Octave Tutorial

Andrew Ng (video tutorial from "Machine Learning" class)

Transcript written* by *José Soares Augusto*, May 2012 (V1.0c)

1 Basic Operations

In this video I'm going to teach you a programming language, Octave, which will allow you to implement quickly the learning algorithms presented in the "Machine Learning" course.

Octave is the language I recommend, after having teached in the past "Machine Learning" (ML) supported by several languages (C++, Java, Python/Numpy, R). Students are more productive, and learn better, when using high-level languages like Octave, compared to the others I mentioned.

Often people prototype ML algorithms in Octave (which is a very good prototyping language) and only after succeeding in that step they proceed into large-scale implementations of the ML algorithms in Java, C++ or other low-level languages, which are more efficient than Octave when running time is the concern.

By using Octave, students get huge time savings in learning and implementing ML algorithms.

The most common prototyping languages used in ML are Octave, Matlab, Python/Numpy and R. Octave is free and open source. Matlab is also good, and could be used, but it is very expensive. People using R or Python/Numpy also produce good results, but their development pace is slower due to the more complex syntax in R or Python/Numpy.

So, Octave is definitely recommended for this course.

In this video I'll go quickly into a list of Octave commands and show you the range of things that you can do in Octave. The course site has a transcript of what I'll do, so you can go there after seeing this video and study Octave's commands behavior. Also you should install Octave in your computer, and download the transcripts of these videos and then try them by yourself.

1.1 Basic algebra in Octave

Let's get started. This is my Octave prompt (command window). I'll do some **elementary arithetic operations**. They are shown in the figure below, along with my Octave prompt.

*With help from the English subtitles which are available in the "Machine Learning" site.

```
GNU Octave, version 3.6.1
Copyright (C) 2012 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type `warranty'.

Octave was configured for "i686-pc-mingw32".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/help-wanted.html

Read http://www.octave.org/bugs.html to learn how to submit bug reports.

For information about changes from previous versions, type `news'.

 - Use `pkg list' to see a list of installed packages.
 - MSYS shell available (C:\Math\Octave-3.6.1\msys).
 - Graphics backend: qt.

octave-3.6.1.exe:1> 5+6
ans =  11
octave-3.6.1.exe:2> 3-2
ans =  1
octave-3.6.1.exe:3> 5*8
ans =  40
octave-3.6.1.exe:4> 1/2
ans =  0.50000
octave-3.6.1.exe:5> 2^6
ans =  64
octave-3.6.1.exe:6>
```

We can also do **logical operations** in Octave, as shown below. Note that the special character `'%'` in the console starts the beginning of a comment, which will end in the end of the line.

```
octave-3.6.1.exe:6>
octave-3.6.1.exe:6>
octave-3.6.1.exe:6> 1 == 2  % false
ans = 0
octave-3.6.1.exe:7> 1 ~= 2  % true
ans =  1
octave-3.6.1.exe:8> 1 && 0  % AND
ans = 0
octave-3.6.1.exe:9> 1 || 0  % OR
ans =  1
octave-3.6.1.exe:10> xor(1,0)
ans =  1
octave-3.6.1.exe:11> PS1('>> ');
>>
```

```
>>
>> a = 3
a =  3
>> a = 3;  % semicolon supressing output
>>
>> b = 'hi';
>> b
b = hi
>> c = (3>=1);
>> c
c =  1
>> a = pi;
>> a
a =  3.1416
>> disp(a);
 3.1416
>> disp(sprintf('2 decimals: %0.2f', a))
2 decimals: 3.14
>> disp(sprintf('6 decimals: %0.6f', a))
6 decimals: 3.141593
>>
>>
>> format long
>> a
a =  3.14159265358979
>> format short
>> a
a =  3.1416
>>
```

The `'octave-3.6.1.exe:6>'` text is the Octave prompt. This **prompt can be changed** with the command `PS1()`. In the window above it was changed to `'>> '`.

Now we are going to talk about **Octave variables**.

We define `a=3`. The variable here is `a`. The console echoes the commands: if we don't want echoing, we end the commands with a semicolon, `';'`.

We can do **string assignment**. Above, variable `b` is a string. Also, we can assign logical values: thus `'c=(3>=1)'` assigns 1 (a synonym of *true*) to `c` because that condition '`3>=1`' is true.

To **display variables**, we can write them in the console and do *'RET'* ('return' or 'enter') in the keyboard, we can use the command `'disp(variable)'` and we can also use the `sprintf()` command for formatting the variable and displaying it in a more controlled way. The function `sprintf()` is well known to C programmers. The substring `'%0.2f'` indicates to `sprintf()` how it should display in the string the floating point number given in the argument, `a`. In this example, it should be displayed with 2 decimals after the dot `'.'`.

In the console are shown examples for displaying 2 and 6 decimals of `pi` – which in is in Octave the name of π, obviously.

The command `'format long'` can be applied to define that the **displaying of numbers** should be done with many digits; the command `'format short'` restores the default display format, which has few digits.

1.2 Vectors and matrices

Matrix `A` has 2×3 dimension. The organization of matrices in Octave is by lines. Elements in the same line are separated either by spaces or by commas `','`. Lines are separated by semicolons `';'`. The matrix can be entered in the console as a whole, or line by line. See below.

```
>> A = [1 2; 3 4; 5 6]
A =

   1   2
   3   4
   5   6

>> A = [1 2;
> 3 4;
> 5 6]
A =

   1   2
   3   4
   5   6

>> v = [1 2 3]
v =

   1   2   3

>> v = [1; 2; 3]
v =

   1
   2
   3
```

```
>> v = 1:0.1:2
v =

 Columns 1 through 7:

    1.0000    1.1000    1.2000    1.3000    1.4000    1.5000    1.6000

 Columns 8 through 11:

    1.7000    1.8000    1.9000    2.0000

>> v = 1:6
v =

   1   2   3   4   5   6

>> ones(2,3)
ans =

   1   1   1
   1   1   1

>> C = 2 * ones(2,3)
C =

   2   2   2
   2   2   2

>> C = [2 2 2; 2 2 2]
C =

   2   2   2
   2   2   2
```

Vectors can be entered as a $1 \times n$ matrix. Above, `v=[1 2 3]` is first defined as a 1×3 matrix. However, in practical work vectors are usually columns, so `v=[1; 2; 3]`, which produces a column vector, would in general be the choice to represent a vector in Octave, and thus vectors are the same as $n \times 1$ matrices.

Vectors (or matrices) can be **defined by compreension**, by using a *range* and a *step*. In the example, `v=1:0.1:2` defines a line vector with elements starting at 1, and stepping by 0.1 until the end value, which is 2, is found: thus, the range is the interval [1,2] and the step is 0.1.

If only the starting and ending value are given, then Octave assumes the step to be 1. Thus, '`v=1:6`' creates a line-vector with the list of integers between 1 and 6 (these limits included).

```
>> w = ones(1,3)
w =

   1   1   1

>> w = zeros(1,3)
w =

   0   0   0

>> w = rand(1,3)
w =

   0.91188   0.89231   0.37795

>> rand(3,3)
ans =

   0.850414   0.710288   0.635635
   0.390259   0.114534   0.791434
   0.398196   0.257380   0.097012

>> rand(3,3)
ans =

   0.53842   0.38122   0.73889
   0.82536   0.30469   0.67932
   0.60520   0.60370   0.94407
```

Now it's time for some **special matrix creation commands**.

The command `'ones(m,n)'` produces a $m \times n$ matrix filled with **ones**. Multiplying a constant by a matrix, which is allowed in Octave, for example `'C=2*ones(2,3)'`, corresponds to multyplying *all* the elements in the matrix by that constant.

The command `'zeros(m,n)'` creates a matrix of dimensions $m \times n$ filled with **zeros**.

The command `'rand(m,n)'` generates an $m \times n$ **random matrix**. The elements of the matrix are drawn from a **continous uniform distribution** with 0 and 1 as lower and upper limits of the distribution. Note that each time you generate a random matrix it comes with different values, because these are being drawn from a (pseudo) random number generator, always running "inside the PC", always changing the output numbers.

The command `'randn(m,n)'` is just like `rand()`, but instead of using a uniform distribution for generating the elements it uses a **Gaussian distribution** with zero average ($\mu = 0$) and variance σ^2 equal to 1 (or standard deviation σ equal to 1, because $\sigma = \sigma^2 = 1$). That is, the values of the elements in the matrix retrieved by `'randn(m,n)'` are drawn from a distribution $\mathcal{N}(0, 1)$.

Using compact matrix-producing commands like `'ones()'`, `'zeros()'` and `'rand()'` allows very large data structures to be produced on the fly in Octave. Care has to be taken to avoid them being displayed in the console when they are created – this means you should use the semicolon `';'` to terminate the large matrix-producing commands.

The command `'w=-6+sqrt(10)*(randn(1,10000))'` produces a line-vector with 10000 elements. These elements correspond to a Gaussian distribution of average $\mu = -6$ and standard deviation $\sigma = \sqrt{10}$ or, which is equivalent, a variance σ^2 equal to 10.

```
>> randn(1,3)
ans =

  -0.20070  -1.08215   0.14973

>> w = -6 +sqrt(10)*(randn(1,10000));  % produces a 1E4 size line vector
>> hist(w)
>> hist(w,50)
>> eye(4)
ans =

Diagonal Matrix

   1   0   0   0
   0   1   0   0
   0   0   1   0
   0   0   0   1

>> I = eye(6)
I =

Diagonal Matrix

   1   0   0   0   0   0
   0   1   0   0   0   0
   0   0   1   0   0   0
   0   0   0   1   0   0
   0   0   0   0   1   0
   0   0   0   0   0   1
```

The command `'hist(w)'` creates on-the-fly an histogram of that vector (see Fig. 1).

The number of bins in the histogram can be controlled through an additional attribute sent to the command `hist()`. Thus, `'hist(w,50)'` creates the histogram of the vector `w` with 50 bins (see Fig. 1).

Finally we arrive to the last matrix-generating command we are going to see in this video.

So, `'eye(n)'` generates an $n \times n$ **identity matrix** ('*eye*' reads approximately as the beginning of the word '*identity*'.) See the command windows for examples.

Finally, there is in Octave a **very helpful command:** the `'help'` command.

For instance, `'help eye'` brings a window with lots of information about the command `eye()`. Also, `'help rand'` will bring lots of information about the command `rand()`. In general, `'help <octave_command>'` will display information about the command referred to. Of course, `'help help'` will bring you more tips on how you shall use Octave help.

The basic operations in Octave have been presented, and so you should be capable of using them. In the next video we will talk about more sofisticated commands in Octave and how one moves data around.

2 Moving Data Around

This section is about moving data around: how Octave reads machine learning data, how we can insert them in matrices, how do we manipulate them, how do we save them, how do we operate with data.

