MATH 3670 - SCIENTIFIC COMPUTATION I
Fall 2015

Week 2: Introduction to Symbolic Math (Chapter 11). 2-D Plotting (Chapter 5)

Content

Symbolic Computations
- Numeric vs. symbolic math (approximate vs. exact math)
- Converting symbolic values to numeric values
- Review of some symbolic math capabilities:
  - · expand or factor equations
  - · simplify or "beautify" equations
  - · solving equations
  - · differentiate / integrate symbolic functions

2-Dimensional Plotting
- plot an array of x-values vs. array of y-values (`plot()`)
- plot a function (`fplot()`)
- plot annotations: title, legend (specification, positioning), axis labels, formatting

• **Homework # 2**: (Due Friday, Sep 11th)
Chapter 11, pp 384–392 # 5, 8, **19***, 20, **23***
Chapter 5, pg 165 - 167 # 13, 14, 17, 20, **40***
(**Starred*** assignment is mandatory for MATH 3670 students)

Tip of the Day:

In a MATLAB program, no reference can be made to any object unless that object has been somehow "defined". For example, if the first statement is:

```
>> x = 4*y;
```

The result is:

```
??? Undefined function or variable 'y'.
```

Note that it only complained about `y`. The characters `4` and `=` have predefined meanings to MATLAB but the character y does not at this point. So, it is good practice to "play computer yourself", at least in the sense that every line of MATLAB code should be inspected for a possible reference to some object that has yet to be defined.

1. Symbolic Math Toolbox

So far, we have dealt with "numeric" math... no variable can be used unless it has already been assigned a value. Many times, the numeric value used is only an approximation. For example, entering:

```
>> x = 1/3;
>> oneThird = vpa(x)
```

shows how the number 1/3 is stored on this machine (to 32 decimal places):

```
oneThird =

      0.33333333333333331482961625624739
```

The function `vpa()` stands for Variable Precision Arithmetic and allows any number of digits of a result to be displayed.

The point is, most rational numbers can only be approximated in a digital computer. In fact, the only fractions that the computer can represent exactly are those that are sums of fractional powers of 2 (i.e. $1/2, 1/2^2, 1/2^3, ...$*).*

Symbolic math resolves this problem. The main difference between symbolic and numeric variables is that the names of the independent variables has to be declared before the computer can operate with them symbolically. So every time you want to work with a variable, say x, that was not used before, one needs to declare it:

```
>> syms  x
```

If in turn x is a function of other symbolic variables then those variables have to be defined, as in the example below:

Example: Let's define 2 symbolic variables:

```
>> syms k m
>> k=sym(1); m=sym(3);
```

NOTE that we initially defined x = 1/3 which resulted in x being a numeric variable whose value was approximately 1/3. Now, we will redefine the variable x in terms of one or more symbolic variables (i.e. k and m), so x then becomes a symbolic variable.

```
>> x = k/m
```

Which results in the following output:

```
x =

      1/3
```

NOTE that x is now a symbolic variable with value of 1/3 as opposed to the first usage of x where it was a numeric variable with a value CLOSE TO BUT NOT EQUAL TO 1/3.

We can convert a symbolic variable's value to a numeric value by using MATLAB's `double()` function which converts the specified symbolic variable (x) to a numeric value and assigns it to y:

```
>> y = double(x)
```

IMPORTANT: symbolic arithmetic is very slow for the computer compared to numeric arithmetic. Some simple numeric calculations can be done in a fraction of a second but would take hours if done symbolically.

2. Usages of Symbolic Math

• Expressions can be expanded or factored:

First, one can define an expression (called S below) using symbolic variables; this causes S to also be symbolic

```
>> syms a x y % declare variables a, x, y as symbolic variables
>> S = -(a-x) * (x+4) * (x+5)
```

Since S was defined in terms of a symbolic variable, S itself is now a symbolic variable
Now, use the `expand()` function to symbolically multiply these terms together:

```
>> T = expand(S)
```

The variable T is now a symbolic variable that consists of equation S multiplied out:

```
T =

     20*x - 20*a - 9*a*x - a*x^2 + 9*x^2 + x^3
```

MATLAB's `factor()`function is the inverse of expand():

```
>> U = factor(T)
```

Results in:

```
U =

       -(x + 5)*(x + 4)*(a - x)
```

• Expressions can be made to look "pretty" or simplified:

Create a symbolically defined expression:

```
>> syms x
>> S = (x^3 - 4*x^2 + 16*x) / (x^3 + 64)
```

Now, display the above equation in a more readable form using MATLAB's `pretty()` function:

```
>> pretty(S)
```

Which results in the following output:

```
   3      2
  x  - 4 x  + 16 x
  ----------------
       3
      x  + 64
```

