MATLAB Tutorial

Everything starts in the MATLAB command line.

Contents

Declaring a Variable

The first thing we'll do is make a variable. It's pretty easy actually; just use the equals sign.

```
x=4
```

```
x =

     4
```

Note how it outputs to the window. That can be annoying. See?

```
x = 5
y = 6
z = 7
```

```
x =

     5


y =

     6
```

```
z =

     7
```

How do we stop it? Simple: put a semicolon!

```
x = 5;
y = 6;
z = 7;
```

MATLAB's real power is in dealing with vectors and matrices. To utilize that power, first we must make a vector and a matrix. Let's make a vector first.

```
x = 1:11

x =

     1     2     3     4     5     6     7     8     9    10    11
```

Note that we have made a vector that goes from 1 to 6 with stepsize 1. Pretty cool, eh? What if we wanted a different stepsize, like 2?

```
x = 1:2:11

x =

     1     3     5     7     9    11
```

That's how. If we want to make an arbitrary vector, we use brackets:

```
x = [1 4 3 2 5 6]

x =

     1     4     3     2     5     6
```

That gives us a row vector, but what about a column vector? Well, just transpose the row vector. To transpose a MATLAB vector, use the single quote.

```
x = x'
```

```
x =

     1
     4
     3
     2
     5
     6
```

Matrices are vectors put together. We go back to the brackets ...

```
x = [1 2 3 4; 5 6 7 8]
```

```
x =

     1     2     3     4
     5     6     7     8
```

Note how I used the semicolon in the brackets to put the rows on top of each other. Suppose I wanted to put two columns together instead; then, I use a comma or a space

```
y = [1 2 3 4]'
z = [5 6 7 8]'
x = [y, z]
x = [y z]
```

```
y =

     1
     2
     3
     4


z =

     5
     6
     7
     8
```

```
x =

     1     5
     2     6
     3     7
     4     8


x =

     1     5
     2     6
     3     7
     4     8
```

All this "concatenation" can be boiled down to

- Put elements together by column with spaces or commas.
- Put elements together by row with semicolons.

Think of everything in MATLAB as a matrix. So numbers are 1x1 matrices, rows are 1xn matrices and columns are nx1 matrices. That will make the manipulation a lot easier. Addition and subtraction are easy ...

```
x = y - z
```

```
x =

    -4
    -4
    -4
    -4
```

... as long as the elements are the same. One issue is multiplication and division. In MATLAB, the multiplication operator is MATRIX multiplication and not element-wise multiplication. What does that mean? It means

```
x = [1 2; 3 4];
```

```
y = [2 2; 2 2];
z = x*y
```

```
z =

     6     6
    14    14
```

is not just element-wise; it's real matrix multiplication. But what if we just want to multiply each element together? We can do that! For multiplication, division and exponentiation, we put a period before it to indicate element-wise application of the operation. See?

```
z = x.*y
z = x.^y
```

```
z =

     2     4
     6     8


z =

     1     4
     9    16
```

We also have some special vectors that can be made via functions:

```
x = zeros(3)
y = ones(3,1)
z = rand(4,2)
```

```
x =

     0     0     0
     0     0     0
     0     0     0

y =
```

```
     1
     1
     1


z =

    0.8147    0.6324
    0.9058    0.0975
    0.1270    0.2785
    0.9134    0.5469
```

The last one is very important, as it allows us to make a matrix of values that are random (from a uniform distribution to be exact). To draw from a Gaussian distribution, use

```
z = randn(4,2)
```

```
z =

    3.5784    0.7254
    2.7694   -0.0631
   -1.3499    0.7147
    3.0349   -0.2050
```

Functions

Also, the last example shows us that calling functions are easy. Given a function name you can figure out how the function works by the help command

```
help interp1
```

```
interp1 - 1-D data interpolation (table lookup)

    This MATLAB function returns interpolated values of a 1-D function at
specific
    query points using linear interpolation.

    vq = interp1(x,v,xq)
```

```
vq = interp1(x,v,xq,method)
vq = interp1(x,v,xq,method,extrapolation)
vq = interp1(v,xq)
vq = interp1(v,xq,method)
vq = interp1(v,xq,method,extrapolation)
pp = interp1(x,v,method,'pp')

See also griddedInterpolant, interp2, interp3, intern
doc interp1
```

This tells us how the interp1 function works. Pretty cool function, eh? If you want to see all the description and examples of each syntax, one click on doc interp1.

If you want to see all the variables, use

```
who
```

```
Your variables are:

ans  fs   m    t    x    y    z
```