Let's buid again the matrix `A`, with dimension 3×2. The `'size(matrix)'` command returns the **size of a matrix**. In fact `'size(matrix)'` returns a 1×2 matrix (or line-vector), where the first element is the number of

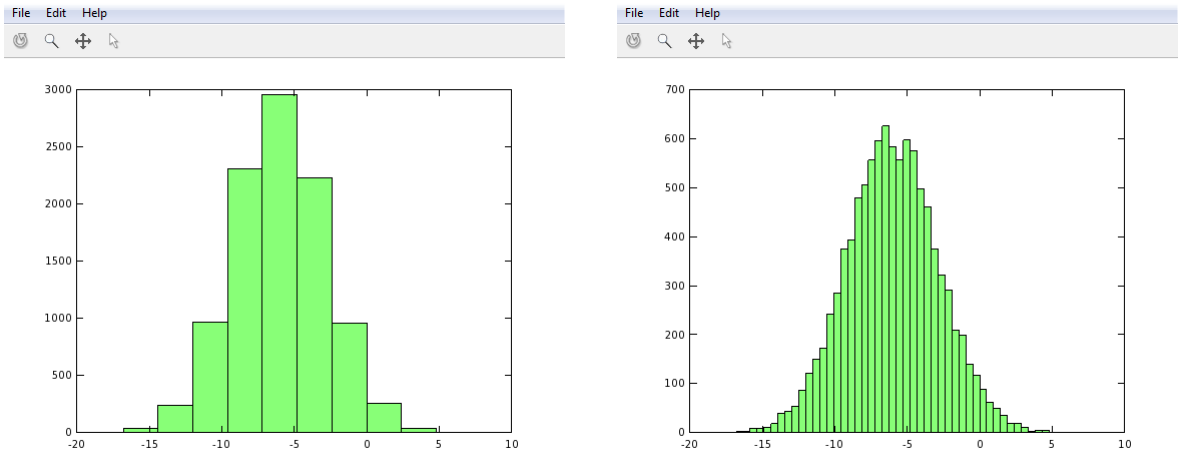


Figure 1: Histograms of a Gaussian random vector with different number of bins. In the right histogram there are 50 bins.

lines in `matrix`, and the second is the number of columns in `matrix`. Thus '`sr=size(A)`' creates a line-vector such that `sr[1]` is 3 and `sr[2]` is 2.

```
octave-3.6.1.exe:1> A = [1 2; 3 4; 5 6]
A =

   1   2
   3   4
   5   6

octave-3.6.1.exe:2> size(A)
ans =

   3   2

octave-3.6.1.exe:3> sz = size(A)
sz =

   3   2

octave-3.6.1.exe:4> size(sz)
ans =

   1   2

octave-3.6.1.exe:5> size(A,1)
ans =  3
octave-3.6.1.exe:6> size(A,2)
ans =  2
```

```
octave-3.6.1.exe:5> v = [1 2 3 4]
v =

   1   2   3   4

octave-3.6.1.exe:6> length(v)
ans =  4
octave-3.6.1.exe:7> length(A)
ans =  3
octave-3.6.1.exe:8> length([1;2;3;4;5])
ans =  5
```

We can call `size()` with the index of the dimension we wish to know. Thus '`size(A,1)`' is 3 (the same value of `sr[1]`) and '`size(A,2)`' gives the number of columns of `A` (it uses the second dimension index, which is `sr[2]`), which is 2 in this case.

If you have a vector `v`, say `v=[1 2 3 4]`, then '`length(v)`' gives the larger dimension of `v`.

We can do the same to the already defined matrix `A`, with dimension 3×2. In this case, '`lenght(A)`' gives 3, the number of lines in `A`, which is its larger dimension.

However, usually `lenght()` is only applied to vectors, because it's of no much use and can be confusing when applied to matrices.

Now let's talk about loading data and finding data in the computer file system.

The '`pwd`' command (familiar to Unix and Linux users, and meaning '*print working directory*') shows the directory in the computer where our Octave window is currently "pointing to" (and fetching data from.)

The '`cd`' command allows one to change directory. Below I changed the Octave work environment to my 'Desktop' directory.

The command '`ls`' lists the files in the current directory, which is my 'Desktop' in this example.

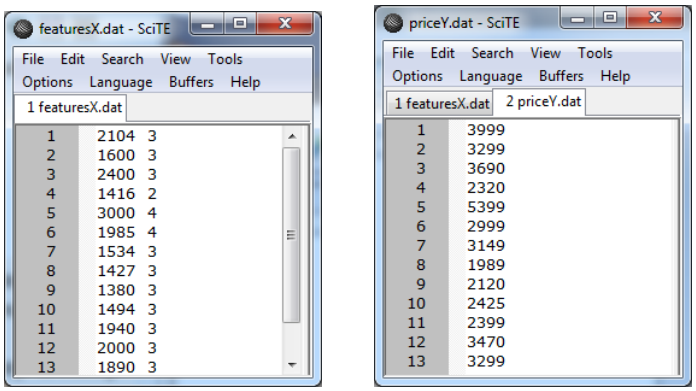


Figure 2: Snapshot of datafiles: *featuresX.dat* and *priceY.dat*.

```
>> pwd
ans = C:\Math\Octave-3.6.1
>> cd 'C:\Users\jasa\Desktop'
>> pwd
ans = C:\Users\jasa\Desktop
>> ls
 Volume in drive C has no label.
 Volume Serial Number is XXXX

 Directory of C:\Users\jasa\Desktop

[.]                         kompozer.lnk
[..]                        Launch TopSpice Demo.lnk
5Spice.lnk                  Notepad++.lnk
AbiWord 2.8.lnk             PARI.lnk
```

Now in the Desktop there are some files picked from ML problems data. One file is *featuresX.dat* which has two columns – corresponding to the houses' areas and number of rooms data (a snapshot of the file is shown in Fig.2, grabbed from Andrew Ng's video images.)

The price of the houses is in the file *priceY.dat*, also shown in Fig. 2. Note that there are thirteen values in each of the *.dat* files, while in the original ones there are 47 values (according to Andrew's own words. Our two files in Fig. 2 are not synchronized with Andrew's values, regarding prices and area and number of rooms, but they do well as examples of Andrew's files.)

Thus, *featuresX.dat* and *priceY.dat* are files with machine learning training data.

We can **load files in Octave** using the `load` command. Thus, '`load featuresX.dat`' and '`load priceY.dat`' do load the mentioned data files.

We can use the `load` command also by giving as argument the name of the files *as strings*.

So, "`load('featuresX.dat')`" is equivalent to '`load featuresX.dat`'.

```
>> load featuresX.dat
>> load priceY.dat
>> load('featuresX.dat')
>> who
Variables in the current scope:

ans        featuresX  priceY

>> featuresX
featuresX =

   2104      3
   1600      3
   2400      3
   1416      2
   3000      4
   1985      4
   1534      3
   1427      3
   1380      3
   1494      3
   1940      3
   2000      3
   1890      3

>> _
```

```
>> size(featuresX)
ans =

   13    2

>> size(priceY)
ans =

   13    1

>> whos
Variables in the current scope:

   Attr Name           Size                     Bytes  Class
   ==== ====           ====                     =====  =====
        ans            1x2                         16  double
        featuresX     13x2                        208  double
        priceY        13x1                        104  double

Total is 41 elements using 328 bytes

>> clear featuresX
>> whos
Variables in the current scope:

   Attr Name        Size                     Bytes  Class
   ==== ====        ====                     =====  =====
        ans         1x2                         16  double
        priceY     13x1                        104  double

Total is 15 elements using 120 bytes
```

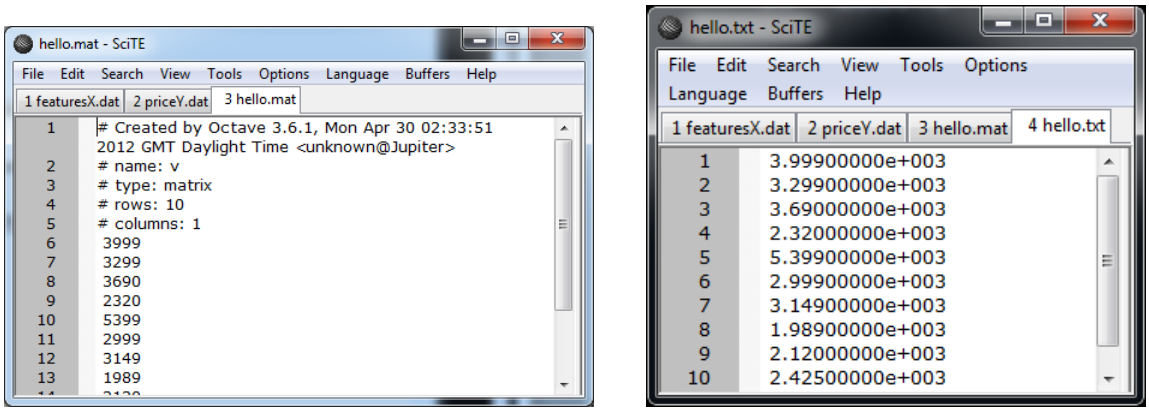


Figure 3: Vector `v` with prices, saved as a *.mat* file (left) and as an ASCII text file (right). The *.mat* file follows a pattern for data saving which was defined by the Matlab tool.

In Octave, **strings** are given inside a pair of '...'; for example, *'featuresX.dat'* is a string.

The '`who`' command **shows all variables active in the Octave session**. We can see their values by entering their name in the command prompt and doing RET. We do this to *featuresX* and see the contents of our file *featuresX.dat* listed in the Octave window, because they have been imported to the vector `featuresX` which has the same name of the file (but the suffix).

We can see the size of *featuresX*, and this gives '13 2'; so it is like a matrix with 13 lines and 2 columns. In the same way, *priceY* is a column vector with size 13. It also can be listed in the console by writing `'priceY'` and doing RET.

The `'whos'` command also shows the **variables in the current session**, but gives **details** about them: the type (double, char, logical,...), size (e. g. 13×1) and the memory space they use.

Sometimes we want to **get rid of some variables**. The `'clear'` command does just that. In the above window we execute `'clear featuresX'` and that variable disappears from our workspace, what is confirmed by the subsequent `'whos'` command.

Now let's see **how to save data**. We create the vector `'v=priceY(1:10)'` which has a slice of *priceY*. Then, `'whos'` shows us that new vector also in the workspace.

