14:440:127– Introduction to Computers for Engineers

Notes for Lecture 04

Rutgers University, Spring 2010 Instructor- Blase E. Ur

1 2-D Plotting

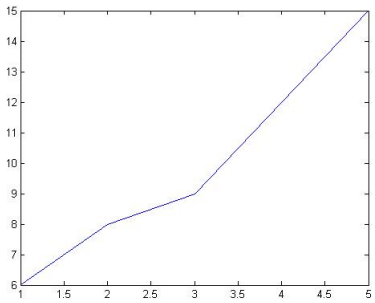
Matlab has a number of functions built-in that allow you to plot graphs, making it quite easy to create scientific reports using Matlab. The key idea is that Matlab uses vectors or matrices of numbers to specify which points to plot.

1.1 X-Y Line Plots

X-Y Line Plots are the most basic types of plots. The key idea is that you must give Matlab two separate (and equal size) vectors specifying the x and y points separately. If you ask Matlab to plot([1 2 3],[5 6 7]) , then the points (1,5), (2,6), and (3,7) are plotted. With the *plot* command, always be sure it has two input vectors of equal size. The first vector is the set of all x points, and the second vector is the set of all y points.

```
plot([ 1 2 3 4 5 ], [ 6 8 9 12 15 ])
```

The code above will plot the points (1,6), (2,8), (3,9), (4,12), and (5,15), and connect the points with a line. The following graph is displayed:



Of course, it often doesn't make sense to type in all of the x points and all of the y points by hand. Thus, if you wanted to plot $y = x^3$, you're more likely to type the following set of commands. Note that you first create a vector of all the x points, using either the colon operator or the *linspace* function. You then perform mathematical operations on x to get y, usually with element by element (dot) operators like .*, ./, +, or -

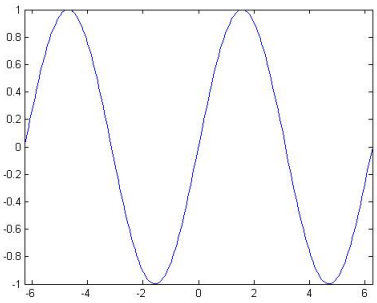
```
x = linspace(0,20,100);   % 100 points between 0 and 20
y = x.^3;
plot(x, y)
```

1.2 Function Plots

Using the *plot* command to plot x^2, you needed to create a vector of x points using *linspace*, and then calculate the y points using $y = x.^2$. However, you can also create a 'function plot,' which obviates creating those vectors. The *fplot* function takes a function (as a *character string*, in single quotes) and vector containing the range for the x points as its inputs. Note that we're taking a bit of a shortcut in the way we specify the function; we'll discuss this in the later lecture on symbolic math. The most important

change is that you should not use dot operations when specifying functions in this way.

```
fplot('sin(x)',[-2*pi,2*pi])
% note that sin(x) must be in single quotes
% -2pi,2pi is the interval for x.
```

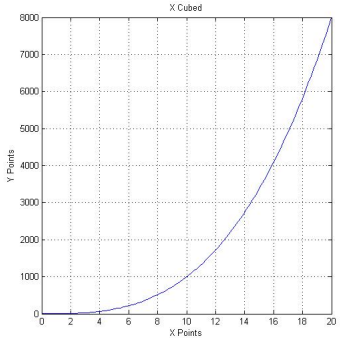


2 Tools for All Plotting Functions

2.1 Titles, Labels, Grids

To start making your graph look nicer, you can add things like titles and labels for the axes. To add a title to your graph, simply type *title('My First Graph')* right after you use the plot command. You must first create the graph, and then you can change the title. To add a label to the x axis, just type *xlabel('My X Axis')*, and similarly type *ylabel('My Y Axis')* for the y axis. For the title and labels, be sure to put the descriptions in single quotes since they are character strings. To add grid lines to your plot, type *grid on*. After turning the grid on, you can turn it off again using the command *grid off*. By default, the grid is off.