Now, simplify this equation using MATLAB's `simplify()` function:

```
>> T = simple(S)
```

Which results in:

```
T =

      x/(x + 4)
```

• Equations can be solved using MATLAB's `solve()` function:

Suppose we want to solve $e^{2x} = 5$.

```
>> syms x
>> f = exp(2*x)-5
```

NOTE: the `solve()` function below sets the expression equal to zero so we defined the expression to be $e^{2x} - 5$. Now, solve the this equation $e^{2x} - 5 = 0$ using MATLAB's `solve()` function:

```
>> k = solve(f)
```

The variable k is now a symbolic variable whose value is the symbolic solution to $e^{2x} - 5 = 0$:

```
k =
       log(5)/2
```

We can display an approximate numeric value of this solution using the `double()` function:

```
>> v = double(k)
```

```
v =
       0.804718956217050
```

REMEMBER: the above value is the computer's approximate representation of log(5)/2 whereas the symbolic variable k has value of 'log(5)/2'

NOW, let's create and solve an equation with more than one variable:

```
>> ff = '4*t*h^2 + 20*t - 5*g'
```

NOTE that this equation has 3 variables, t, h, g, none of which were symbolically defined. Also note that the symbol 'ff' is defined as a character string with the desired equation. Use MATLAB's `solve()` function to solve this equation with respect to h. NOTE that variable h is referenced as character 'h':

```
>> ffs = solve(ff, 'h')
```

This results in:

```
ffs =
         -(5^(1/2)*(g - 4*t)^(1/2))/(2*t^(1/2))
           (5^(1/2)*(g - 4*t)^(1/2))/(2*t^(1/2))
```

NOTE how there are 2 solutions which should be expected since the variable h in the original equation is of 2nd order.

If a variable is not specified, then the `solve()` function decides which variable to solve for according to rules specified in section 11.1.3 (pg 353)

```
>> ffs = solve(ff)
```

Since no variable was specified, the `solve()` function decided which variable is to be solved for in accordance to the rules specified in section 11.1.3 which, in this case, is variable t:

```
ffs =
         (5*g)/(4*h^2 + 20)
```

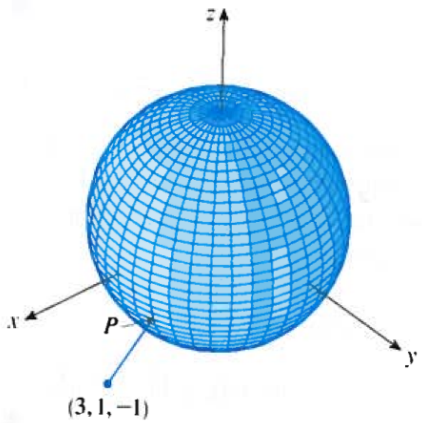
• Systems of equations can be solved (sometimes)

Several equations can be solved simultaneously using the `solve()` function.

Let's do a math problem: Find the points on the sphere

$$x^2 + y^2 + z^2 = 4$$

that are closest to and farthest from the point (3, 1, -1)



This is a Calc III problem easily solved using the **Lagrange Multiplier** method.
First, note the distance from a point (x, y, z) on the sphere to the point $(3, 1, -1)$ is given by

$$d = \sqrt{(x-3)^2 + (y-1)^2 + (z+1)^2)}$$

Squaring both sides for easier manipulation yields the function

$$f(x, y, z) = (x-3)^2 + (y-1)^2 + (z+1)^2$$

The method of Lagrange multipliers allows us to find the maximum and minimum of this function (square of distance between the given point and the surface of the given sphere) subject to the constraint equation (of the sphere):

$$g(x, y, z) = x^2 + y^2 + z^2 - 4 = 0$$

The method requires to solve the system of 4 equations with 4 unknowns

$$\frac{\partial f}{\partial x} = \lambda \frac{\partial g}{\partial x} \tag{1}$$
$$\frac{\partial f}{\partial y} = \lambda \frac{\partial g}{\partial y} \tag{2}$$
$$\frac{\partial f}{\partial z} = \lambda \frac{\partial g}{\partial z} \tag{3}$$
$$g(x,y,z) = 0 \tag{4}$$

In our case this system reads

$$2(x-3) = 2x\lambda \tag{5}$$
$$2(y-1) = 2y\lambda \tag{6}$$
$$2(z+1) = 2z\lambda \tag{7}$$
$$x^2 + y^2 + z^2 = 4 \tag{8}$$