Note that we **access elements or slices of vectors or matrices** by writing comma-separated indices inside curved[1] parenthesis '()'. E.g., `priceY(3)`, or `priceY(1:10)` access a vector element and a slice, respectively.

```
>> v = priceY(1:10)
v =

   3999
   3299
   3690
   2320
   5399
   2999
   3149
   1989
   2120
   2425

>> whos
Variables in the current scope:

   Attr Name        Size                     Bytes  Class
   ==== ====        ====                     =====  =====
        ans         1x2                         16  double
        priceY     13x1                        104  double
        v          10x1                         80  double

Total is 25 elements using 200 bytes
```

Now lets save `v`. The command to do that is precisely `'save'`. So, `'save hello.mat v;'` creates a new file *hello.mat* on the 'Desktop' (our current directory, if you recall). This file has the contents of our saved vector `v`, but with some more information added (size, creation date, etc..., see Fig. 3.)

Now if I run the command `'clear'` with no variables in the argument, **all the workspace variables are cleared**, what is confirmed by the subsequent `'whos'` command.

[1]Most computer languages use square brackets '[]' for indexing vectors and matrices.

```
>> save hello.mat v;
>> clear
>> whos
>> load hello.mat
>> whos
Variables in the current scope:

   Attr Name        Size                     Bytes  Class
   ==== ====        ====                     =====  =====
        v           10x1                        80  double

Total is 10 elements using 80 bytes

>> v
v =

   3999
   3299
   3690
   2320
   5399
   2999
   3149
   1989
   2120
   2425

>> save hello.txt v -ascii  % save as text (ASCII)
```

By executing `'load hello.mat'` we **recover the vector** `v` which was saved before in that file. It enters our workspace and can be displayed.

If we want to **save the vector in text format** we run the command `'save hello.txt v -ascii'`. The flag `-ascii` indicates to Octave that it shall write `v` in the file *hello.txt* in a human-friendly format – the ASCII format. This *hello.txt* file is shown in Fig. 3.

So we discussed how to save data. Now let's see more on how to **manipulate data inside matrices**. We create the 3×2 matrix `A` again. Now let's see how we do **indexing in matrices**. So `A(3,2)` gives us the element in the third line and second column of `A` – which is 6 in this example.

The expression `A(2,:)` grabs everything in the second row of `A`. That is, it is the complete second row in A. The colon `':'` means "everything along that row or column". Similarly, `A(:,2)` gives the second column of A.

```
>> A = [1 2; 3 4; 5 6]
A =

   1   2
   3   4
   5   6

>> A(3,2)
ans =  6
>> A(2,:)  % ":" means everything along that row/column
ans =

   3   4

>> A(:,2)
ans =

   2
   4
   6
```

We can do more complicated **'slicing' of matrices**. We can give vectors with the indices of rows or columns which we want to select. For instance, `A([1 3],:)` gives the complete (i.e., including all the columns) lines 1 and 3 from the matrix `A`. These slicing operations are not very common in practice.

We can use slicing to do **assignments to blocks of matrices** also. So, `A(:,2)=[10; 11; 12]` assigns that 3×1 vector `[10; 11; 12]` to the second column of `A`.

```
>> A([1 3], :)
ans =

   1   2
   5   6

>> A(:,2) = [10; 11; 12]
A =

    1   10
    3   11
    5   12

>> A = [A , [100; 101; 102]]  % append another colun to right of A
A =

     1    10   100
     3    11   101
     5    12   102

>> size(A)
ans =

   3   3
```

The size of the matrices can be increased dynamically. So '`A = [A , [100; 101; 102]]`' adds another column to the matrix `A`. So now `A` is a larger 3×3 matrix, with nine elements.

Finally, one neat trick. The command `'A(:)'` **transforms** `A` from a 3×3 **matrix into a column vector** with nine elements – a 9×1 vector (or matrix). This is a somewhat special case syntax.

```
>> A(:)  % put all elements of A into a single vector
ans =

     1
     3
     5
    10
    11
    12
   100
   101
   102

>> A = [1 2; 3 4; 5 6];
>> B = [11 12; 13 14; 15 16]
B =

   11   12
   13   14
   15   16

>> C = [A B]
C =

    1    2   11   12
    3    4   13   14
    5    6   15   16
```

```
>> C=[A ; B]
C =

    1    2
    3    4
    5    6
   11   12
   13   14
   15   16

>> size(C)
ans =

   6   2

>> [A, B]
ans =

    1    2   11   12
    3    4   13   14
    5    6   15   16
```

Just a couple more examples. I set again `A=[1 2; 3 4; 5 6]` and `B=[11 12; 13 14; 15 16]`. I can create a new matrix `C=[A B]` which is the concatenation of `A` and `B`. I'm taking these two matrices and just **concatenating** onto each other. So matrix `A` is on the left and matrix `B` is on the right. And that's how I formed matrix `C`, by putting them together side by side.

I can also do `C=[A ; B]`. The semicolon notation means that I go to *put the next thing at the bottom.* It also puts the matrices `A` and `B` together, except that it now puts them on top of each other (`A` on top of `B`). `C` is now a 6×2 matrix. The semicolon usually means go to the next line: so, `C` is comprised initially by `A` , and then go to the bottom of it, and then put `B` in the bottom.

By the way, `[A B]` is the same as `[A, B]`, either of these gives you the same result, the comma between `A` and `B` can be ommited here.

So, hopefully, you now know how to construct matrices, and know about some commands that quickly slam matrices together to form bigger matrices with just a few lines of code. Octave is very convenient in terms of how quickly we can assemble complex matrices and move data around.

In the next video we'll start to talk about how to actually do complex computations on our data. With just a few commands, you can very quickly move data around in Octave. You load and save vectors and matrices, load and save data, put together matrices to create bigger matrices, index into or select specific elements or slices in the matrices.

I went through a lot of commands. So I think the best thing for you to do is to download the transcript of the video from the course site, look through it, type some commands into Octave yourself and start playing with these commands. And, obviously, there's no point at all to try to memorize all these commands.

Hopefully, from this video you got a sense of the sorts of things you can do. When you are trying to program

learning algorithms yourself, if you are trying to find a specific Octave command that you think you might have seen here, you should refer to the transcript of the session and look through that.

In the next video I will tell you how to do complex computations on data and actually start to implement learning algorithms.

3 Computing on Data

Now you know how to load and save data in Octave, put your data into matrices, etc. In this video I'll show you how to do computational operations on data, and later on we'll be using this sorts of operations to implement our learning algorithms.

Here's my Octave window. Let me just initialize some variables to use for examples, and set `A` to be a 3×2 matrix, set `B` to be a 3×2 matrix and set `C` to be a 2×2 matrix.

```
>> A = [1 2; 3 4; 5 6]
A =

   1   2
   3   4
   5   6

>> B = [11 12; 13 14; 15 16]
B =

   11   12
   13   14
   15   16

>> C = [1 1; 2 2]
C =

   1   1
   2   2
```

```
>> A*C
ans =

    5    5
   11   11
   17   17

>> A .* B
ans =

   11   24
   39   56
   75   96

>> A .^2
ans =

    1    4
    9   16
   25   36
```

Now I want to **multiply two of my matrices**, say $A \times C$. I just type `A*C`. So, it's a 3×2 matrix times a 2×2 matrix. This gives me this 3×2 matrix: `[5 5; 11 11; 17 17]`.

You can also do **element wise operations**. The expression `A .* B` takes each element of `A` and multiply it by the corresponding element of `B`. So, for example, the first element gives 1 times 11, which gives 11. The second element gives 2 times 12, which gives 24, and so on. We get `[11 24; 39 56; 75 96]`.

So, that was the **element wise multiplication** of two matrices. In general the dot `'.'` before an operation denotes element wise operation in Octave. The size of the two operands must conform in dot operations. When there is only an operand (e.g. in `A.^2`) the conformation issue is absent.

So, here's matrix `A` and let's dot square `A`, by executing '`A .^ 2`'. This gives the **element wise squaring** of `A`, so 1^2 is 1, 2^2 is 4, and so on.

```
>> v = [1; 2; 3]
v =

   1
   2
   3

>> 1 ./ v
ans =

   1.00000
   0.50000
   0.33333

>> 1 ./ A
ans =

   1.00000   0.50000
   0.33333   0.25000
   0.20000   0.16667
```

```
>> log(v)
ans =

   0.00000
   0.69315
   1.09861

>> exp(v)
ans =

    2.7183
    7.3891
   20.0855

>> abs(v)
ans =

   1
   2
   3

>> abs([-1; 2; -3])
ans =

   1
   2
   3
```

Let's set `v=[1; 2; 3]` as a column vector. You can also do `1 ./ v` to do the **element wise reciprocal** of `v` so this gives me 1/1, 1/2 and 1/3. This works too for matrices; so `1./A` is the element wise inverse of `A`. Once again the dot `'.'` gives us a clue that this is an element wise operation.

We can also do things like `log(v)`. This is an **element wise logarithm** of `v`. Also, `exp(v)` is the base e exponentiation of these elements of `v`: this is e, this is e^2, this is e^3... And I can also do `abs(v)` to take the **element**

wise absolute value of `v`. The example `v` was all positive, but in `abs([-1; 2; -3])` the element wise absolute value gives me back these non-negative values: 1, 2, 3.

```
>> -v
ans =

  -1
  -2
  -3

>> -1 * v  % -v
ans =

  -1
  -2
  -3

>> v + ones(length(v),1)
ans =

   2
   3
   4

>> length(v)
ans =  3
>> ones(3,1)
ans =

   1
   1
   1

>> v + 1
ans =

   2
   3
   4
```

```
>> A
A =

   1   2
   3   4
   5   6

>> A'
ans =

   1   3   5
   2   4   6

>> (A')'
ans =

   1   2
   3   4
   5   6
```

Also, `-v` gives me the **symmetric** of `v`. This is the same as `-1*v` but usually you just write `-v`.

Here's another neat trick. Let's say I want to take `v` and increment its elements by 1. Well, one way to do it is by constructing a 3×1 vector, all ones, and adding that to `v`. This increments `v` to `[2;3;4]`. The length of `v` is 3. So `ones(length(v),1)` that's a 3×1 vector of ones. And so this increments `v` by one.

A simpler way to do that is to just type `v+1`, right? It also means to **add 1 element wise** to all the elements of `v`.

Now, let's talk about more matrix and vector operations.

If you want `A` **transpose** you write `A'` (that reads `A` 'prime'). This is a standard quotation mark[2]. If I transpose that again as in `(A')'` then I should get back my matrix `A`.

```
>> a = [1 15 2 0.5]
a =

    1.00000   15.00000    2.00000    0.50000

>> val = max(a)
val =  15
>> [val, ind] = max(a)
val =  15
ind =  2
>> max(A)
ans =

   5   6

>> A
A =

   1   2
   3   4
   5   6
```

```
>> a < 3
ans =

   1   0   1   1

>> find(a < 3)
ans =

   1   3   4
```

Some more useful functions. Let's say `a=[1 15 2 0.5]`, a 1×4 matrix. Let's set `val=max(a)`. This returns the **maximum value** of `a`, which in this case is 15. I can do `[val, ind] = max(a)` and this returns `val`, the maximum value of `A` which is 15, as well as the index where the maximum is located. So `A(2)`, that is 15. So, `ind` is my index, it is 2 in this case.