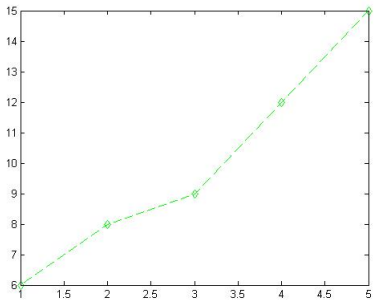
```
x = linspace(0,20,100);
y = x.^3;
plot(x, y)
title('X Cubed')
xlabel('X Points')
ylabel('Y Points')
grid on
```



2.2 Changing Line Colors Like a MySpace Kid

You can also change the color of the lines in a graph, or even whether the line is solid, dashed, etc., using an extra argument to the *plot* function. Rather than just typing *plot(x,y)*, you can type something like *plot(x,y,'- -dg')*. This command specifies that the lines in the plot should be dashed (- -), that all of the points in the plot should be highlighted as diamonds (d), and that the line should be green (g). Here's what that looks like:

```
x = 1:5;
y = [6 8 9 12 15];
plot(x, y,'--dg')
```



One interesting use of these supplements to the *plot* function is that you're able to create a scatter plot (displaying all of the points without any lines connecting them) with a call like `plot(x,y,'x')`, which displays each point as an X.

2.3 Opening Multiple Windows of Graphs

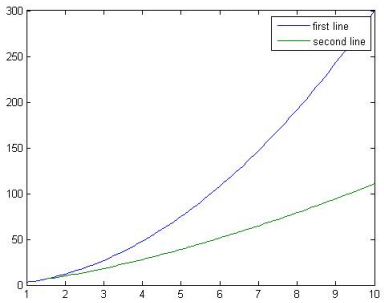
If you run a plot command in Matlab, and then run another plot command later in your program, the first plot will be overwritten and replaced by the second plot. However, if you instead want to have multiple windows open, with just one plot in each window, execute the *figure* command before each *plot* command, passing the number of the figure as the argument. The following code creates one figure, then creates a second, and then overwrites the first:

```
figure(1)
plot(xpoints, ypoints)
figure(2)
plot(x,y)
figure(1)  % go back and overwrite the first figure
plot(x,y)
```

2.4 Two Graphs, One Figure– Method One, and Legends

Rather than creating multiple figures, sometimes you want to plot multiple lines on one set of axes. There are two methods to do this. The first is to specify multiple sets of x points and y points. Let's say that we have x points stored in *x1* and *x2*, and y points stored in *y1* and *y2*. The *x1 and y1* points go together, and the *x2 and y2* points go together. In the following example, I use that method to graph two lines on one set of axes. You'll also notice that I was fancy and included a legend using the *legend* command. The first character string I send to the *legend* command will be matched with the first line that's plotted; the second character string with the second line that is plotted, and so on.

```
x1 = 1:0.1:10;
x2 = 1.5:0.1:10;
y1 = 3*x1.^2;
y2 = 3.5*x2.^(1.5);
plot(x1,y1,x2,y2)
legend('first line', 'second line')
```



2.5 Two Graphs, One Figure– Method Two

Rather than having one *plot* command with multiple sets of points, you can instead use the *hold on* command. Typing *hold on* indicates to Matlab that it should not overwrite the previous figure, but instead just add additional lines to that figure. Once you type *hold off*, though, Matlab will overwrite the current figure. Note that legends don't work very well using this method, so if having a legend is essential to your graph, use the first method. However, this method lets you have different kind of graphs in the same figure.

```
x1 = 1:0.1:10;
x2 = 1.5:0.1:10;
y1 = 3*x1.^2;
y2 = 3.5*x2.^(1.5);
plot(x1,y1)
hold on;
plot(x2,y2)
%%% later on
hold off;
```

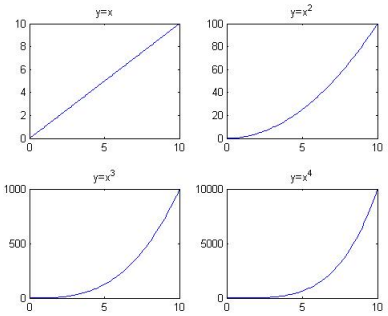
2.6 Subplots

Sometimes, you'll want to look at multiple graphs side by side in order to compare them. Matlab allows you to do this using the *subplot* command. Before plotting a figure, you use the *subplot* command to essentially create a grid of graphs (sets of axes) . There are three arguments to *subplot*: first, you indicate the total number of rows of graphs; second, you indicate the total number of columns of graphs. This is analogous to the way you consider the rows and columns of a matrix. Finally, you indicate which graph in the subplot you will currently reference. The graphs in the subplot are numbered from left to right within each row. Thus, the number system for a 2x3 subplot would be:

```
Graph 1   Graph 2   Graph 3
Graph 4   Graph 5   Graph 6
```