This is a system of NONLINEAR equations, for which there is hardly any method to predict the number of solutions. Let's write a little script file that uses the MATLAB function `solve()` to solve this system of equations:

```
% Define the 4 symbolic variables
syms x y z lambda

% Define the 4 equations.  Note how these equations have been rewritten to equal zero
eq1 = 2*(x - 3) - 2*x*lambda;
eq2 = 2*(y - 1) - 2*y*lambda;
eq3 = 2*(z + 1) - 2*z*lambda;
eq4 = x^2 + y^2 + z^2 - 4;

% Solve the 4 equations simultaneously, specifying the 4 variables explicitly
[la,xa,ya,za] = solve (eq1, eq2, eq3, eq4, x, y, z, lambda)
```

The output is:

```
la =
  1 - 11^(1/2)/2
 11^(1/2)/2 + 1

xa =
   (6*11^(1/2))/11
 -(6*11^(1/2))/11

ya =
   (2*11^(1/2))/11
 -(2*11^(1/2))/11

za =
 -(2*11^(1/2))/11
  (2*11^(1/2))/11
```

Often, we prefer to convert the symbolic answer to numeric format (in this case only the x,y,z values are of interest)

```
>> xa=double(xa);
>> ya=double(ya);
>> za=double(za);
```

To single out the points (x, y, z), simply use the command

```
>> P1=[xa(1), ya(1), za(1)]
>> P2=[xa(2), ya(2), za(2)]
```

```
P1 =
    1.8091    0.6030   -0.6030
P2 =
   -1.8091   -0.6030    0.6030
```

It is now easy to show that P_1 is closer than P_2 to (3, 1, -1) so ...

Conclusion: The closest point on the given sphere to (3, 1, -1) is (approximately) $P_1(1.8, 0.6, -0.6)$ and the farthest point on the sphere is $P_2(-1.8, -0.6, 0.6)$.

• Differentiation and Integration of symbolic expressions:

Define expression $e^(x^4)$ and then differentiate with respect to x using MATLAB's `diff()` function:

```
>> syms x
>> S = exp(x^4)
>> dS = diff(S)
```

The value of the symbolic variable dS is the differentiation of $e^(x^4)$:

```
dS =

4*x^3*exp(x^4)
```

Let's do another expression: $5x^2\cos(3t)$

```
>> syms x t
>> y = 5*x^2 * cos(3*t)
>> dy = diff(y,x)
```

Results in:

```
dy =

10*x*cos(3*t)
```

Now, for integration. First, indefinite integration is illustrated by using MATLAB's `int()` function to integrate the above expression $y = 5x^2\cos(3t)$

```
>> iy = int(y)
```

which results in:

```
iy =

(5*x^3*cos(3*t))/3
```

DEFINITE INTEGRATION
Below, equation y is integrated with respect to x and then evaluated between x = 0 and x = pi. The output is expressed in terms of unknown variable t

```
>> iy = int(y, x, 0, pi)
```

Which results in:

```
iy =

(5*pi^3*cos(3*t))/3
```

But what if we want an actual number which one might expect from definite integration? The issue is that the original equation was expressed in 2 variables, x and t. Integration was performed with respect to x so variable t is part of the result. We can determine the value of this result for specific values of t by using MATLAB's `subs()` function.

Below are 2 examples of seeing what the value of the above definite integration would be with $t = 3$ and $t = 4.6$:

```
>> subs(iy, t, 3)
```

Results in:

```
r =
         -47.084594986317207


>> subs(iy, t, 4.6)
```

Results in:

```
r =
          17.095562726079695
```

SUMMARY: Things to remember so far

All objects referenced in any MATLAB program such as variable names must be appropriately defined prior to any usage.

Numeric objects are defined by setting a value to them when first referenced:

```
>> y = 4
```

Symbolic objects are defined by declaration WITHOUT setting a value to them:

```
>> syms y
```

• **Numeric vs. Symbolic:**
Numeric values/computations
- Most numeric values in any digital computer are approximations.
- All numeric computations require all variables used to already have values

Symbolic values/computations
- Symbolic values and computations are "exact"
- Any declaration that uses one or more symbolic objects is itself symbolic
- Symbolic manipulation can be exceedingly slow for the computer to perform
- Symbolically defined functions can be solved for in general terms or for specific values
- Symbolically defined functions can be differentiated and integrated (definite, indefinite)