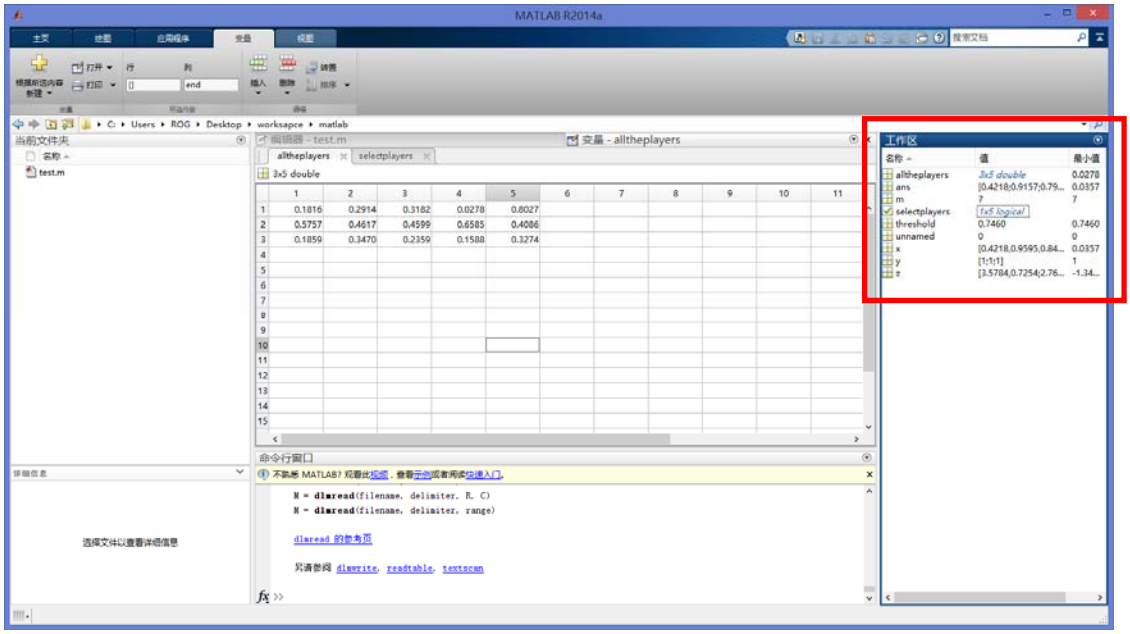
and to see the details (how big the variables are, etc.) use

```
whos
```

```
  Name      Size            Bytes  Class     Attributes

  ans       1x27               54  char
  fs        1x1                 8  double
  m         1x1                 8  double
  t         1x21              168  double
  x         3x3                72  double
  y         3x1                24  double
  z         4x2                64  double
```

which shows the current variables in your "workspace". We could also click into your "workspace" to see the details



and

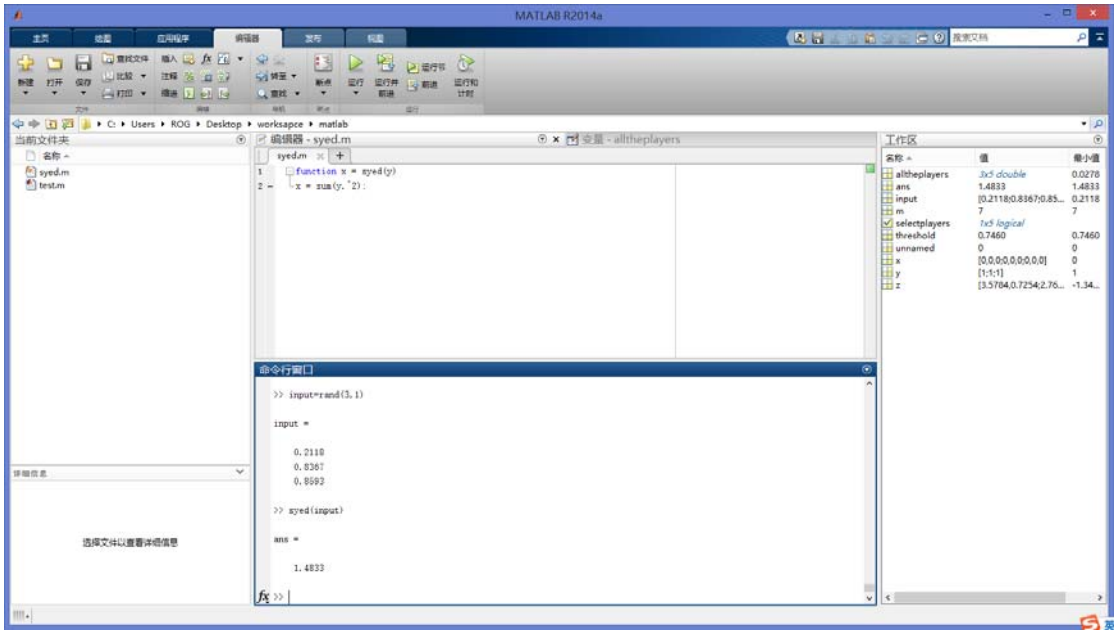
```
clear x
```

which clears the specified variables. To clear all, you just

```
clear all
```

Writing Functions and Scripts

At some point you guys will be making your own functions. To do that, you make a new M-file and name it by the function name. Let's say our function is called syed; then the new file will be syed.m ... let's let it do something simple: function x = syed(y) x = sum(y.^2) end



Doing a help will tell you that "sum" sums up all the elements of a vector, and for a matrix it sums up each row. Since the .^2 squares the elements, we get a sum-of-squares! Note first the keyword function. If that keyword is not there, then we have a script. What's the difference? A script is just a list of commands that are run. A function has inputs and outputs. So if you want a script (generally the "main" file for your work will be a script) then make sure the first word is not function. When you make a script, it is also an M-file. For example, this tutorial was created in an M-file called tutorial.m and I called it in the command line just by typing the M-file name "tutorial" ... make sure the M-file is in your path. You can see your path at the top of MATLAB or by the command

```
pwd

ans =

C:\Users\ROG\Desktop\worksapce\matlab
```

Your current directory is the directory you're working in, so generally you start

by changing your directory using the command cd

```
cd C:\Users\ROG\Desktop\worksapce\matlab
```

and to see all the files in your directory, use

```
ls
```

```
. .. tutorial.m
```