Here, we'll use *subplot* to display the graphs of four powers of x next to each other:

```
x = linspace(0,10,100);
y1 = x;
y2 = x.^2;
y3 = x.^3;
y4 = x.^4;
subplot(2,2,1);plot(x,y1);title('y=x');
subplot(2,2,2);plot(x,y2);title('y=x^2');
subplot(2,2,3);plot(x,y3);title('y=x^3');
subplot(2,2,4);plot(x,y4);title('y=x^4');
```



2.7 Getting Input from a Plot- Click Here!

Sometimes, when you display a graph, you want the user to click on it (and then have your program find out where they clicked). Use *ginput*, which is a function that allows you to get the coordinates of where in a graph the user clicks. To force the user to click only once, you'd use *[x y] = ginput(1)*. To instead keep allowing the user to click until they hit return, instead use *[x y] = ginput()*. The coordinates that are returned are based on the axes of the graph.

```
x = linspace(0,10,100);
y = x.^2;
plot(x,y)
[xclick yclick] = ginput(1);
fprintf('The user clicked on the point (%f, %f)\n',xclick,yclick)
```

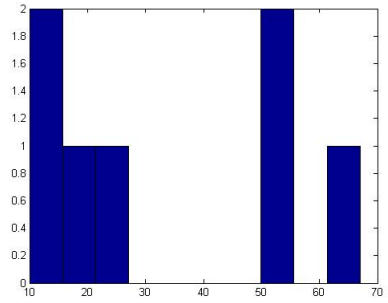
2.8 Saving Plots

When a figure is displayed, you can simply go to file-save as, and then save the plot you've just created. Be sure to change the file type to a common file format such as a .jpg file if you're going to use the figure outside of Matlab.

3 Histogram

Matlab also lets you create all sorts of other graphs; for instance, you can create a histogram using the *hist* function. The *hist* function takes two arguments: the data set, followed by the number of bins (which, by default, is 10).

```
hist([23 10 54 21 55 12 67])
```



There are also functions to make pie charts, box and whisker plots, and anything you ever wanted. Try searching Matlab's help feature for "Specialized Plotting".

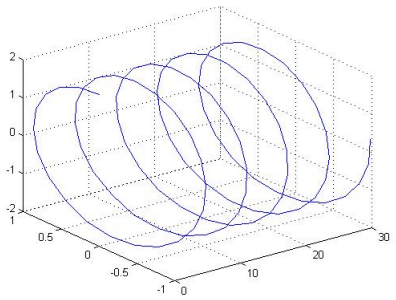
4 3-D Plotting

Matlab also lets you plot in 3 dimensions. You can plot a single line running through 3-D space (using *plot3*) by using 3 vectors (for x, y, and z), similar to the *plot* function. You can also plot surfaces, in which every combination of x and y points has a z value, or height. Since we need to calculate the height (z) for every combination of x and y, we need to use *meshgrid* to generate matrices of points. Then, we can use the *mesh* or *surf* functions to plot the surface, or create a top-down view of the surface's elevations using *pcolor* or *contour*.

4.1 3-D Line Plot

The simplest plot is a line plot, which is often used when y and z vary based on x. Using the *plot3* command, Matlab plots each (x,y,z) triplet and then connects those points. As with plot, the vectors of x points, y points, and z points need to be the same size. Note: the x, y, and z points must be VECTORS. For instance, you can create a corkscrew by typing the following:

```
x = linspace(0,30,100);
y=sin(x);
z=2*cos(x);
plot3(x,y,z)
grid on
```



4.2 3-D Surfaces

Whereas a 3-D line plot is essentially 2 variables (y and z) as a function of x, a 3-D Surface can be thought of as 1 variable (z) as a function of x and y.