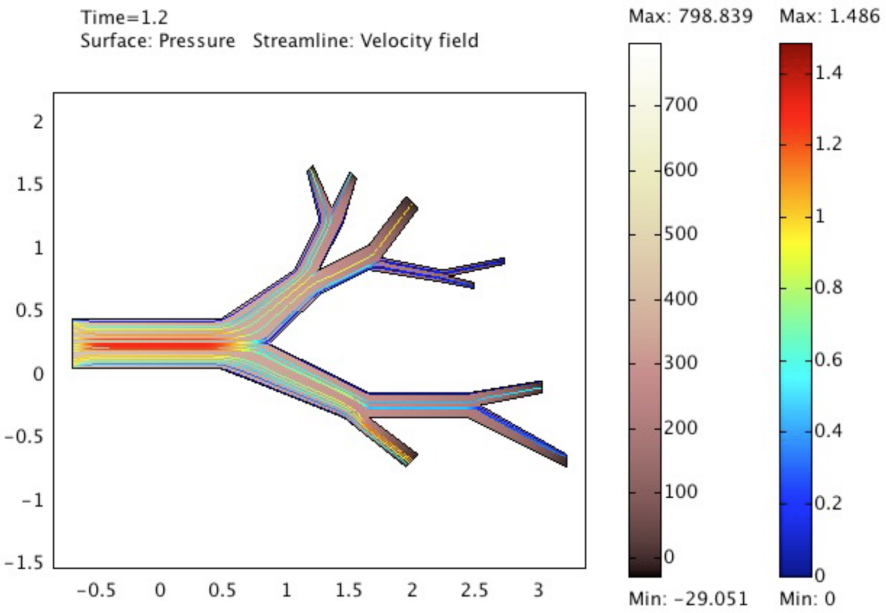
• **Numeric values from symbolic variables:**
If a symbolic variable's value consists only of numbers (no variables), then usage of MATLAB's double() function will convert the symbolic value to a numeric value

If a symbolic variable contains one or more variables, a numeric value of this symbolic variable can be obtained by usage of MATLAB's subs() function.

---

3. Basic Plotting

One of the powerful features of MATLAB is its versatility in producing publication-ready graphics. Plotting numerical data is versatile since it gives the user the ability to control almost every aspect of the plot. Here is a sample plot:

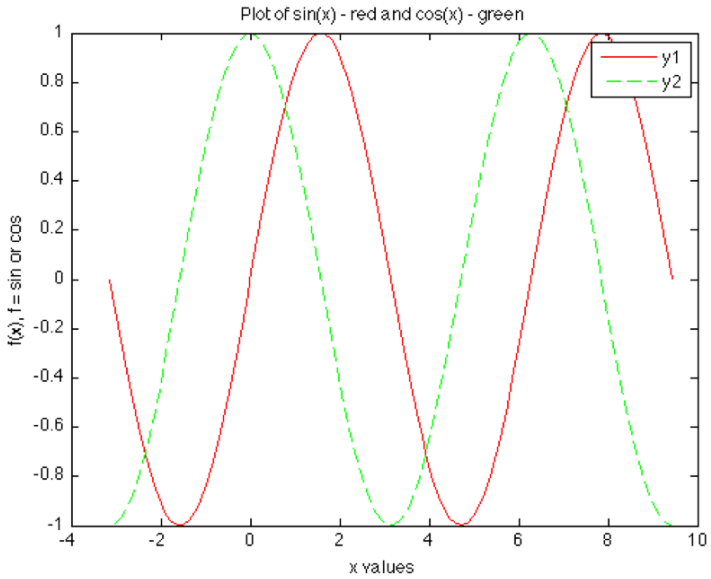


To see a few of the plotting capabilities, let's make a basic plot by typing:

```
>> x = linspace(-pi, 3*pi, 100);
>> y1 = sin(x);
>> y2 = cos(x);
>> plot(x, y1, 'r', x, y2, '--g');
>> legend('y1', 'y2');
>> title('Plot of sin(x) - red and cos(x) - green');
>> xlabel('x values');
>> ylabel('f(x), f = sin or cos');
```

The above code snippet:
- created the 100 element row vector **x** with 100 equally spaced points between $[-\pi, 3\pi]$
- created the 100 element row vector **y1** whose values are the sine of the x-elements
- created the 100 element row vector **y2** whose values are the cosine of the x-elements
- created a Plot y1 vs x, made it red by specifying 'r' and also plot y2 vs. x, made it dashed green via '–g'
- added a legend - both plotting variables are automatically assigned their colors in the legend
- created a title
- annotated the x and y axes

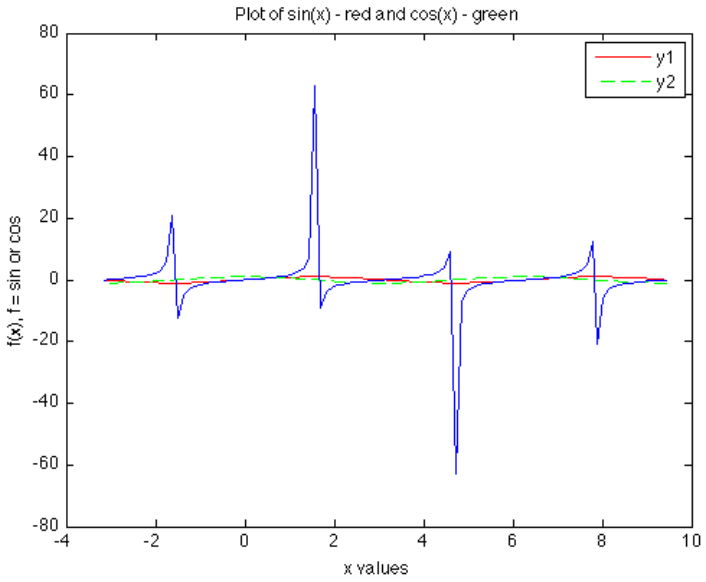
These commands generate the following output:



