\Local\Temp\1501422609(1).png* is not invertible.

If *C:\Users\Alava\AppData\Local\Temp\1501422609(1).png* is **noninvertible,** the common causes might be having :

* Redundant features, where two features are very closely related (i.e. they are linearly dependent)
* Too many features (e.g. m ≤ n). In this case, delete some features or use "regularization" (to be explained in a later lesson).

Solutions to the above problems include deleting a feature that is linearly dependent with another or deleting one or more features when there are too many features.

## Subject: Confused about "h(x) = theta' \* x" vs. "h(x) = X \* theta?"

### ****Text:****

The lectures and exercise PDF files are based on Prof. Ng's feeling that novice programmers will adapt to for-loop techniques more readily than vectorized methods. So the videos (and PDF files) are organized toward processing one training example at a time. The course uses column vectors (in most cases), so h (a scalar for one training example) is theta' \* x.

Lower-case x typically indicates a single training example.

The more efficient vectorized techniques always use X as a matrix of all training examples, with each example as a row, and the features as columns. That makes X have dimensions of (m x n). where m is the number of training examples. This leaves us with h (a vector of all the hypothesis values for the entire training set) as X \* theta, with dimensions of (m x 1).

X (as a matrix of all training examples) is denoted as upper-case X.

Throughout this course, dimensional analysis is your friend.

## Subject: Tips from the Mentors: submit problems and fixing program errors

### ****Text:****

This post contains some frequently-used tips about the course, and to help get your programs working correctly.

### ****The Most Important Tip:****

Search the forum before posting a new question. If you've got a question, the chances are that someone else has already posted it, and received an answer. Save time for yourself and the Forum users by searching for topics before posting a new one.

### ****Running your scripts:****

At the Octave/Matlab command line, you do not need to include the ".m" portion of the script file name. If you include the ".m", you'll get an error message about an invalid indexing operation. So, run the Exercise 1 script by typing just "ex1" at the command line.

You also do not need to include parenthesis () when using the submit script. Just type "submit".

You cannot execute your functions by simply typing the name. All of the functions you will work on require a set of parameter values, enter between a set of parenthesis. Your three methods of testing your code are:

1 - use an exercise script, such as "ex1"

2 - use a Unit Test (see below) where you type-in the entire command line including the parameters.

3 - use the submit script.

### ****Making the grader happy:****

The submit grader uses a different test case than what is in the PDF file. These test cases use a different size of data set and are more sensitive to small errors than the ex test cases. Your code must work correctly with any size of data set.

Your functions must handle the general case. This means:

- You should avoid using hard-coded array indexes.

- You should avoid having fixed-length arrays and matrices.

It is very common for students to think that getting the same answer as listed in the PDF file means they should get full credit from the grader. This is a false hope. The PDF file is just one test case. The grader uses a different test case.

Also, the grader does not like your code to send any additional outputs to the workspace. So, every line of code should end with a semicolon.

### ****Getting Help:****

When you want help from the Forum community, please use this two-step procedure:

1 - Search the Forum for keywords that relate to your problem. Searching by the function name is a good start.

2 - If you don't find a suitable thread, then do this:

2a - Find the unit tests for that exercise (see below), and run the appropriate test. Attempt to debug your code.

2b - Take a screen capture of your whole console workspace (including the command line), and post it to the forum, along with any other useful information (computer type, Octave/Matlab version, other tests you've tried, etc).

### Debugging:

If your code runs but gives the wrong answers, you can insert a "keyboard" command in your script, just before the function ends. This will cause the program to exit to the debugger, so you can inspect all your variables from the command line. This often is very helpful in analysing math errors, or trying out what commands to use to implement your function.

There are additional test cases and tutorials listed in pinned threads under "All Course Discussions". The test cases are especially helpful in debugging in situations where you get the expected output in ex but get no points or an error when submitting.

### Unit Tests:

Each programming assignment has a "Discussions" area in the Forum. In this section you can often find "unit tests". These are additional test cases, which give you a command to type, and provides the expected results. It is always a good idea to test your functions using the unit tests before submitting to the grader.

If you run a unit test and do not get the correct results, you can most easily get help on the forums by posting a screen capture of your workspace - including the command line you entered, and the results.

### Having trouble submitting your work to the grader?:

- This section will need to be supplemented with info appropriate to the new submission system. If you run the submit script and get a message that your identity can't be verified, be sure that you have logged-in using your Coursera account email and your Programming Assignment submission password.

- If you get the message "submit undefined", first check that you are in the working directory where you extracted the files from the ZIP archive. Use "cd" to get there if necessary.

- If the "submit undefined" error persists, or any other "function undefined" messages appear, try using the "addpath(pwd)" command to add your present working directory (pwd) to the Octave execution path.

-If the submit script crashes with an error message, please see the thread "Mentor tips for submitting your work" under "All Course Discussions".

-The submit script does not ask for what part of the exercise you want to submit. It automatically grades any function you have modified.

### Found some errata in the course materials?

This course material has been used for many previous sessions. Most likely all of the errata has been discovered, and it's all documented in the 'Errata' section under 'Supplementary Materials'. Please check there before posting errata to the Forum.

Error messages with fmincg()

The "short-circuit" warnings are due to use a change in the syntax for conditional expressions (| and & vs || and &&) in the newer versions of Matlab. You can edit the fmincg.m file and the warnings may be resolved.

### Warning messages about "automatic broadcasting"?

See [this](https://www.gnu.org/software/octave/doc/interpreter/Broadcasting.html) link for info.

### Warnings about "divide by zero"

These are normal in some of the exercises, and do not represent a problem in your function. You can ignore them - Octave senses the issue and substitutes a +Inf or -Inf value so your program continues to execute.