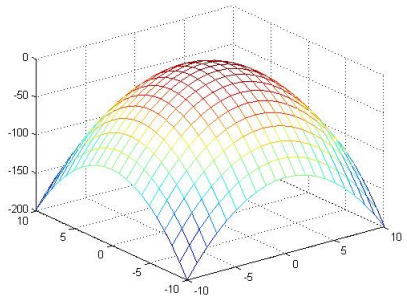
For every possible combination of x and y points, you need a value for z. Recall from Lecture 3 that you use *meshgrid* to do this. For instance, if you wanted to plot some surface for x and y running from 0 to 10, first create a *meshgrid* as follows. (Note that we want a lot of points, so we use 0.1 as our spacing for the *meshgrid*):

```
[x y] = meshgrid(0:0.1:10, 0:0.1:10);
```

4.3 Mesh

Now, you can take your *meshgrid*, perform element by element calculations on it, and then use the *mesh* function to create a wire-mesh 3-D surface:

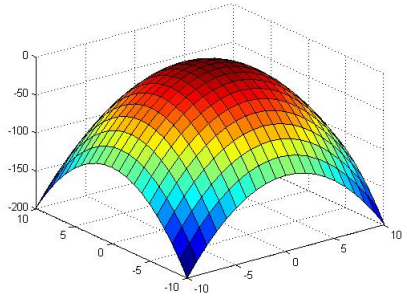
```
[x y] = meshgrid(-10:10, -10:10);
z = -x.^2-y.^2;
mesh(x,y,z)
```



4.4 Surf

Whereas the *mesh* function creates essentially a wire-mesh display of a surface, the *surf* function colors in the grid for you based on the elevation of z. Essentially, all points with the same z value will be the same color. To repeat the last example as a *surf* plot, just do:

```
[x y] = meshgrid(-10:1:10, -10:1:10);
z = -x.^2-y.^2;
surf(x,y,z)
```



4.4.1 Colormaps for your Surface

If you really want to pimp your Matlab graphs like a 13 year old's MySpace page, you can use the colormap command for a surface to specify a new colormap, which is essentially a mapping of z values to colors. In addition to typing *colomap jet*, which is the default, you can replace *jet* with *autumn, spring, summer, winter, bone, colorcube, cool, copper, flag, hot, hsv, pink, prism, white*. Enjoy.

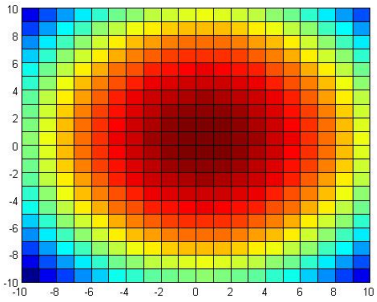
4.5 Pseudo Color Plot

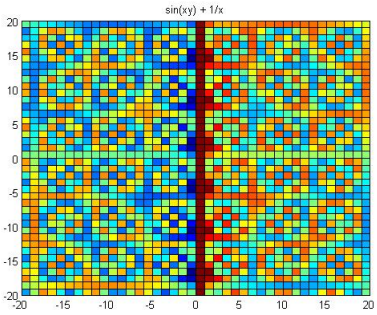
If you'd rather see a 2-D overhead representation of 3-D data where the z value is just represented by a color, you can use the *pcolor* (short for pseudocolor) function:

```
[x y] = meshgrid(-10:1:10, -10:1:10);
z = -x.^2-y.^2;
pcolor(x,y,z)

% second picture

[x y] = meshgrid(-20:20,-20:20);
z = sin(x.*y) + 1./x;
pcolor(x,y,z); title('sin(xy) + 1/x')
```

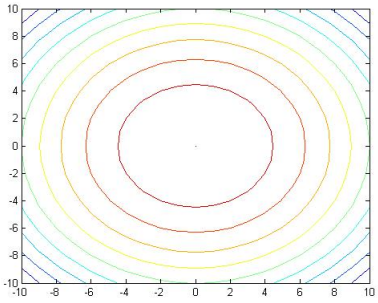




4.6 Contour Plot

Instead, if you want to just display lines that connect all of the points at the same z elevation, you can use the *contour* function. The lines drawn are known as contour lines or equipotential lines (if you work in EE/CE).

```
[x y] = meshgrid(-10:1:10, -10:1:10);
z = -x.^2-y.^2;
contour(x,y,z)
```



5 Interpolation

Oftentimes when you gather data, you'll want to estimate the values of intermediate points, which are points in between your measured data points. To do this in Matlab, you use *interpolation*, those methods that draw lines or curves between measured (or observed) points so that you can estimate intermediate values.