Just as a warning: if you do `max(A)` where `A` is a matrix, this actually does the **column wise maximum**.

If I do `a<3`, this does the **element wise logical operation**. So, I get a boolean vector (of 0s and 1s, meaning 'false' and 'true', respectively), which is `[1 0 1 1]`. So this is just the element wise comparison of all four element in `a` with 3 and it returns true or false (or 0 or 1) depending on whether or not it's less than 3.

[2]In the text of this document we write very often commands between primes and with typewriter font, such as in `'eye(3)'`; do not confuse those primes with the prime denoting transposition of a matrix.

Now, if I do `find(a<3)`, this would tell me the **indices of the elements of `a` which satisfy the condition**, in this case the 1st, 3rd and 4th elements.

For my next example let me set `A=magic(3)`. Let's consult '`help magic`'. The `magic(N)` function returns matrices of dimensions $N \times N$ called **magic squares**. They have this mathematical property that all of their rows, columns and diagonals sum up to the same value, which is 15, for N=3. It's not actually useful for machine learning, as far as I know, but I'm just using `magic(3)` as a convenient way to generate a 3×3 matrix which is useful for teaching Octave or for doing demos.

```
>> A=magic(3)
A =

   8   1   6
   3   5   7
   4   9   2

>> [r,c] = find(A >= 7)
r =

   1
   3
   2

c =

   1
   2
   3

>> A(2,3)
ans =  7
```

The expression `[r,c] = find(A >= 7)` **finds all the elements** of `A` that are >=7 and so `r` and `c` sense the rows and columns of those elements. So, `A(1,1)>=7` as well as `A(3,2)` and `A(2,3)` – check this by yourself. I actually don't even memorize myself what these find function variations do, I type '`help find`' to look up what I want.

```
>> a
a =

    1.00000   15.00000    2.00000    0.50000

>> sum(a)
ans =  18.500
>> prod(a)
ans =  15
>> floor(a)
ans =

    1   15    2    0

>> ceil(a)
ans =

    1   15    2    1
```

Okay, just two more things. One is the `sum()` function. So here's my `a` and I type '`sum(a)`'. This adds up all the elements of a. And if I want to multiply them together, I type '`prod(a)`' and `prod()` returns the product of these four elements of `a`. Also, '`floor(a)`' *rounds down* these elements of `a`, so 0.5 gets rounded down to 0. And '`ceil(a)`', the ceiling of `a`, gets values *rounded up*, so 0.5 is rounded up to the nearest integer which is 1. We can also do these operations on matrices.

```
>> rand(3)
ans =

   0.5160515   0.4653368   0.8373977
   0.0072488   0.3880227   0.2283935
   0.6843117   0.1810154   0.9917831

>> max(rand(3),rand(3))
ans =

   0.72359   0.98333   0.13938
   0.95963   0.72763   0.42005
   0.24204   0.78635   0.91711

>> A
A =

   8   1   6
   3   5   7
   4   9   2

>> max(A,[],1)
ans =

   8   9   7

>> max(A,[],2)
ans =

   8
   7
   9
```

Let's see; let me type `'rand(3)'`. This generally sets a random 3×3 matrix. If I type `'max(rand(3),rand(3))'`, it takes the **element wise maximum** of 2 random 3×3 matrices, returning a 3×3 matrix. So, you'll notice all these numbers tend to be a bit on the large side, i.e. they are closer to 1 than to 0 (recall that with `rand()` we were drawing the entries in the two matrices from a uniform distribution spread between 0 and 1), because each of these values is actually the `max()` of two randomly generated element values in matrices.

Now, `A` is my magic 3×3 square. Let's say I type `'max(A,[],1)'`. What this takes is the column wise maximum. So, the maximum of the first column is 8, the max of the second column is 9, the max of the third column is 7. This 1 means to take the max along the first dimension of A. In contrast, if I type `'max(A,[],2)'` then this takes the per row maximum. So, the maximum for the first row is 8, max of second row is 7, max of the third row is 9 and so this allows you to take row max. You know, you can do it per row or per column.

Remember `max(A)` defaults to column wise maximum of the elements, so if you want to find the maximum element in the entire matrix `A`, you can type `'max(max(A))'`, which is 9. Or you can turn the matrix `A` into a column vector with `A(:)` and thus `'max(A(:))'` also takes its maximum element.

```
>> max(A)
ans =

   8   9   7

>> max(max(A))
ans =  9
>> max(A(:))
ans =  9
>> A = magic(9)
A =

   47   58   69   80    1   12   23   34   45
   57   68   79    9   11   22   33   44   46
   67   78    8   10   21   32   43   54   56
   77    7   18   20   31   42   53   55   66
    6   17   19   30   41   52   63   65   76
   16   27   29   40   51   62   64   75    5
   26   28   39   50   61   72   74    4   15
   36   38   49   60   71   73    3   14   25
   37   48   59   70   81    2   13   24   35

>>
```

```
>> sum(A,1)
ans =

   369   369   369   369   369   369   369   369   369

>> sum(A,2)
ans =

   369
   369
   369
   369
   369
   369
   369
   369
   369

>> A .* eye(9)
ans =

   47    0    0    0    0    0    0    0    0
    0   68    0    0    0    0    0    0    0
    0    0    8    0    0    0    0    0    0
    0    0    0   20    0    0    0    0    0
    0    0    0    0   41    0    0    0    0
    0    0    0    0    0   62    0    0    0
    0    0    0    0    0    0   74    0    0
    0    0    0    0    0    0    0   14    0
    0    0    0    0    0    0    0    0   35

>> sum(sum(A .* eye(9)))
ans =  369
```

Finally, let's set `'A=magic(9)'`. The magic square has this property that every column, every row and every diagonal sums the same thing. So here is the 9×9 magic square. Let me just do `'sum(A,1)'` so this does a per column sum. And so I'm going to take each column of A and add them up and this lets us verify that that property is indeed true for the 9 by 9 magic square. Every column adds up to 369. Now, let's do the row wise sum. So,

'sum(A,2)' sums up each row of A, and each row of A also sums up to 369.

Now let's sum the diagonal elements of A and make sure that they also sum up to the same value. I construct a 9×9 identity matrix, that's eye(9), and let me take A and multiply A and eye(9) element wise. So what 'A .* eye(9)' will do is take the element wise product of these 2 matrices, and so this should wipe out everything except for the diagonal entries. Now I'm going to sum, and 'sum(sum(A .* eye(9)))' gives me the sum of these diagonal elements, and indeed it is 369.

```
>> sum(sum(A .* flipud(eye(9))))
ans =  369
>> eye(9)
ans =

Diagonal Matrix

   1   0   0   0   0   0   0   0   0
   0   1   0   0   0   0   0   0   0
   0   0   1   0   0   0   0   0   0
   0   0   0   1   0   0   0   0   0
   0   0   0   0   1   0   0   0   0
   0   0   0   0   0   1   0   0   0
   0   0   0   0   0   0   1   0   0
   0   0   0   0   0   0   0   1   0
   0   0   0   0   0   0   0   0   1

>> flipud(eye(9))
ans =

Permutation Matrix

   0   0   0   0   0   0   0   0   1
   0   0   0   0   0   0   0   1   0
   0   0   0   0   0   0   1   0   0
   0   0   0   0   0   1   0   0   0
   0   0   0   0   1   0   0   0   0
   0   0   0   1   0   0   0   0   0
   0   0   1   0   0   0   0   0   0
   0   1   0   0   0   0   0   0   0
   1   0   0   0   0   0   0   0   0
```

```
>> A = magic(3)
A =

   8   1   6
   3   5   7
   4   9   2

>> pinv(A)
ans =

   0.147222  -0.144444   0.063889
  -0.061111   0.022222   0.105556
  -0.019444   0.188889  -0.102778

>> inv(A)
ans =

   0.147222  -0.144444   0.063889
  -0.061111   0.022222   0.105556
  -0.019444   0.188889  -0.102778

>> pinv(A) * A
ans =

  1.0000e+000  1.8041e-016  -2.8033e-015
  -6.0507e-015  1.0000e+000  6.3283e-015
  3.1641e-015  -4.0246e-016  1.0000e+000

>> A * pinv(A)
ans =

  1.0000e+000  -1.2434e-014  6.1340e-015
  2.6715e-016  1.0000e+000  6.9389e-017
  -6.1062e-015  1.2434e-014  1.0000e+000

>> _
```

You can sum up the other diagonal as well. The command for this is somewhat more cryptic. You don't really need to know this. I'm just showing, just in case any of you are curious. The command flipud() stands for flip up/down the matrix. If you do that to the identity matrix, then 'sum(sum(A.*flipud(eye(9))))' sums up the elements of the other diagonal, and those elements also sum up to 369. Let me show you, whereas eye(9) is the 9-size identity matrix, flipud(eye(9)) takes the identity matrix and flips it vertically, so you end up with ones on this opposite diagonal.

The command fliplr(matrix) stands for flip left/rigth the given matrix. So we could also have used 'sum(sum(A.*fliplr(eye(9))))' to sum up the opposite diagonal.

Just one last command, and then that's it for this video. Let's say A=magic(3) again. If you want to invert the matrix, you type pinv(A), where this command takes what is called a **pseudo inverse**. Think of it as basically being the same as the common **matrix inverse** of A, which is given by inv(A) (check that the equality is true in the example). The products pinv(A)*A and A*pinv(A) give the identity matrix of size 3, but for rounding errors in the order of 10^{-15} in the non-diagonal elements. Those products are indeed the identity matrix eye(3) with essentially ones in the principal diagonal and zeros in the off-diagonal elements, up to a numerical round-off.

So, that's it for how to do computational operations on the data kept in matrices.

After running a learning algorithm, often one of the most useful things is to be able to look at the results, to plot or visualize your results. In the next video I'll show you how, again, with one or two lines of code using Octave, you can quickly visualize your data or plot your data and use that picture to better understand what your learning algorithms are doing.

4 Plotting Data

When developing learning algorithms, very often a few simple plots can give you a better sense of what the algorithm is doing and just sanity check that everything is going okay and the algorithms are doing what is supposed to.

For example, in an earlier video, I talked about how plotting the cost function $J(\Theta)$ can help you make sure that gradient descent is converging. Often, plots of the data or of the learning algorithm outputs will also give you ideas for how to improve your learning algorithm.