Loops and Vectorization

The next step to complicate things is loops. To make a for loop use the for command:

```
for m = 1:10
x(m) = m^2;
end
```

Note the syntax: after the for loop you give the iterating variable (don't use i or j because in MATLAB, they are the square root of -1). Then you set it equal to a vector. Note I said vector and not just 1:10. See?

```
for m = [2 4 3 7]
   m^2
end
```

```
ans =

     4


ans =

    16


ans =

     9


ans =
```

```
     49
```

So for loops are cool right? WRONG. At all costs, do not use for loops. They are super-slow and basically only for the most necessary situations. Take the previous for loop. Instead of doing that, we could just do

```
x = (1:10).^2
```

```
x =

     1     4     9    16    25    36    49    64    81   100
```

and get what we want much faster. Always try to vectorize your code. One way to help you out is this tool, which takes any matrix and turns it into a vector:

```
x = rand(3,3)
x(:)
```

```
x =

    0.4218    0.9595    0.8491
    0.9157    0.6557    0.9340
    0.7922    0.0357    0.6787


ans =

    0.4218
    0.9157
    0.7922
    0.9595
    0.6557
    0.0357
    0.8491
    0.9340
    0.6787
```

If we want to compare every # of a vector with a variable, just do

```
alltheplayers = rand(3,5);
threshold = rand(1,1);
selectplayers = (alltheplayers(1,:)>threshold);
```

```
alltheplayers =

    0.1816    0.2914    0.3182    0.0278    0.8027
    0.5757    0.4617    0.4599    0.6585    0.4086
    0.1859    0.3470    0.2359    0.1588    0.3274


threshold =

    0.7460


selectplayers =

     0     0     0     0     1
```

we compare the first row of matrix alltheplayers with a single variable threshold,

and express the result in logical.

If you have to use a for loop, try to initialize the variable that's being updated.

So in our for loop, we should do

```
x = zeros(10,1);
for m = 1:10
   x(m) = m.^2;
end
```

to prevent major time issues due to having to constantly add size to the vector.

For conditioning (that is, if-else ladders) use this construct

```
m = 7;
if m == 5
  disp('Hi.');
elseif m == 7
  disp('What''s up?');
```

```
else
  disp('Hello.');
end
```

```
What's up?
```

Note that the "end" ends the if loop. disp displays a string; note how you use consecutive single quotes (not double quotes) to get a single quote.

Plotting Results

So let's make some time series:

```
fs = 20;
t = 0:1/fs:1; % Time segment of one second with sampling rate 20
x = t.^2;
```

Whoa, wait. What was that percent sign and green thing? You probably guessed that it was a comment. The percent sign is MATLAB's version of the // for C. Use it often, and use it well. Please. For our sanity. And yours. We want comments that are good and frequent. They help; I promise. Ok, back to plotting. The plot command does the trick

```
figure, plot(t,x);
```