5.1 Linear Interpolation

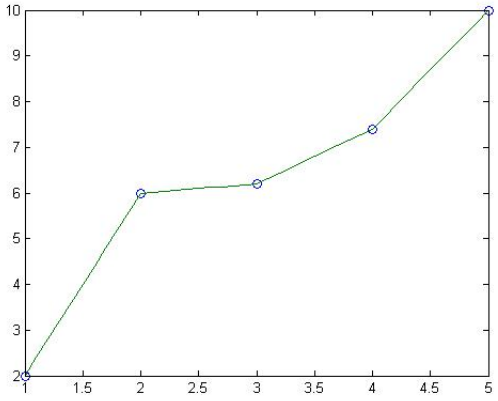
The simplest method of interpolation is linear interpolation, which basically encompasses drawing a line between each consecutive measured data point. Then, to estimate any intermediate value, you just find that point on those lines. Note that linear interpolation *doesn't* draw one line that fits all of the data

points. It draws many small lines that connect every consecutive point. Thus, each measured data point is the endpoint of a line.

To perform linear interpolation in Matlab, you use the *interp1* function. Note that the last character in the function's name is the number 'one,' *not* the letter 'L.' The *interp1* function requires 3 input arguments: a vector containing the x points you measured, a vector containing the y points you measured, and a third vector containing the new x values for which you want to estimate the values of y points. The function returns a vector of estimates for those y values.

The example below shows 5 'measured' data points, stored as x and y. We then use *interp1* to estimate y values for new x values from 1 to 5, in intervals of 0.1. Below, you can see the resulting plot, in which the measured points are simply O's, whereas the lines connecting those points are actually the interpolated values. Note that the interpolated lines go through each of the measured data points.

```
x = 1:5;
y = [2 6 6.2 7.4 10];
newx = 1:0.1:5;
newy = interp1(x,y,newx);
plot(x,y,'O',newx,newy)
```



5.2 Cubic Spline Interpolation

Instead of drawing lines between points, you sometimes instead want to draw curves that snake through each of those points. This method of using curves to smoothly connect points is known as *cubic spline interpolation* and is used extensively in computer graphics (i.e. keyframe animation), image processing, and motion control for robots.

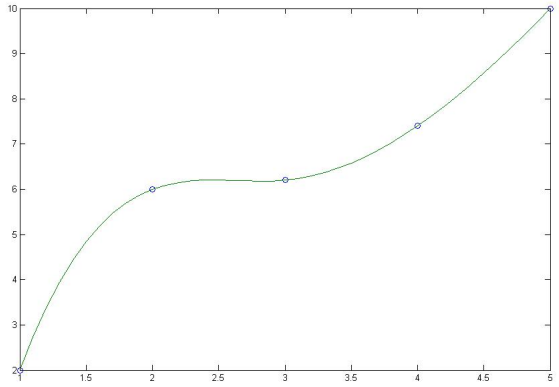
In *cubic spline interpolation*, we are essentially creating curves $Ax^3 + Bx^2 + Cx + D$ between each adjacent set of 2 measured points. Thus, n data points will result in *n-1* different curves. To calculate the coefficients for each of these points analytically, we use three different conditions for each polynomial. Recall that each curve connects two adjacent data points:

- The polynomial must pass through both of its endpoints (the two measured data points it connects).
- The slopes (first derivatives) of adjacent polynomials must be equal in order for the curve to be smooth.
- The curvature (second derivatives) of adjacent polynomials must be equal in order for the curve to be smooth.

This method takes quite a while by hand. Thankfully, Matlab does it for you using the *spline* function. As with *interp1*, you pass the *spline* function a vector containing the measured x points, a vector containing the measured y points, and a vector containing all of the new x points for which you wish to estimate y.