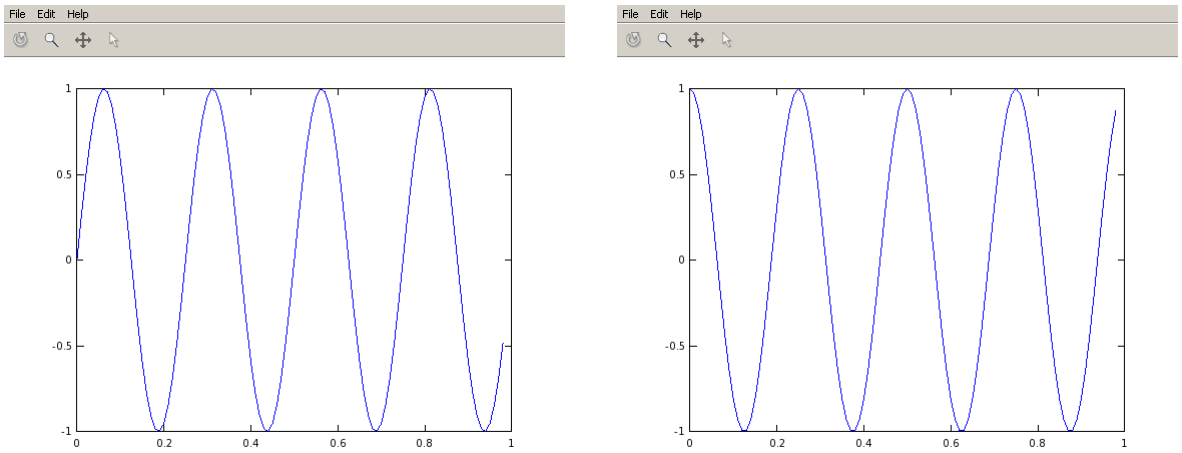


Figure 4: $y1 = \sin(2\pi 4t)$ at left and $y2 = \cos(2\pi 4t)$ at right.

Fortunately, Octave has very simple tools to generate lots of different plots. When I use learning algorithms I find that plotting the data and plotting the learning algorithm output are, often, an important part of how I get ideas for improving the algorithms. In this video, I'd like to show you some of these Octave tools for plotting and visualizing your data.

```
octave-3.6.1.exe:1> PS1('>> ');
>>
>>
>> t = [0:0.01:0.98];
>> y1 = sin(2*pi*4*t);
>> plot(t,y1);
>> y2 = cos(2*pi*4*t);
>> plot(t,y2);
>> plot(t,y1);
>> hold on;
>> plot(t,y2);
>> plot(t,y2,'r');
>> xlabel('Time')
>> ylabel('Value')
>> legend('sin', 'cos')
>> title('my plot')
>> print -dpng 'myPlot.png'
```

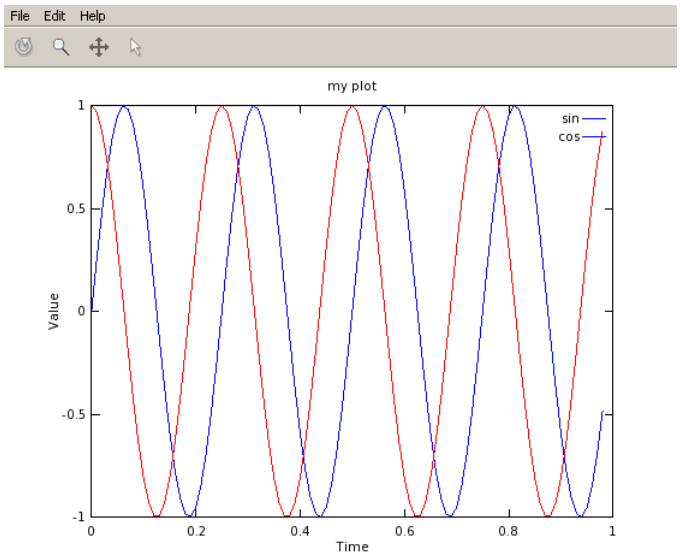
```
>> print -dpng 'myPlot.png'
>> close
>> figure(1); plot(t,y1);
>> figure(2); plot(t,y2);
>> subplot(1,2,1);  % Divides plot in a 1x2 grid, aceess first element
>> plot(t,y1);
>> subplot(1,2,2);
>> plot(t,y2);
>> axis([0.5 1 -1 1])
>> clf;
>>
```

Let's quickly generate some data for us to plot. So I'm going to set `t = [0:0.01:0.98]`, the time instants array. Let's set $y1 = \sin(2\pi 4t)$ and if I want to plot the sine function I just run `'plot(t,y1)'`. And up comes this plot (see Fig. 4) where the horizontal axis is the t variable and the vertical axis is `y1`, which is the sine.

Let's set $y2 = \cos(2\pi 4t)$. And if I execute `'plot(t,y2)'` Octave will replace my sine plot with this cosine function.

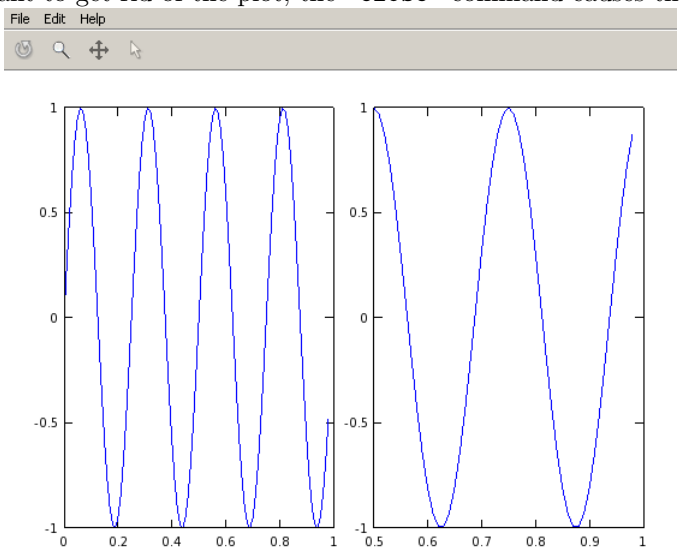
Now, what if I want to have both the sine and the cosine plots on top of each other? I'm going to type `'plot(t,y1)'`, so here's my sine function, and then I'm going to use the function `'hold on'` which indicates Octave to stack new figures on top of the old ones. Let me now `'plot(t,y2,'r')'` which plots the cosine function in a different color, where the char 'r' indicates it is red (or 'g' for green, 'b' for blue,...).

There are additional commands `'xlabel('Time')'` to label the X or horizontal axis, and `'ylabel('Value')'` to label the vertical axis, and I can also label my two curves in the plot with `'legend('sin','cos')'` and this puts a double legend in the top right corner of the figure window. And, finally, `'title('my plot')'` sets the title at the top of this figure.



Lastly, if you want to save this figure, you type `'print -dpng myPlot.png'`.

PNG is a **graphics file format and print** will let you save this as a .png file. Let me actually change directory to my 'Desktop', and then I will print that out. So this will take a while depending on how your Octave configuration is setup. Here's *myplot.png* which Octave has saved as the PNG file. Octave can save thousands other formats as well. You can type `'help plot'`, if you want to see the other file formats, rather than PNG, that you can save figures in. And, lastly, if you want to get rid of the plot, the `'close'` command causes the figure to go away.



Octave also lets you number figures. You type `'figure(1); plot(t,y1);'` and that starts up the first figure and plots `y1`. And then if you want a second figure, you specify a different figure number. So I do `'figure(2); plot(t,y2);'` and now, on my desktop, I actually have two figures.

Here's one other neat command that I often use, which is the `subplot()` command. So, we're going to use `'subplot(1,2,1)'`. It **sub-divides the plot** into a one-by-two grid (the first two parameters 1,2 indicate this), and it starts to access the first element (third parameter, 1). And if I do `'plot(t,y1)'` it now fills up this first element. And if I do `'subplot(1,2,2)'`, I'm going to start to access the second element and `'plot(t,y2)'` will throw in `y2` in the right hand side, or in the second element of the grid.

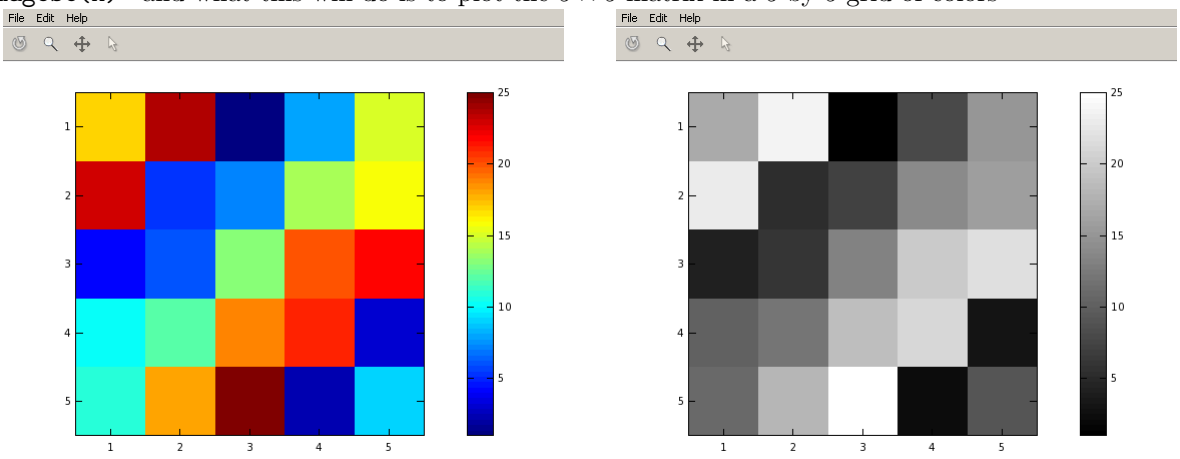
You can also change the axis scales; `'axis([0.5 1 -1 1])'` sets the x-range and y-range for the figure on the right (the figure currently accessed in the subplot), thus t is between 0.5 to 1, and the vertical axis values use the range from -1 to 1. You don't need to memorize all these commands, you can get the details from the usual Octave `'help'` command.

Finally, just a couple last commands.

`clf` clears the figure, and here's a unique trick...