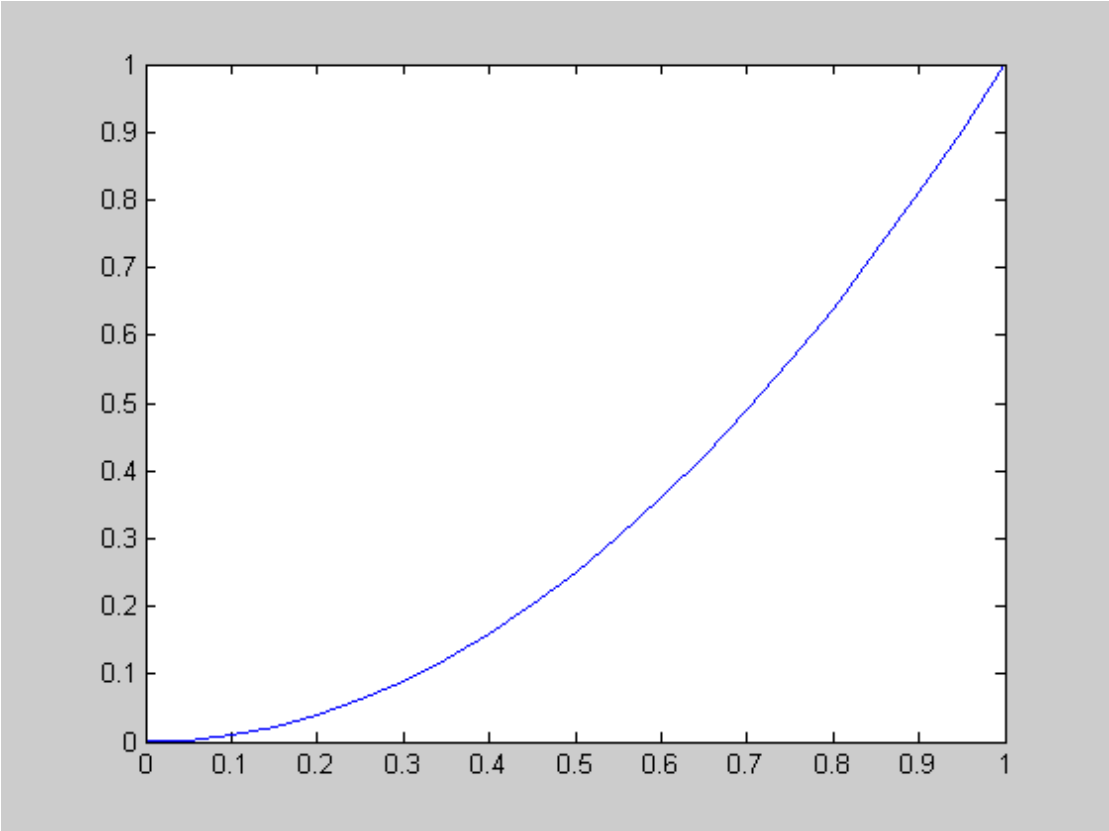
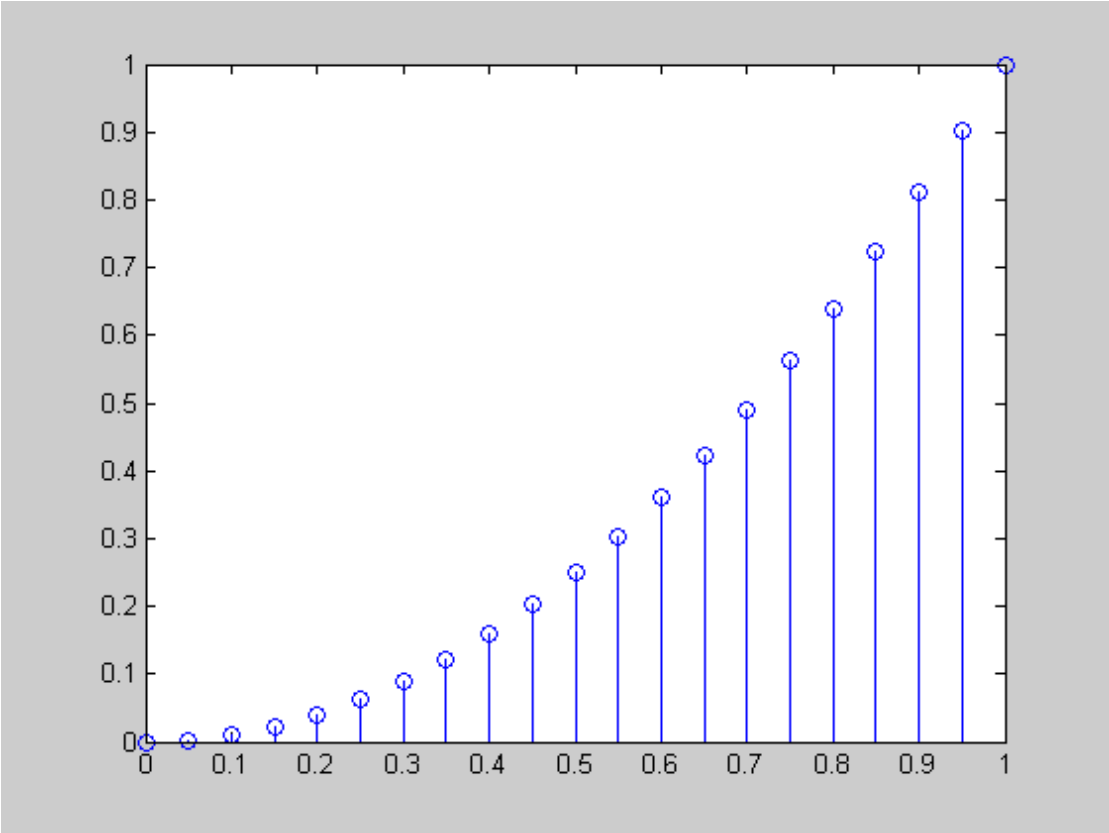


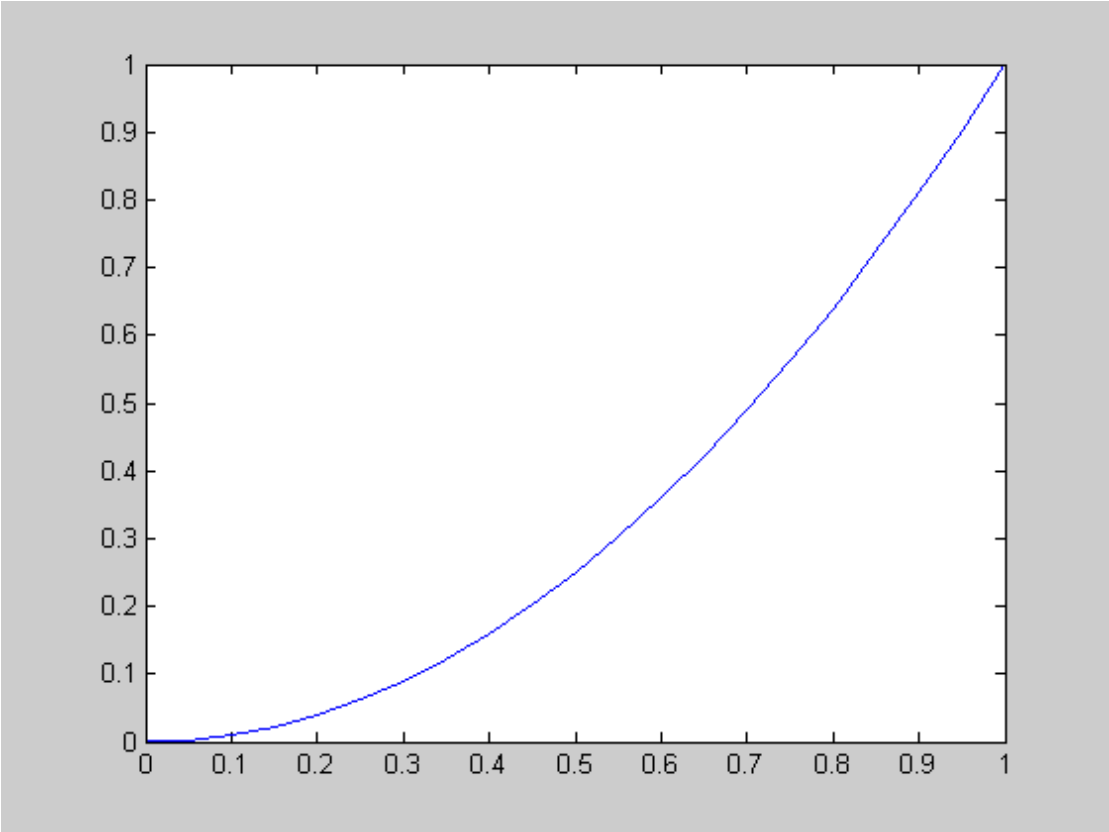
figure creates a new figure upon which we plot the data. Note that the xaxis goes first, followed by the y-axis. This plot is pretty lame. If x were filter coefficents, we'd like for you to use plot's cousin, stem

```
stem(t,x);
```