The example below shows the same 5 "measured" data points as before, stored as x and y. We then use *spline* to estimate y values for x values from 1 to 5, in intervals of 0.1. Below, you can see the resulting plot, in which the measured points are simply O's, whereas the curves connecting those points are actually the interpolated values. Note that the curves smoothly connect each of the measured data points.

```
x = 1:5;
y = [2 6 6.2 7.4 10];
newx = 1:0.1:5;
newy = spline(x,y,newx);
plot(x,y,'O',newx,newy)
```



6 Curve Fitting

Rather than taking our measured data points and estimating values for other x points in between our measured values, we sometimes want to calculate a "line or curve of *best fit*" for our data. Oftentimes, the line or curve of best fit won't pass through most, or even any, of the points. How then do we determine *best fit*? In the simplest cases, we use a method called *least squares regression*:

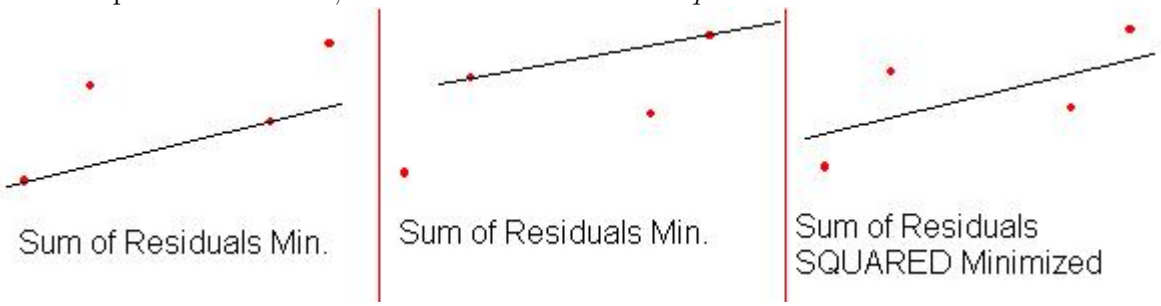
6.1 Least Squares Regression

We define the *residuals* of a fitted line or polynomial to be the vertical distances between the line/polynomial and each data point. The idea of least squares regression is to determine the line (or polynomial) that minimizes the sum of the squares of the residuals, hence the name "least squares." This method makes intuitive sense for 2 major reasons:

We need to take the absolute value of the residuals before summing them; equivalently, we can raise them to an even numbered power. Otherwise, the residuals of points that are above the fitted line and points that are below the fitted line will cancel each other out.

Minimizing the sum of the squares, rather than minimizing the sum, avoids having points that are very far away from the line. In fact, given 4 points, minimizing just the sum would make all three examples in the graphic below equally valid as fitted lines. In essence, passing through 2 of the data points (and thus being a distance of, say, x away from the 2 other points) is as valid as splitting the difference between all of the points (and thus being a distance of x/2 away from each of the 4 points). Since we want to find a trend

for the data, the third example in this illustration is the line we'd want since it best characterizes all of the data points. Therefore, we want to minimize the *squares* of the residuals.



6.2 Fitting to a Line Or Polynomial

To calculate a line or polynomial of best fit, we use the *polyfit* function. The *polyfit* function takes as input vectors of the x points and y points, and the degree of the polynomial. It returns a vector containing the coefficients of the polynomial, from the coefficient of the highest degree term to the coefficient of the lowest degree term.

```
x = [ 1 2 3 4 5];
y = [ 9 22 41 66 97];
coeffs = polyfit(x,y,2)

%%% We find that
>>coeffs =
    3.0000   4.0000   2.0000
```

This answer indicates that the polynomial of best fit is $3 * x^2 + 4 * x + 2$.