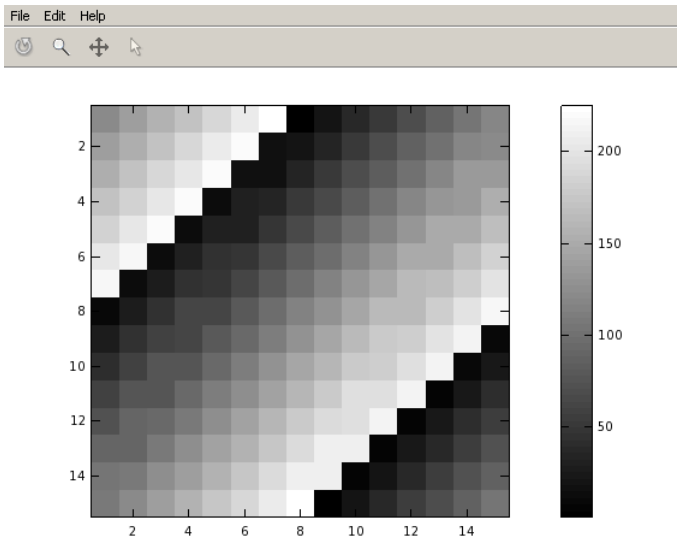
```
>>
>> A = magic(5)
A =

   17   24    1    8   15
   23    5    7   14   16
    4    6   13   20   22
   10   12   19   21    3
   11   18   25    2    9

>> imagesc(A)
>> imagesc(A), colorbar
>> imagesc(A), colorbar, colormap gray;
>> A(1,2)
ans =  24
>> A(4,5)
ans =  3
>> imagesc(magic(15)), colorbar, colormap gray;
>>
>> a=1, b=2, c=3
a =  1
b =  2
c =  3
>> a=1; b=2; c=3
c =  3
>> a=1; b=2; c=3;
>>
```

Let's set `A` to be equal to a 5 by 5 magic square. So, a neat trick that I sometimes use to visualize the matrix is to call `'imagesc(A)'` and what this will do is to plot the 5×5 matrix in a 5 by 5 grid of colors



Colored and gray maps of the 5×5 magic square matrix

where the different colors correspond to the different values in the matrix `A`. So, concretely, I can also run the `colorbar` command option to see the "legend" of the colors in the side of the picture.



Gray map of the 15×15 magic square matrix

Let me use a more sophisticated command `'imagesc(A), colorbar, colormap gray;'`. This is actually running three commands at a time. And what this does, is it sets a color map, a gray color map, and on the right of the picture it also puts in a color bar. The color bar shows what values the different shades of color correspond to. Concretely, the upper left element of `A` is 17, and so that corresponds to kind of a mint shade of gray. Whereas in contrast `A(1,2)` is 24. So that corresponds to a square which is nearly a shade of white. And a small value, say `A(4,5)` which is 3, corresponds to a much darker shade.

Here's another example, I can plot a larger matrix. Thus, `'magic(15)'` returns a 15×15 magic square and `'imagesc(magic(15)), colorbar, colormap gray;'` gives me a plot of what my 15×15 magic squares values looks like.

And finally to wrap up this video, what you've seen me do above with the `imagesc()` command, and so on, is use **comma chaining of function calls**. If I type `a=1, b=2, c=3` and hit RET, then this really is carrying out three commands, one after another, and it prints out all three results. If I use semicolons instead of a comma, `a=1; b=2; c=3;` it doesn't print out. So this thing here we call *comma chaining of commands*. And, it's just another convenient way in Octave to join multiple commands like '`imagesc(A), colorbar, colormap gray;`', to put multi-commands on the same line.

So, that's it. You now know how to plot figures in Octave, and in the next video I want to tell you about control statements like `if`, `while` and `for`, as well as how to define and use functions.

5 Control Statements: for, while, if

In this video, I'd like to tell you how to write control statements for your Octave programs, things like `'for'`, `'while'` and `'if'` statements and also how to define and use functions.

Here's my Octave window. Let me first show you how to use a **`'for'` loop**. I'm going to start by setting a column vector with ten zeros, `v=zeros(10,1)`.

```
octave-3.6.1.exe:1> PS1('>> ')
>>
>> v=zeros(10,1);
>> for i=1:10,
>>   v(i) = 2^i;
>>  end;
>> v
v =

      2
      4
      8
     16
     32
     64
    128
    256
    512
   1024

>> indices = 1:10;
>> indices
indices =

    1    2    3    4    5    6    7    8    9   10

>> for i=indices,
>>   disp(i);
>>  end;
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
>>
>> i = 1;
>> while i <= 5,
>>    v(i) = 100;
>>    i = i+1;
>>  end;
>> v
v =

   100
   100
   100
   100
   100
    64
   128
   256
   512
  1024

>> i = 1;
>> while true,
>>    v(i) = 999;
>>    i = i+1;
>>    if i == 6,
>>      break;
>>    end;
>>  end;
>> v
v =

   999
   999
   999
   999
   999
    64
   128
   256
   512
  1024

>>
```

Now, I write a `for` loop `'for i=1:10'`. And let's see, I'm going to set `v=2^i` for all indices, and finally `end;`. The white space does not matter, so I am putting the spaces just to make it look nicely indented. The result is that `v` elements get set to 2^1, 2^2, and so on. So this syntax `for i=1:10,` makes `i` loop through the values 1 through 10. And, by the way, you can also do this by setting your `indices=1:10;` and so the indices array goes from 1 to 10. You then can write `for i=indices`. And this is actually the same as `i=1:10`. You can call `display(i)` and this would do the same sequence of indices (although they are different commands, `disp(i)` gives the same as `display(i)` in this example).

So, that is a `for` loop. If you are familiar with `'break'` and `'continue'` statements, you can also use those inside loops in Octave, but first let me show you how a `'while'` loop works.

So, here's my vector `v` with powers of 2, `2^i`. Let's write the `while` **loop**. `i=1, while i<=5,` let's set `v(i)=100` and increment `i` by 1, `end;`.

So this says what? I start off with `i=1`, and then I'm going to set `v(i)=100` and increment `i` by 1 until `i>5`. And as a result of that, whereas previously `v` was this `2^i` vector, I've now overwritten the first 5 elements of my vector `v` with this value 100.

```
>>
>> v(1)
ans =  999
>> v(1) = 2;
>> if v(1)==1,
>>    disp('The value is one');
>>  elseif v(1) == 2,
>>    disp('The value is two');
>>  else
>>    disp('The value is not one or two.');
>>  end;
The value is two
>>
```

So that's the syntax for a `while` loop.

Let's do another example. `i=1; while true`, and here I want to show you how to use a `break` statement. Let's say `v(i)=999` and `i=i+1, if i==6, break;` and `end;`. And this is also our first use of an `if` **statement**, so I hope the logic of this makes sense.

In the beginning `i=1` and the while loop repeatedly sets `v(i)=999` and increments `i` by 1, and then when `i` gets up to 6 we do a `break` which breaks out the `while` loop and so, the effect is that the first 5 elements of the vector `v` are set to 999, but the remaining elements in `v` do not change.

So, this is the syntax for `'if'` statements, and for `'while'` statements, and notice the `end` word. We have two

'ends' here. One `end` ends the `if` statement, and the second `end` ends the `while` statement.

Now let me show you the more general syntax for how to use an **if-else statement**. So, let's see, `v(1)` now equals 999, let's type `'v(1)=2'` for this example. So, let me type `'if v(1)==1'` display *'The value is one'*. Here's how you write an `'else'` statement, or rather here's an `'elseif v(1)==2'`; in case that's true in our example, display *'The value is 2'*, `else` display *'The value is not one or two.'*. Okay, so that's a `if-elseif-else` statement. And, of course, here we've just set `v(1)=2`, so hopefully, yup, it displays that *'The value is two'*.

I don't think I talked about this earlier, but if you ever need to **exit Octave**, you can type the `'exit'` command, hit RET and that will cause Octave to quit. Or else the `'quit'` command and RET also works.

Finally, let's talk about **functions and how to define them and use them**.

I have predefined a file and saved it on my 'Desktop', a file called *squareThisNumber.m*. For your convenience, this is what is written in the file:

```
function y = squareThisNumber(x)

y = x^2;
```

So, this is how you define functions in Octave. You create a file with your function name and ending in *.m*, and when Octave finds this file, it knows that this is where it should look for the definition of the function *squareThisNumber.m*.

Let's open up this file. The Microsoft program Wordpad (not Notepad!) is suggested to open up this file. If you have a different text editor (as Scite, emacs, vi, Notepad++, Jedit, Jed, nano or many others) that's fine too, but Notepad sometimes messes up the spacing, so try not to use it.

So, here's how you define the function in Octave. This file has just three lines in it (the function is shown above). The first line says `function y = squareThisNumber(x)`, this tells Octave that I'm gonna return only one value, `y`, and moreover, it tells Octave that this function has one argument, `x`, and the way the function body is defined is `y = x^2;`.

```
>> cd ..
>> squareThisNumber(5)
error: `squareThisNumber' undefined near line 30 column 1
>> cd 'D:\Coursera\MachLearn'
>> squareThisNumber(5)
ans =  25
>>
>>  % Octave search path (advanced/optional)
>> addpath('D:\Coursera\MachLearn')
>> cd 'C:\'
>> squareThisNumber(5)
ans =  25
>> pwd
ans = C:\
>>
```

So, let's try to call this function and find the square of 5; and this actually isn't going to work, Octave says that 'squareThisNumber is undefined'. That's because Octave doesn't know where to find this file. So, as usual, let's `cd` to my[3] directory, `'cd C:\Users\ang\Desktop'`. That's where my desktop is. And if I now type `squareThisNumber(5)`, it returns the correct answer, 25.

As kind of an advanced feature, this is only for those of you that know what the term *search path* means. But so, if you want to modify the Octave search path... – you just think of this next part as advanced, or optional, material. Only for those who are already familiar with the concepts of *search paths* and permissions. So, you can use the term `'addpath(<directory>)'` to add `<directory>` (Andrew used here its own 'Desktop' directory) to the Octave search path so that even if you go to some other directory Octave still knows it has to look in `<directory>` for functions, so that even though I'm in a different directory now, it still knows where to find the `squareThisNumber()` function.

Okay? But if you're not familiar with the concept of search path, don't worry about it. Just make sure you use the `'cd'` command to go to the directory of your function before you run it, and that actually works just fine.

One concept that Octave has, that many other programming languages don't, is that it lets you define functions that return multiple values.

```
function [y1,y2] = squareAndCubeThisNumber(x)

y1 = x^2;
y2 = x^3;
```

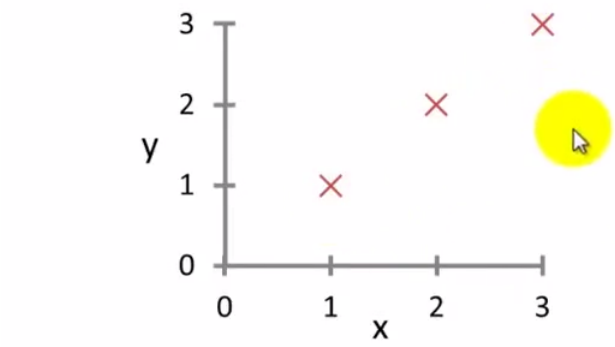
[3] In the writer's PC, it is not the Desktop but the directory *D:\Coursera\MachLearn* instead.