But for signals, let's go back to plot.

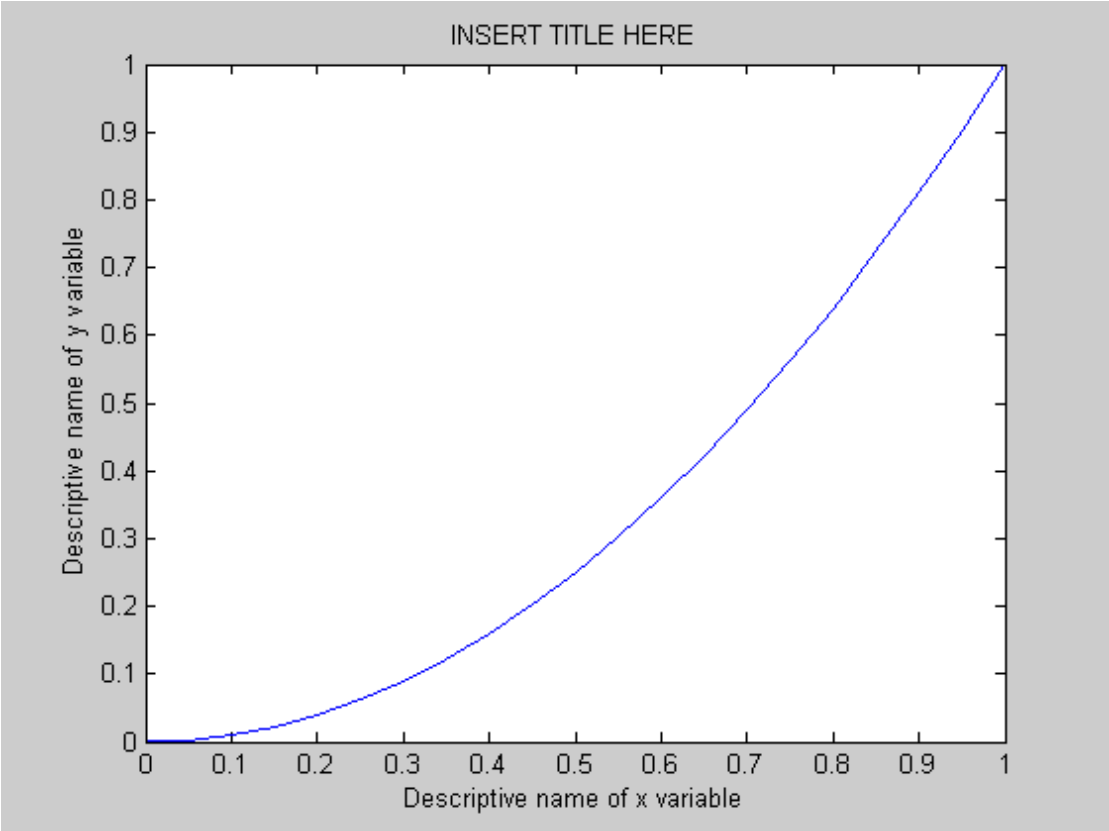
```
plot(t,x);
```



Note my habit of using semicolons? Very useful. Ok how do we edit the plot?