6.2.1 Polyval

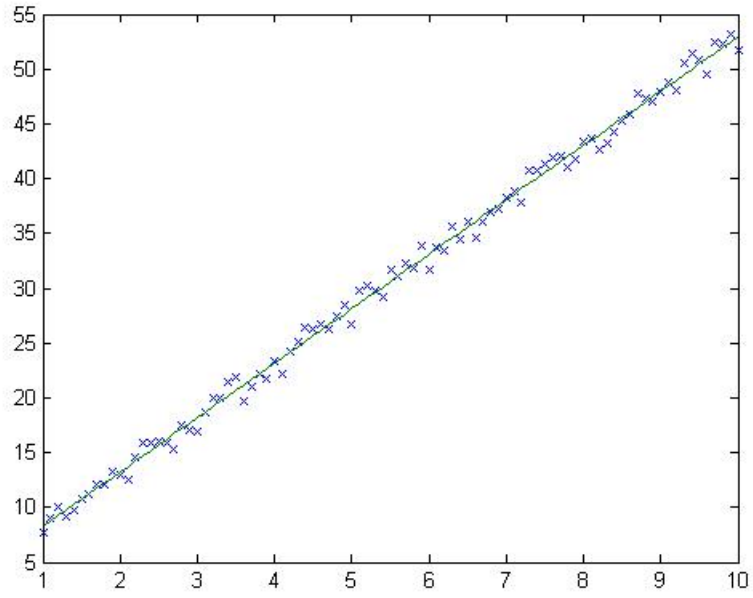
Since *polyfit* returns a vector containing coefficients, you might wonder how to efficiently calculate other points from those coefficients. There is a function called *polyval* that takes two input arguments: a vector containing the coefficients of a polynomial (from greatest to least degree), plus a vector of all the x points at which you'd like to evaluate that polynomial.

As a simple example, let's first use *polyval* to plot $y = x^2 + 3x + 2$:

```
x = 1:0.1:10;
y = polyval([1 3 2],x);
plot(x,y)
```

In the fairly complex example below, we create "noisy" data by adding random numbers between -1.5 and 1.5 to our y points. We use *polyfit* to calculate the coefficients of a first degree polynomial that best fits this noisy data, and use polyval to calculate the "idealized" y points. We plot both our original, noisy data points, and the line that *polyfit* finds on the same graph:

```
%%% I already made vectors x and y of data points, where y was noisy
%%% (In other words, y randomly varied to be too high or low)
coeffs = polyfit(x,y,1);
yfit = polyval(coeffs,x);
plot(x,y,'X',x,yfit)
%%% display the noisy points as X's, idealized points as lines
```



7 Fitting To More Complicated Functions

Now you know how to find the line of best fit for some data points if they are linearly related. You can use this fact, along with the idea of "linearizing data", to find more complicated relationships. "Linearizing" means making a few substitutions for variables, which result in an equation for a line. Note that "linearization" has some pretty complex ramifications about what it does to your assumptions about (and ability to determine) the experimental error of your data– if you're interested, see `http://en.wikipedia.org/wiki/Nonlinear_regression`

7.1 Mathematical Derivations

In a number of scientific processes, y and x are exponentially or logarithmically related i.e. *y=exp(x)* or vice versa. However, it's sometimes hard when looking at a graph to determine whether or not there is an exponential relationship, or what sort of exponential relationship exists. Matlab contains alternatives to the *plot* function that display exponential values for one or both axes, which is equivalent to taking the logarithm of x, y, or both x and y.

7.1.1 Logarithmic Data

If your data has logarithmic relationship of the form $y = K * log(x) + C$, it will look like a straight line when you plot the y values vs. the logarithm of the x values. (Note that this doesn't imply that all data that looks straight when plotted in this manner is necessarily logarithmically related, but that a logarithmic function roughly FITS the data points that are available).

To rephrase this point in Matlab-specific terms, logarithmic data will appear straight if you `plot(log(x),y))`. (Alternatively, you could have replaced this *plot* function with *semilogx(x,y)*, which makes the graph look the same by creating an exponential x axis). Why does taking the log of the x points make the graph look straight?