So here's an example of that. Define the function called `squareAndCubeThisNumber(x)`, and this function returns 2 values, `y1` and `y2`. Thus, `y1` is `x` squared, `y2` is `x` cubed and this really returns 2 numbers.

```
>>
>> [a,b] = squareAndCubeThisNumber(5);
>> a
a =  25
>> b
b =  125
>> X = [1 1; 1 2; 1 3]
X =

   1   1
   1   2
   1   3

>> y = [1; 2; 3]
y =

   1
   2
   3

>> theta = [0; 1]
theta =

   0
   1
```

So, some of you, depending on what programming language you use, e.g. if you're familiar with C, C++, often you think the function can return in just one value. But the syntax in Octave allows returning multiple values. Now back in the Octave window, if I type `'[a,b]=squareAndCubeThisNumber(5);'` then `a` is now equal to 25 and `b` is equal to the cube of 5, thus equal to 125. So, this is often convenient, to define a function that returns multiple values.

Finally, I'm going to show you just one more sophisticated example of a function. Let's say I have a data set that looks like this, with data points at (1,1), (2,2) and (3, 3).



Goal: Define a function to compute the cost function $J(\theta)$.

And what I'd like to do is to define an Octave function to compute the cost function, $J(\theta)$, for different values of θ. First let's put the data into Octave. So I set my design matrix to be `X=[1 1;1 2;1 3]`. So, this is my design matrix `X`, with the first column being 1s, and the second column being the x-values of my three training examples. And let me set `y=[1; 2; 3]`, which are the y-axis values. So, let's say also that `theta=[0; 1]`.

Here at my 'Desktop', I've predefined in a *.m* file the `costFunctionJ()` (see below).

```
function J = costFunctionJ(X, y, theta)

% X is the "design matrix" containing our training examples.
% y is the class labels

m = size(X,1);                 % number of training examples
predictions = X*theta;         % predictions of hypothesis on all m examples
sqrErrors = (predictions-y) .^ 2;       % squared errors

J = 1/(2*m) * sum(sqrErrors);
```

So, the first line is `function J = costFunctionJ(X, y, theta)`, then some comments specifying the inputs

and then very few steps set `m` to be the number training examples, thus the number of rows in `X`.

Then I compute the predictions, `predictions=X*theta;`.

Afterward I compute `sqrErrors` by taking the difference between `predictions` and the `y` values and then taking element wise squaring, and then finally computing the cost function, `J`. Octave knows that `J` is a value I want to return, because `J` appeared in the function definition.

```
>> j = costFunctionJ(X,y,theta);
>> j
j = 0
>> theta = [0;0];
>> j = costFunctionJ(X,y,theta)
j =  2.3333
>> (1^2 + 2^2 + 3^2)/ (2*3)
ans =  2.3333
```

I run it in Octave, that is I run `j = costFunctionJ(X,y,theta);` it computes `j=0` because my data set was, you know, (1,1), (2,2) and (3, 3), and then setting $\theta_0 = 0$ and $\theta_1 = 1$ gives me exactly the 45-degree line that fits my data set perfectly.

Whereas, in contrast, if I set `theta=[0;0]`, then this hypothesis is predicting 0s on everything, the `costFunctionJ()` then is 2.333 and that's actually equal to $(1^2 + 2^2 + 3^2)/(2 \times 3)$, the sum of squares divided by `2*m` (recall that m is the number of training examples), 2*3 in this case, and (1^2 + 2^2 + 3^2)/(2*3) is indeed equal to 2.333.

And so, that sanity checks that we're computing the correct cost function, in the couple of examples we tried out on our simple training example. And so that sanity check tracks that `costFunctionJ()`, as defined here, seems to be correct, at least on our simple training set.

So, now you know how to write control statements like `for` loops, `while` loops and `if` statements in Octave, as well as how to define and use functions. In the next video, I'm going to tell you about vectorization, which is an idea for how to make your Octave programs run faster. And, finally, in the final Octave tutorial video, I'll just very quickly step you through the logistics of working on and submitting problem sets for this class and how to use our submission system.

6 Vectorization

In this video, I'd like to tell you about the idea of *vectorization*. Whether you're using Octave or a similar language like MATLAB, or whether you're using Python and NumPy, or Java, or C++, all of these languages have either built into them, or readily and easily accessible, different numerical linear algebra libraries. They're usually very well written, highly optimized, often developed by people really specialized in numerical computing.

And when you're implementing machine learning algorithms, if you're able to take advantage of these linear algebra libraries or these numerical linear algebra libraries, and mix the routine calls to them, rather than sort of develop yourself things that these libraries could be doing, you get into big time and headache savings.

If you do that, then often you get: first, more efficiency, so things just run more quickly and take better advantage of any parallel hardware your computer may have; and second, it also means that you end up with less code that you need to write. So you have a simpler implementation that is, therefore, more likely to be bug free.

And as a concrete example, rather than writing code yourself to multiply matrices, if you let Octave do it by typing `A*B` that will use a very efficient routine to multiply the two matrices. And there's a bunch of examples like these where you use appropriate vectorized implementations, such that you get much simpler and much more efficient code.

Let's look at some examples.

Here's our usual hypothesis of linear regression, and if you want to compute $h_\theta(x)$,

$$h_\theta(x) \quad = \quad \sum_{j=0}^{n} \theta_j x_j$$

notice that there is a sum on the right. And so one thing you could do is compute the sum from $j = 0$ to $j = n$.

Another way to think of this is

$$h_\theta(x) = \theta^T x$$

and you can think of this as computing the inner product $\theta^T x$ between two vectors, where θ is

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}$$

because you have $n = 2$ features in your model. If you think of x as this vector

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

these two views, $h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$ and $h_\theta(x) = \theta^T x$, give you two valid but different implementations for the calculations.

Here's what I mean. Here's an **unvectorized implementation** for how to compute $h_\theta(x)$, and by unvectorized I mean without vectorization.

```
prediction = 0.0;
for j = 1:n+1,
        prediction = prediction + theta(j) * x(j);
end;
```

We might first initialize `prediction=0.0`. The prediction is going to be $h_\theta(x)$ and then I'm going to have a `for` loop `'for j=1:n+1,'` where `prediction` gets incremented by `theta(j)*x(j)`.

By the way, I should mention that the vectors θ and x in the mathematical equations above had the index 0. So, I had θ_0, θ_1 and θ_2 there, but because MATLAB and Octave vectors start at index 1 (the index 0 doesn't exist), then in the program we end up representing them as `theta(1)`, `theta(2)` and `theta(3)`, and the same indices 1, 2 and 3 to `x(j)`. This is also why I have a for loop where `j=1:n+1,` rather than `j=0:n,` right?

But the above code is an unvectorized implementation, where a for loop sums up the n elements of the sum.

In contrast, here's a **vectorized implementation**

```
prediction = theta' * x;
```

where you think of `x` and `theta` as vectors, and you just set prediction equals $\theta^T x$. Instead of writing all the lines of code with the `for` loop, you instead have one line of code and this line of vectorized code uses Octaves highly optimized numerical linear algebra routines to compute this inner product $\theta^T x$. And not only is the vectorized implementation simpler, it will also run more efficiently.

So, that was an example in Octave, but the issue of vectorization applies to other programming languages as well.

Let's look at an example in C++. Here's what an unvectorized implementation might look like.

```
double prediction = 0.0;
for (int j = 0; j <= n; j++)
        prediction += theta[j] * x[j];
```

We again initialize `prediction=0.0` and then we now have a full loop `for j=0` up to `n`, wrapping `'prediction+= theta[j] * x[j];'`. Again, you have a `for` loop that you write yourself (in C and C++ the indices of vectors start at 0).

In contrast, using a good numerical linear algebra library in C++, you can instead write code that might look like this

```
double prediction
        =theta.transpose() * x;
```

So, depending on the details of your numerical linear algebra library, you might be able to have a C++ object which is the vector `theta`, and a C++ object which is the vector `x`, and you just take `'theta.transpose()*x;'`, where '*' is an overloaded 'times' operator so that you can just multiply these two vectors in C++.

And depending on the details of your numerical and linear algebra library, you might end up using a slightly different syntax. But by relying on a library to do the inner product, you can get a much simpler and much more efficient piece of code.

Let's now look at a more sophisticated example. Just to remind you, here's our **update rule for the gradient descent for linear regression**

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \qquad \text{(for all } j\text{=0,1,2,...)}$$

and so, we update θ_j using this rule for all values of $j = 0, 1, 2, ...$, and so on. And if I just write out these equations for θ_0, θ_1 and θ_2

$$\begin{aligned}
\theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_0^{(i)} \\
\theta_1 &:= \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_1^{(i)} \\
\theta_2 &:= \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_2^{(i)}
\end{aligned} \tag{1}$$

assuming two features, so $n = 2$, then these are the updates we perform to θ_0, θ_1 and θ_2, where you might remember that these should be **simultaneous updates**.

So let's see if we can come up with a vectorized implementation of this. You can imagine that a way to implement these three lines of math is to have a `for` loop that for $j = 0, 1, 2$ updates θ_j. But, instead, let's come up with a simpler vectorized implementation which compress these three lines of math, or a for loop that does them one at a time, into one single line of vectorized code.

Let's see how to compress these 3 math steps into one line of vectorized code. Here's the idea. I'm going to think of θ as a vector and I'm going to update

$$\theta := \theta - \alpha\delta \tag{2}$$

where, according to (1), the vector δ is going to be

$$\delta = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x^{(i)} \tag{3}$$

So, let me explain what's going on here. I'm going to treat $\theta \in \Re^{n+1}$ as a $n+1$ dimensional vector. θ gets updated as this: α is a real number and δ is a vector $\in \Re^{n+1}$. So, $\theta - \alpha\delta$ is a vector subtraction, because $\alpha\delta$ is a vector and so θ gets $\alpha\delta$ subtracted from it.