Let's add another plot on this same graph:

```
>> y3 = tan(x);     % y3 is a vector of same dimension as x above
>> hold on          % This is how to add another plot onto an existing one
>> plot(x,y3,'*') % plot y3 vs. x
```

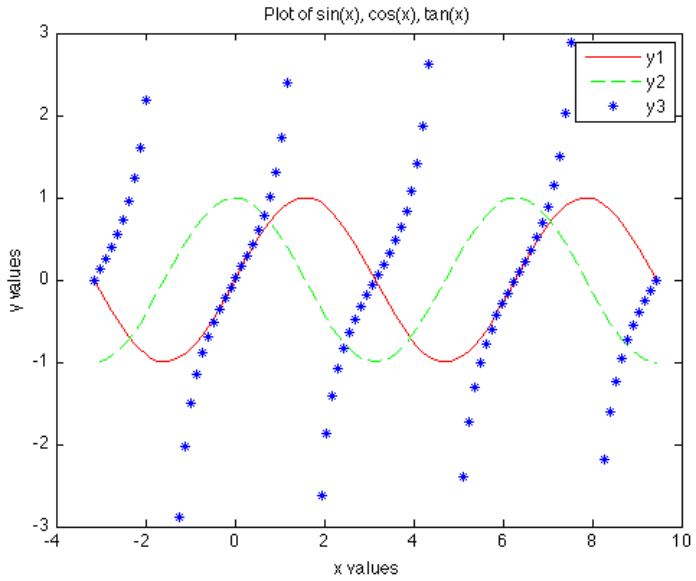
And this produces the following:



This is not satisfactory because the red and green lines (sin and cos) are now almost horizontal lines due to the much expanded y-axis which goes from [-80, 80] instead of [-1 1] as in the first plot.
Let's force the y-axis range to [-3 3]:

```
>> axis([-4 10 -3 3]);
>> legend('y1', 'y2', 'y3');
>> title('Plot of sin(x), cos(x), tan(x)');
```

Which produces:



Note how the `axis()` function forced the x-axis range to be [-4 10] and y-axis to be [-3 3] Also, it was necessary to redo the same plot command with this new range of the axes.
As explained on Page 143, additional plots can be added using the `line()` function which

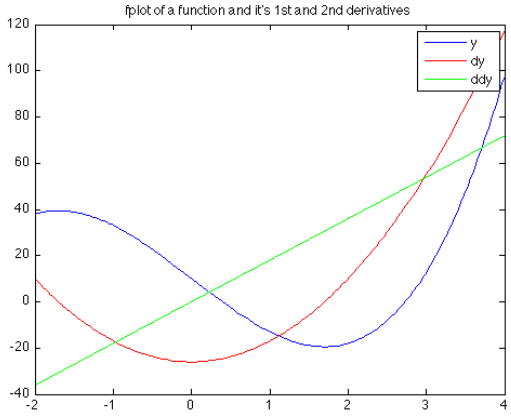
is nearly the same as what was done here with the `hold on` and `plot()` usage.

The `fplot()` function allows one to see a plot of a function without creating arrays of data as is required in the `plot()` function. As explained in the help for this the `fplot()` function, the function must be specified as a character string, not as a numeric expression as done above nor a symbolic expression. To illustrate, let's plot a function and it's first and second derivatives on the same plot:

```
>> y = '3*x^3 - 26*x + 10';
>> dy = '9*x^2 - 26';
>> ddy = '18*x';

>> figure
>> fplot (y, [-2 4], 'b')
>> hold on
>> fplot (dy, [-2 4], 'r')
>> fplot (ddy, [-2 4], 'g')
>> legend('y', 'dy', 'ddy');
>> title ('fplot of a function and it''s 1st and 2nd derivatives');
```

Which produces:



Things to note about the above code snippet:
a) With `fplot()`, the functions are defined as string variables (they are in quotes). A symbolically defined function will not work. This is why the `diff()` function was not used to differentiate; `diff()` requires the input function to be symbolically defined.

b) The `figure` command starts a new figure. This was required because of the previous `hold`on which will cause all future plots to be put onto the same graph until this `figure`(or `hold`off) command was executed.

c) `fplot()` requires as inputs the function followed by the x-domain (range of x values), followed by optional line characteristic identifiers.