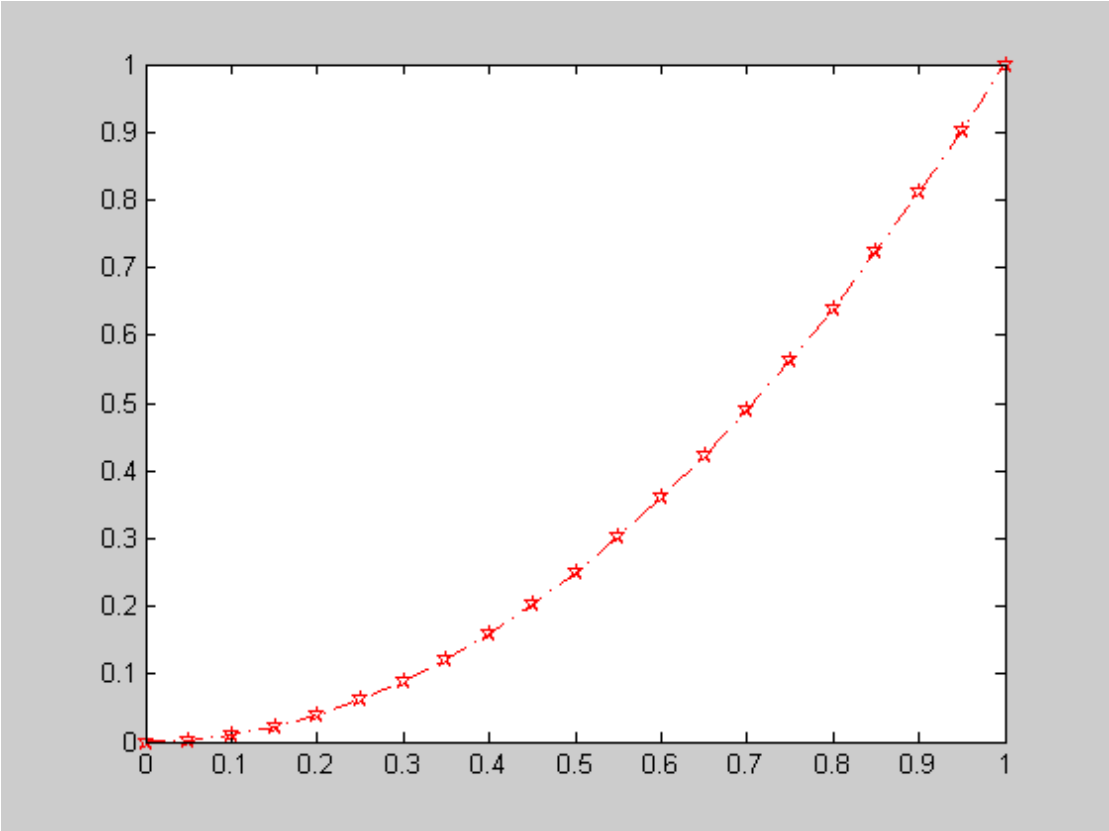
Well, I'm a command-line guy, so I use it a lot. Let's start with annotation:

```
title('INSERT TITLE HERE');
xlabel('Descriptive name of x variable');
ylabel('Descriptive name of y variable');
```



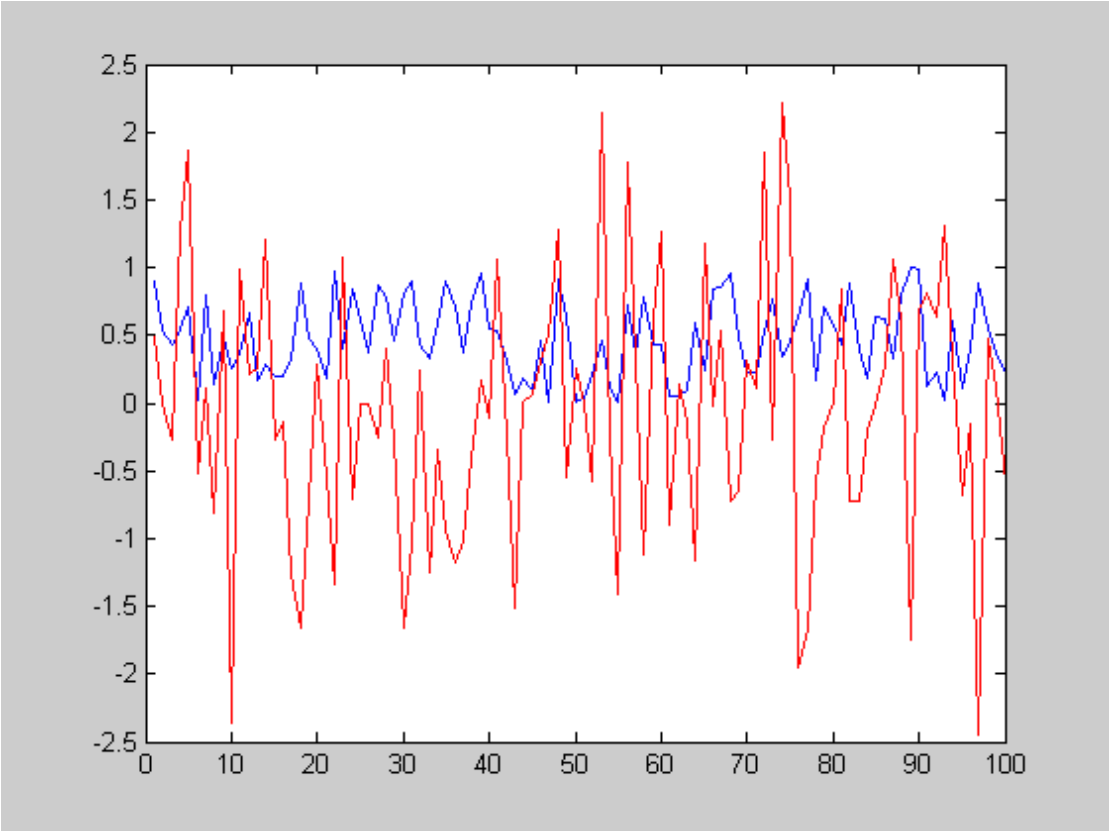
Please add labels to all your plots, k? K. K? K. To change the color or the dots, use help plot to get a list of things to add as the third argument to the plot function. So I can do

```
plot(t, x, 'rp-.');
```