The vector δ will be a $n+1$ dimensional vector, as said

$$\delta = \begin{bmatrix} \delta_0 \\ \delta_1 \\ \delta_2 \end{bmatrix}$$

and, according to (1), δ_0 is going to be equal to

$$\delta_0 = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_0^{(i)}$$

So, let's just make sure that we're on the same page about how δ really is computed. It corresponds to the sum in (3). There, m is a real number, $(h_\theta(x^{(i)}) - y^{(i)})$ is also a real number an $x^{(i)}$ is a vector in $\Re^{n+1}$, right? That would be

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \end{bmatrix}$$

And what is the summation in (3)? Well, the summation develops to

$$\sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x^{(i)} = \left(h_\theta(x^{(1)}) - y^{(1)} \right) x^{(1)} + \cdots + \left(h_\theta(x^{(m)}) - y^{(m)} \right) x^{(m)} \tag{4}$$

So, as i ranges from 1 through m, you get these different terms $\left(h_\theta(x^{(i)}) - y^{(i)} \right) x^{(i)}$, and you're summing up these terms. And the meaning of each of these terms in the sum is a lot like that found in the quiz exercise

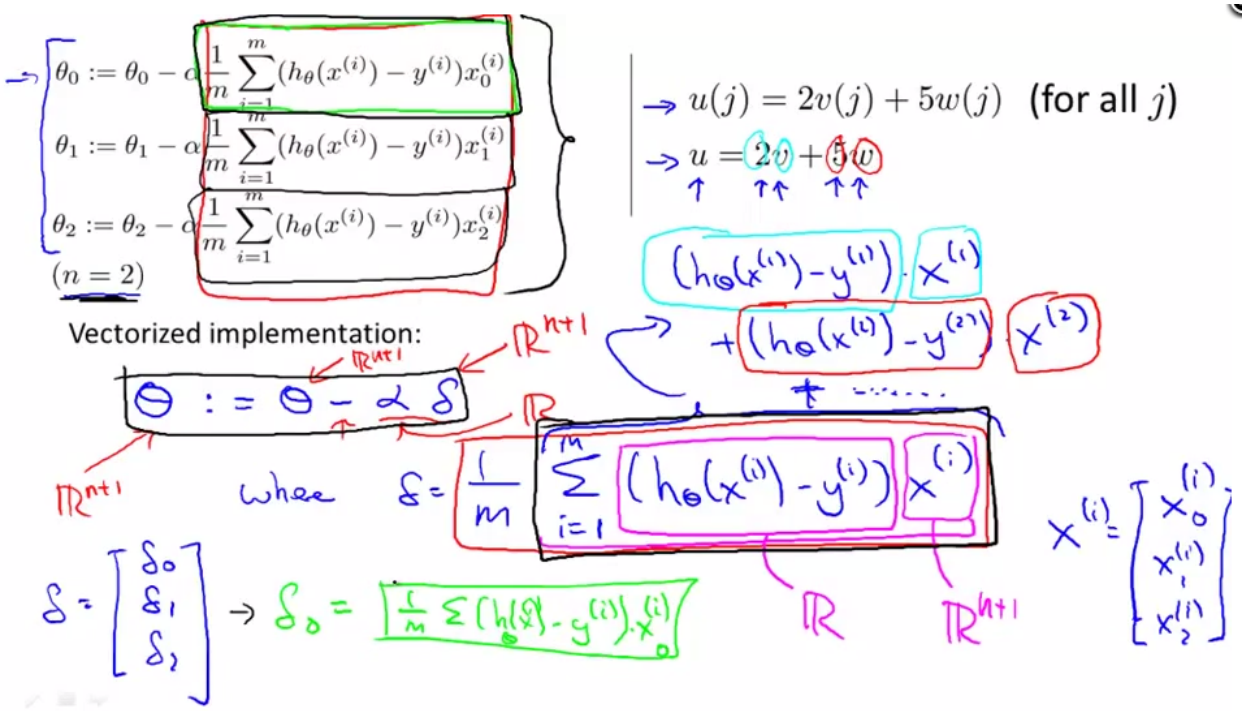


Figure 5: Final aspect of the slide used in the vectorization video.

$$\begin{aligned} u(j) &= 2v(j) + 5w(j) \qquad \text{(for all } j\text{)} \\ u &= 2v + 5w \end{aligned}$$

where that in order to vectorize the upper line code, we will instead set $u = 2v + 5w$ (lower line) and this works with whole vectors, u, v and w, all with the same dimensions, instead of doing a cycle over the vector index, j.

So, it was just an example of how to add different vectors, and the summation (4) is the same thing: the $(h_\theta(x^{(i)}) - y^{(i)})$ are real numbers (like 2 and 5 in the example), and the $x^{(i)}$ are vectors, like v and w. Thus, that whole quantity, δ, is just some vector and, concretely, the 3 elements of δ correspond to the 3 terms $(1/m)\sum\cdots$ which are shown in the 3 equations in (1). Thus, the vectorized θ in (2) ends up having exactly the same simultaneous update as the update rules implicit in (1).

So, I know that a lot happened on the slides, but again feel free to pause the video and I encourage you to step through the slide to make sure you understand why is it that this update $\theta := \theta - \alpha\delta$ works, right? We grab separate computations and compress them into one step with the vector δ, and we can come up with a vectorized implementation of this step of linear regression.

If you're able to figure out why these two alternative steps are equivalent then, hopefully, that would give you a better understanding of vectorization as well and, finally, vectorization increases the efficiency if you're implementing linear regression using more than one or two features.

Sometimes we use linear regression with tens or hundreds of thousands of features; if you use vectorized linear regression, usually that will run much faster than if you had, say, your old `for` loop.

And when you later vectorize algorithms that we'll see in this class, that'l be a good trick whether in Octave or some other language, as C++ or Java, for getting your code to run more efficiently.

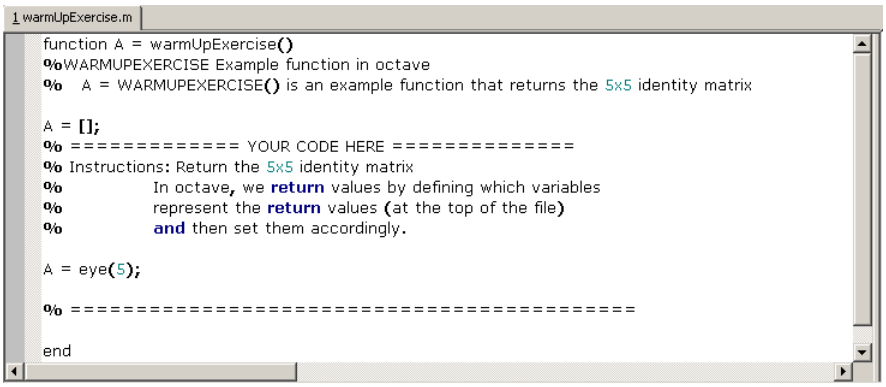


Figure 6: File *warmUpExercise.m* which shows the solution of the first programming question in ML.

7 Working on and Submitting Programming Exercises

In this video, I'll talk about how the homework works and quickly step you through the logistics of the homework submission system. The submission system let's you see immediatly if your answers to the Machine Learning exercises are correct.

Now, let's 'cd' to the directory where the files of the first programming homework are which (in our case) is '*H:\Coursera\MachLearn\ex1*'. The files were packed originally in the archive *ex1.zip*, which is given to the students, and this archive was unpacked to that directory.

I should open the *pdf* file which explains the homework, but for the sake of brevity, I'll skip that step. I just want to familiarize you with the submission system.

Let's open the file *warmUpExercise.m*. In this question you are asked to return a 5×5 identity matrix. It is obvious that you shall use the `eye()` function. So, we modify the *.m* file using any text editor – but Windows Notepad (Scite[4], is used in this case), writing our answer in the middle of it, where it is supposed to introduce the code; our answer is the command `A=eye(5);`.

We save the edited file and go back to the Octave window. We are supposed to be already in the directory which has the files for the programming exercises – this can be checked with `'ls'`. If not, we go to that directory by using `'cd <directory>'`.

```
octave-3.6.1.exe:1> PS1('>> ');
>> cd 'H:\Coursera\MachLearn\ex1'
>> ls
 Volume in drive H is KINGSTON
 Volume Serial Number is 3CE0-6527

 Directory of H:\Coursera\MachLearn\ex1

[.]                    featureNormalize.m      warmUpExercise.m
[..]                   gradientDescent.m       ex1data1.txt
computeCost.m          gradientDescentMulti.m  ex1data2.txt
computeCostMulti.m     normalEqn.m
ex1.m                  plotData.m
ex1_multi.m            submit.m
              13 File(s)         31.929 bytes
               2 Dir(s)  14.027.071.488 bytes free
>> warmUpExercise()
ans =

Diagonal Matrix

   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
```

To check if our exercise was done correctly, we can call the file from inside the Octave console. This is done with the command `'warmUpExercise()'`.

In fact we obtain the correct answer: the 5-size identity matrix.

[4]Can be downloaded for free at `http://www.scintilla.org/SciTE.html`

```
>> submit()
==
== [ml-class] Submitting Solutions | Programming Exercise 1
==
== Select which part(s) to submit:
==   1) Warm up exercise  [ warmUpExercise.m ]
==   2) Computing Cost (for one variable) [ computeCost.m ]
==   3) Gradient Descent (for one variable) [ gradientDescent.m ]
==   4) Feature Normalization [ featureNormalize.m ]
==   5) Computing Cost (for multiple variables) [ computeCostMulti.m ]
==   6) Gradient Descent (for multiple variables) [ gradientDescentMulti.m ]
==   7) Normal Equations [ normalEqn.m ]
==   8) All of the above
==
Enter your choice [1-8]: 1
Login (Email address): xxxxxxxx @gmail.com
Password: V xxxxxxxx

== Connecting to ml-class ...
== [ml-class] Submitted Assignment 1 - Part 1 - Warm up exercise
==
>>
```

Now it is time to submit the code. We call the function `'submit()'` in the console, which executes the m-file called *submit.m* which is in our directory. This displays our options, numbered from 1 upto 8[5]. We are ready to submit Part 1, the *warmUpExercise.m* file, so we enter 1.

Afterwards, we are asked for our email (the one which was used to enroll in the course, and which is shown in yours Coursera programming submission page) and so we write it (or copy it, directly from the html page.)

Then we are asked the password, and a 'generated password', also shown in the programming submission page, shall be introduced.

Afterwards our computer connects to the ML class submission page, and returns

`== [ml-class] Submitted Assignment 1 - Part 1 - Warm up exercise`

In the version of submission used in Prof. Ng's video, it is also shown the results of the assignment, but in the current version of submission (April 2012, Octave 3.6.1) that is not happening when the answer is correct, *it only warns you if you are submitting a wrong answer*. However, if after submitting the correct answer you refresh the submission page in the browser, you will see the points awarded to you in that exercise (in this example, it should be *10.00/10*).

So, in any case you have prompt feedback about the correctness of your submission.

You can use the automatic-generated password shown in the Web submission page, and even you can re-generate it, to submit the homework. However, you can also resort to the regular password you use for logging in into the ML course. The availability of an automatic password, visible in the submission page, has the goal of avoiding the need of you to be forced to write your regular password – depending on the Operating System you use, it could be visible in the Octave window, – which then could be seen and misused by other people.

So, that's how you submit the homeworks. Good luck, and when you get around to homeworks I hope you get all of them right. And finally, in the next and final Octave tutorial video, I want to tell you about vectorization, which is a way to get your Octave code to run much more efficiently (in this transcription, vectorization has been already presented in section 6).

[5] In the original video, which was released in the previous run of the Machine Learning course, there were 9 options.