MORE EXAMPLES (issues applicable to homework 2)

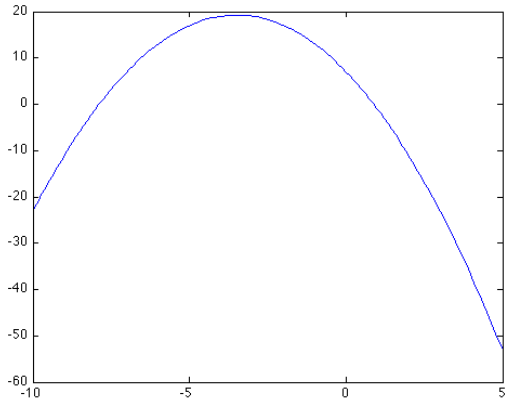
1) What if you are asked by an architect to create a plot that models an arch that she wants to design. She says that this arch will be built according to the following formula:

$$f(x) = -x^2 - 7x + 7. \tag{9}$$

How would you proceed?
A first attempt could go something like this:

```
>> fn = '-x^2 - 7*x + 7'; % Define the function given by the architect
>> fplot (fn, [-10, 5]);  % Plot this function with arbitrarily chosen limits
```

This snippet would produce:



Notice that much of the plot of this arch has negative values. The function provided by the architect is defined for all values of x. However, she is likely to be only interested in the part of this arch that is above ground; i.e. the part of the arch where the y-values are at least 0. What is an efficient way to plot only the part of the arch that is above ground (y at least 0)?
One way is to use MATLAB's `solve()` function to obtain the x-value limits, then use these limits in the `fplot()` function instead of the above "guess" limits.

REMEMBER that the `solve()` function requires the function to be symbolically defined. However, `plot()` or `fplot()` do not allow for symbolically defined functions. What to do?

One way is to symbolically define this function for the purpose of finding these limits, then use `fplot()` (with the string version of the function) to plot the function with the limits computed by the `solve()` function:

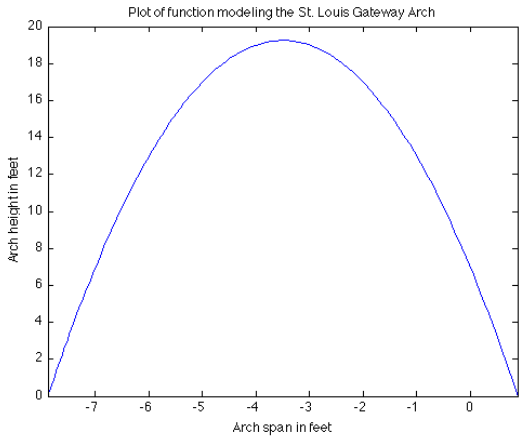
```
>> syms x
>> limits = double(solve('-x^2 - 7*x + 7'));
>> fn = '-x^2 - 7*x + 7';
>> fprintf ('The function describing this arch''s shape has positive values between\n');
>> fprintf ('  %6.2f and %6.2f feet - represents the arch''s ground level footprint\n',..
>>     limits(1), limits(2));
```

We will soon cover the `fprintf()` function in detail but these 2 `fprintf()` statements produces:

```
The function describing this shape has positive values between
   -7.89 and   0.89
```

Now use the `fplot()`:

```
>> fplot (fn, [limits(1), limits(2)]);
>> title('Plot of function modeling the St. Louis Gateway Arch');
>> ylabel (' Arch height in feet');
>> xlabel (' Arch span in feet');
```



As you can see, this plot consists only the part of the function that has positive values.

Notes from the above code snippet:
- `solve()` was used to find values of x where this function is = 0. This function is a quadratic so that there will be 2 solutions for x. Since the 2nd order term is negative, it is open pointing down which means that all function values between these 2 solutions will be positive.
- REMEMBER that the output of the solve() function are symbolic variables that need to be converted to numeric values via the double() function.