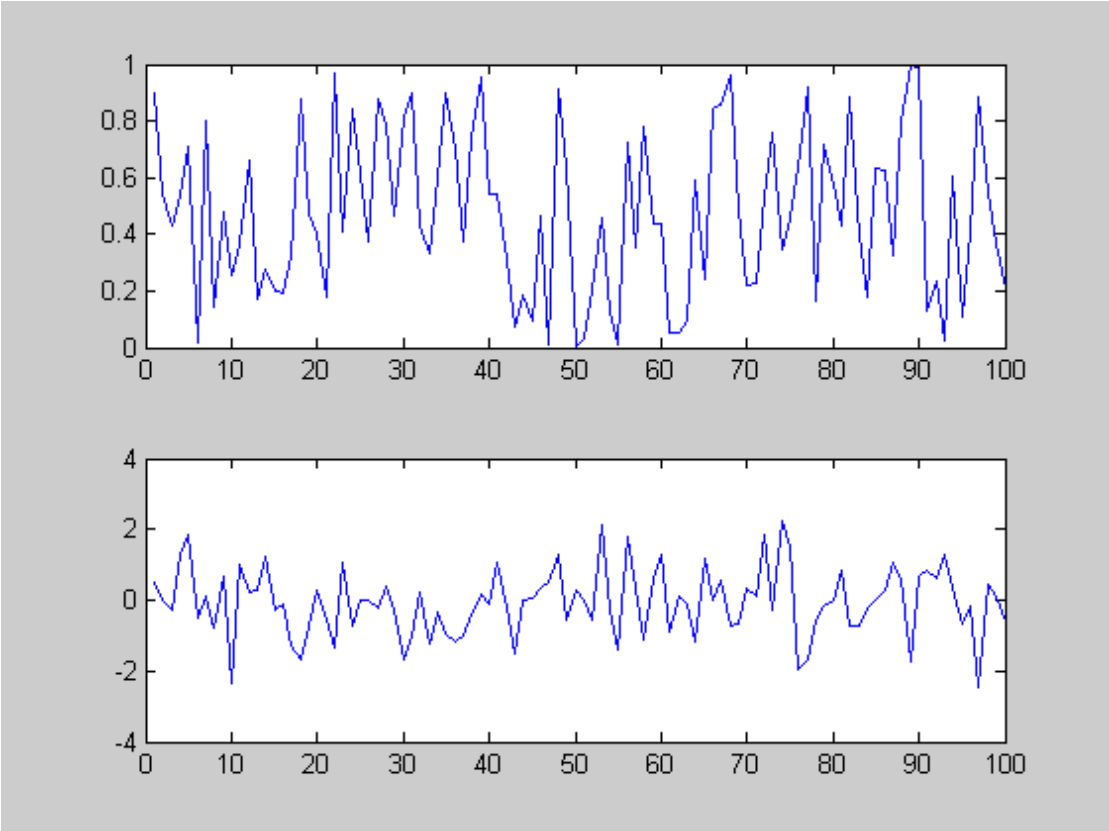
to make the plot red with pentagrams at the points and a dot-dashed line. As you guys get better you guys can mess around with the Figure window and make really good plots. We find it very important that you make good plots because good plots mean good work :) Suppose you wanted to plot two signals on the same axis. How do we do that?

```
x = rand(100,1);
y = randn(100,1); % Constant vector of 1.1
plot(x)
hold on, plot(y, 'r'), hold off;
```



hold on means every plot afterwards goes on the same axis, until you give a hold off command. If you would rather have the plots side by side on the same figure, use the subplot command

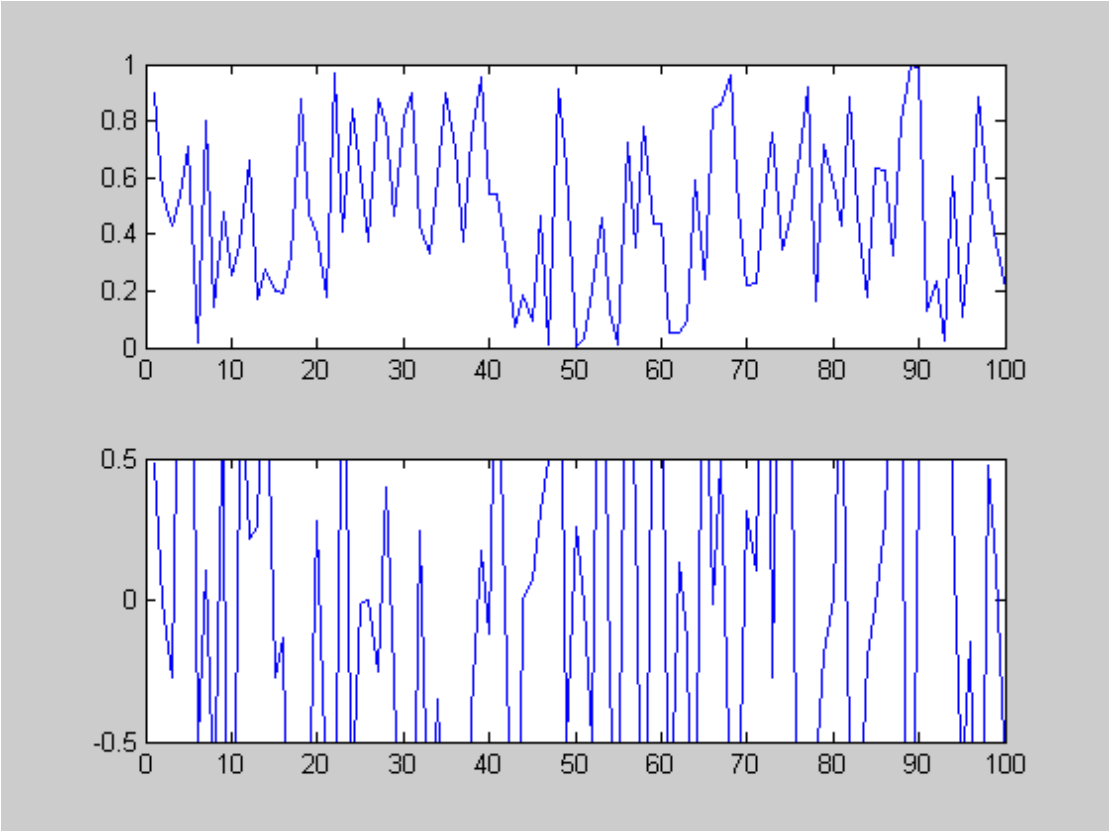
```
subplot(2,1,1), plot(x);
subplot(2,1,2), plot(y);
```