$y = K * log(x) + C$
SUBSTITUTE: $x_2 = log(x)$
$y = K * x_2 + C$, which is linear

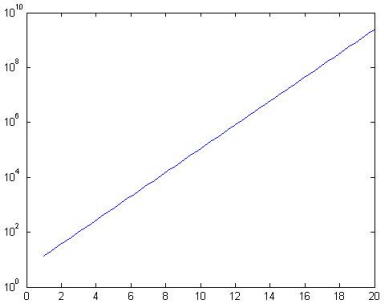
7.1.2 Exponential Data

If you data has an exponential relationship of the form $y = C * e^{Kx}$, plotting the logarithm of the y values vs. the x values will look like a straight line. To rephrase this point in Matlab-specific terms, logarithmic data will appear straight if you `plot(x,log(y))`. (Alternatively, you could have replaced this *plot* function with *semilogy(x,y)*, which makes the graph look the same by creating an exponential y axis). Why?

$y = C * e^{Kx}$
Take the natural log of both sides of the equation:
$log(y) = log(C * e^{Kx})$
$log(y) = log(e^{Kx}) + log(C)$
$log(y) = K * x + log(C)$
SUBSTITUTE: $y_2 = log(y)$ AND $C_2 = log(C)$
$y_2 = K * x + C_2$, which is linear
Solve for the coefficiencts K and C_2;
Remember that $C = e^{C_2}$!!!

Here's a *semilogy* plot in action:

```
x = linspace(1,20,100);
y = 5*exp(x);
semilogy(x,y)
```



7.1.3 Power Function Data

If you data has a power relationship of the form of the form $y = C * x^K$, plotting the logarithm of the y values vs. the logarithm of the x values will look like a straight line. Note that you might initially think that power functions are merely degree K polynomials and could just be solved using polyfit. However, consider the cases where K is not an integer or where you're trying to determine K from the data points and thus don't know what degree polynomial to specify for the regression.

To rephrase this point in Matlab-specific terms, power function data will appear straight if you `plot(plot(x),log(y))`. (Alternatively, you could have replaced this *plot* function with *loglog(x,y)*, which makes the graph look the same by creating exponential x and y axes). Why?

$y = C * x^K$
Take the natural log of both sides of the equation:
$log(y) = log(C * x^K)$
$log(y) = log(C) + log(x^K)$
$log(y) = K * log(x) + log(C)$
SUBSTITUTE: $x_2 = log(x)$, $y_2 = log(y)$, and $C_2 = log(C)$
$y_2 = K * x_2 + C_2$, which is linear
Solve for the coefficiencts K and C_2;
Remember that $C = e^{C_2}$!!!

7.2 Calculating The Coefficients

7.2.1 plot(log(x),y) Straight = Logarithmic Data

If the *plot(log(x),y)* plot looks linear, recall that $y = K * log(x) + C$ describes the data, and we've linearized it (in the previous section) to be $y = K * x_2 + C$ by substituting $x_2 = log(x)$. Thus, find the line of best fit of the log(x) points and y points, which returns [K C], and then use those coefficients to write the equation $y = K * log(x) + C$:

```
plot(log(x),y);   % looks straight
coeffs = polyfit(log(x),y,1);
K = coeffs(1)
C = coeffs(2)
```

7.2.2 plot(x,log(y)) Straight = Power Function Data

If the *plot(x,log(y))* plot looks linear, recall that $y = C * e^{Kx}$ describes the data, and we've linearized it (in the previous section) to be $y_2 = K * x + C_2$ by substituting $y_2 = log(y)$ AND $C_2 = log(C)$. Thus, find the line of best fit of the x points and log(y) points, which returns [K C_2]. Calculate $C = e^{C_2}$ and then use K and C to write the equation $y = C * e^{Kx}$:

```
plot(x,log(y));   % looks straight
coeffs = polyfit(x,log(y),1);
K = coeffs(1)
C = exp(coeffs(2))
```

7.2.3 plot(log(x),log(y)) Straight = Exponential Data

If the *plot(log(x),log(y))* plot looks linear, recall that $y = C * x^K$ describes the data, and we've linearized it (in the previous section) to be $y_2 = K * x_2 + C_2$ by substituting $x_2 = log(x)$, $y_2 = log(y)$, and $C_2 = log(C)$. Thus, find the line of best fit of the log(x) points and log(y) points, which returns [K C_2]. Calculate $C = e^{C_2}$ and then use K and C to write the equation $y = C * x^K$:

```
plot(log(x),log(y));   % looks straight
coeffs = polyfit(log(x),log(y),1);
K = coeffs(1)
C = exp(coeffs(2))
```