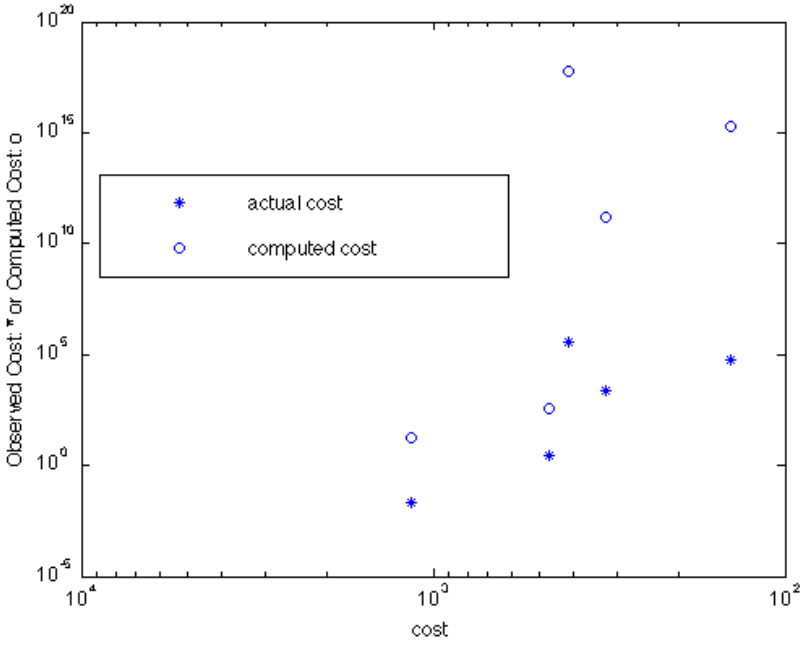
2) Let's say that you want to model production costs of 5 items. And let's assume that your factory has done a few preliminary studies showing the production costs of each of the 5 products. Also, assume that your economic theoretician has come up with a formula that computes the long term production costs for each of these 5 products which depends on a relationship between the quantity of raw materials and each raw material's costs.

So, we will have 3 sets (vectors) of 5 elements: empirical product costs for the 5, each product's raw material cost and the quantity of the 5 raw materials needed for each of the 5 products.
The first vector consists of the actual cost of the 5 items, the 2nd being each prod and material and product and also theoretically derived function that supposedly relates cost

and material to product. And say that you are tasked to plot the actual values of the empirically determined values as well as what these values would be as predicted by this formula.
Consider the following snippet:

```
>> cost = [143 467 324 1159 412];  % preliminary guess of each product's cost
>> rawMaterialCost = [54265 3 2345 .02 362363];
>> rawMaterialAmount = [5, 9, .43, 6, .2];
>> y = 3*rawMaterialAmount + 13*rawMaterialCost.^3; % the theoretical cost
>> loglog (cost, rawMaterialCost, '*');
>> hold on;      % allow for another plot on this graph
>> lineplot = loglog (cost, y, 'o');
>> leg = legend('actual cost', 'computed cost');
>> rect = [0.15, 0.55, .45, .15];
>> set(leg, 'Position', rect)
>> set (gca, 'XDir', 'reverse');
>> xlabel('cost');
>> ylabel ('Observed Cost: * or Computed Cost: o');
```

This produces the following graph:



Notes from the above code snippet:
- The first 3 lines create vectors of 5 empirically derived associations between product cost, material cost and material quantity
- the variable **y** is a theoretical formula that associates these 3 things
- the `loglog()`function produces a line plot where both axes are logarithmic and it's "handle" is assigned to a variable called **lineplot** so that it can be used later
- note that an asterisk is placed at each point without connecting line via the single asterisk

- the `legend()` function is used and it's "handle" is assigned for use in the next statement:
- the **rect** variable defines a rectangle's position and location so that the legend can be placed so that it does not cover data points on the plot (I had to experiment to find a good location for the legend)
- the `set()` function is used to both set the legend's location and then...
- the next `set()` function tells the plot to reverse the x-axis (just like the homework problem)

SUMMARY: Things to remember

- **2-Dimensional Plots**

General Hints: - Some may find it useful to start their script files with:

```
>> clc        % clears the command editor window
>> clear      % clears the work space variables
>> close all  % closes all existing figure windows
```

These commands will allow one to start with a clean slate - an environment uncontaminated by previous usages of MATLAB since it was invoked. Without cleaning the environment, it can happen that one's script execution only works because of some previously defined variables that the current script depends on. This will reduce the intermittent occurrence of "well it ran the last time I tried it" .

The `plot()` function requires one or more pairs of arrays, each pair being a set of x - y values to be plotted such as:

```
>> x = linspace(-pi, pi, 100);
>> y1 = cos(x) .* sin(2*x); % NOTE usage of .* (why??)
>> y2 = 2 * cos(2*x) .* cos(x) - sin(2*x) .* sin(x);
plot (x, y1, x, y2);
```

Whereas the `fplot()` function allows one function, specified as a character string to be plotted within the specified range such as:

```
>> fn = 'cos(x) * sin(2*x)';
>> fplot (fn, [-pi, pi]);
```

A figure, with a plot created by either `plot()` or `fplot()` can have additional plots added to it by using:

```
>> hold on;
```

However, if a separate plot is desired later in the script (such as a new home work problem) then use either

```
>> hold off
```

OR

```
>> figure
```

Otherwise, any new `plot()` or `fplot()` commands will result in the plot being put onto an existing graph.