Subplot takes a # of rows and # of columns to make a "matrix of axes" within the figure. So we have 2 rows and 1 column, which means one plot on top of the other. Then the numbering is done row-wise first, so that 1 means the first axis. 2 means the 2nd axis on that row, etc. We can change

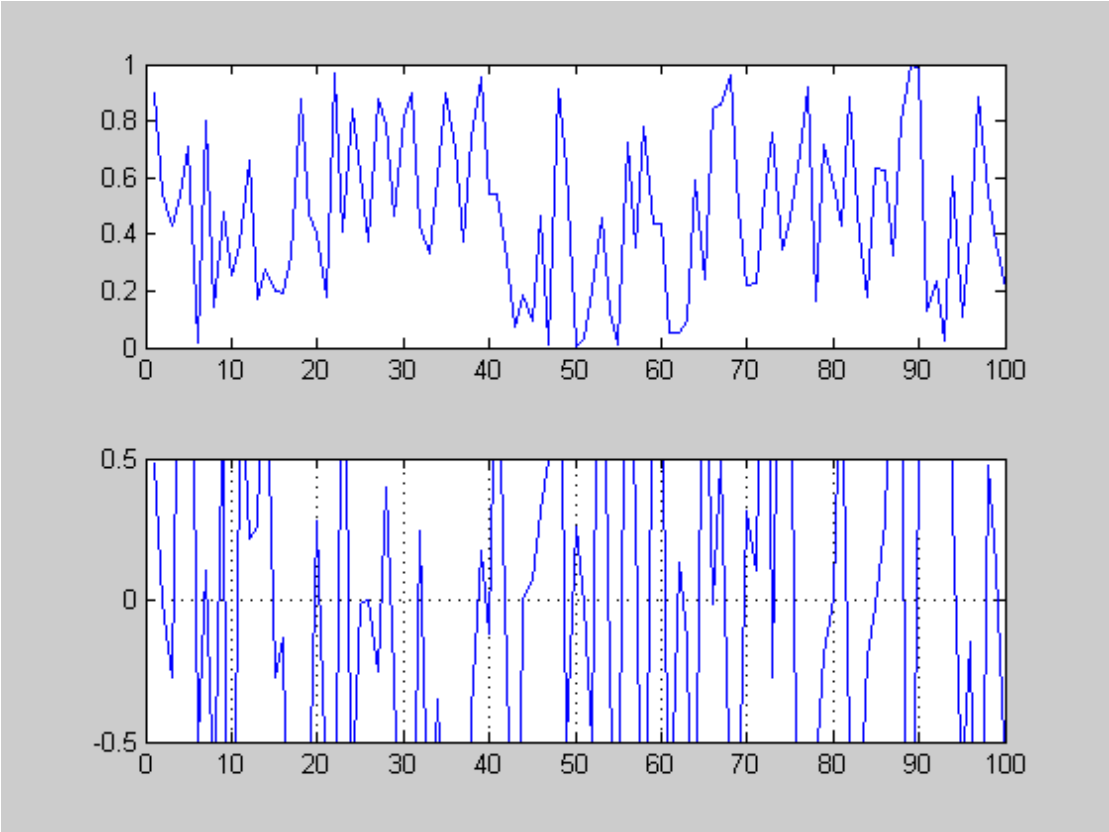
the axis of the second plot by doing

```
axis([0 100 -0.5 0.5]);
```



and we can turn the grid on (to see where the curve hits points better) using

```
grid on;
```



So we've gone over

- setting and initializing variables,
- vectors and matrices,
- concatenation,
- element-wise operations vs matrix operations,
- functions and scripts,
- for loops and the emphasis on vectorization, and
- plotting results.

This should get you a decent head start into MATLAB. Here's a list of other useful functions; use help to determine how they work:

- sum/max/min
- dlmread/dlmwrite
- size/length
- fprintf
- logical
- xor
- ceil/floor/round
- polyfit
- polyval
- interp1

Published with MATLAB® R2014a