MORE ON HOMEWORK SUBMISSION:

Before submitting your work, execute the .m file by pressing the little green arrow at the top of your MATLAB session. The output of this execution is what I use to determine if the assignment was done correctly. If you get any error message (red text in your command window), this means that your .m file did not execute completely. It executed up to the point where an error message was generated and then aborted. You must fix the error then re-run. Continue this process until a you get a clean run. After a clean run THEN...

Publish your work by clicking on file then publish it. This will cause your .m file to execute again and create a file consisting of all your output, by default HTML file. Go in the publish options and switch to PDF. It is THIS OUTPUT (PDF file) that you are to submit. Any output that has errors (such as the above mentioned red text) will not be given full credit.

The `fprintf()` command

One convenient way to display all output is by using the `fprintf()` function call using words to explain (briefly) what the output depicts. Look at the modification (below) of our first iterative example (done on day 1) that approximates the golden ratio. At the end of the EXPLANATION below, there is a discussion about how `fprintf()` is used to express the output. It would be a good idea to peruse the rest of the explanation to understand why the modifications were added. Also, please suppress all output except for the expression of your answers (by ending every executable line of code with a semicolon).

The following is a modification to the iterative code we wrote during the first class to approximate the golden ratio. Please note the discussion about the purpose of the added code. Also, please make note of the usage and discussion of how the output was generated by using the fprintf() function call.

```
initx=10;
tolerance=10^-6;
x=initx;
iterNum = 0;
for k = 1:100
    iterNum = iterNum+1;
    newx = sqrt(1 + x);
    if (abs(newx - x) < tolerance)
        break
```

```
    end
    x = newx;
end

fprintf('the golden ratio found with init value of %11.2f is %7.5f \n which took %i itera
```

Produces the following output:

```
 the golden ratio found with init value of 10.00 is 1.61803
           and took 17 iterations
```

EXPLANATION:

First, the logic was modified so that the number of iterations that are ultimately performed is determined on the fly vs the hard coded limit of 30 we did in first Lab. Note that it is possible that this loop was allowed to execute 100 iterations but actually executed only 17 times as evidenced by the output. The number 100 was chosen arbitrarily - it just needed to be sufficiently large to handle any reasonable situation. Without a limit (which could be implemented by using a **while** loop construct instead of a **for** loop), then there would be a risk of an infinite loop if there was some logic error.
A variable called tolerance is given a value (`10^-6=0.000001`). The variable **iterNum** was also created and initialized to zero which gets incremented by one as many times as the loop is iterated. Then, a new x is calculated. If the absolute value of the difference between the new and old x is less than this tolerance, then the loop is terminated by the **break** command. The break statement transfers control to the first statement after the end statement which is the **fprintf()** function call.
These modifications show how a typical iteration sequence is done... the program decides while executing how many iterations are done as opposed to setting a predefined amount. This necessitates storing the initial x and new x as separate variables (otherwise, as before when x was simply reset to a new value of x, there would be no way to compare the new x with the old x). Additionally, **iterNum** incremented so that our answer can include how many iterations it took to converge to within our predetermined tolerance.
Please look carefully at the **fprintf()** function call. This is a far superior way to express answers compared to simply unsupressing a variable that holds the answer. This **fprintf()** can output a sentence with multiple variable values within. This usage creates a sentence with the values of 3 variables. Basically, what is inside the quotes gets reproduced EXCEPT for the following:
A percent sign and the few following characters are not outputted; **fprintf** interprets this symbol as expecting the characters following it to depict the desired output format. Here, there are 3 different format identifiers within the quoted string, each one associated with one of the 3 variables in the list after the closing of that quote. So what **fprintf** outputs is the corresponding variable's value in place of the characters

`%11.2f`

(associated with the first variable, initx). The **f** means that the output is to be in a floating point (decimal) format. The 11 means that **fprintf** is to allocate 11 spaces for this variable's value with 2 spaces to the right of the decimal point. The 2nd formatter is

`%7.5f`

which is associated with the 2nd variable in the list: **x** So this variable's value is allocated 7 spaces, 5 of which are to the right of the decimal. Similar for the 3rd formatter

`%i`

means format as an integer and 3rd variable. The

`\n (backslash n)`

which is read "escape n" is actually just one character and tells **fprintf** to return the cursor to the first column of the next line. Generally, you will want to include

`\n`

as the last character of the quoted string, else your command window will sometimes look quite messed up. Note that there are many other formatting options, described in your text as well as MATLAB's help. I encourage you to use **fprintf()** to express your answers. You may have to experiment a little but it's fairly easy after using it a bit.