# MATH 3670 - SCIENTIFIC COMPUTATION I
# Fall 2015

---

## Week 4: Curve Fitting and Interpolation in MATLAB (Chapter 8). Applications in Numerical Analysis (Chapter 9)

---

### Content

Part I
- Linear Interpolation
- Spline Interpolation
- Polynomial Fitting
- Exponential Fitting
- Review of function calls
- Review of anonymous function usage

Part II
- Solving equations (polynomial, non-polynomial) with one variable
- Find a function's maximum and minimum
- Usage of plots to help guide function solving

• **Homework # 4** (Due Fri, Sept 25):
Chapter 8, pg 290 # 22, **26***, 27
Chapter 9, page 314–315 # 8, 9

---

Today, we will introduce some powerful techniques for studying discrete data. Quite often, one performs a series of observations (such as counting bacterial growth each day) which is an example of discreet data. Sometimes there are pre conceived notions about how the observed phenomena is expected to behave. For example, population growth often occurs exponentially (only for a while). One can then see how well a specific experiment's data is consistent with conventional ideas about the phenomena. Other times, little is known about the intrinsic properties of the phenomena. Data fitting and interpolation are tools that aid in this analysis.

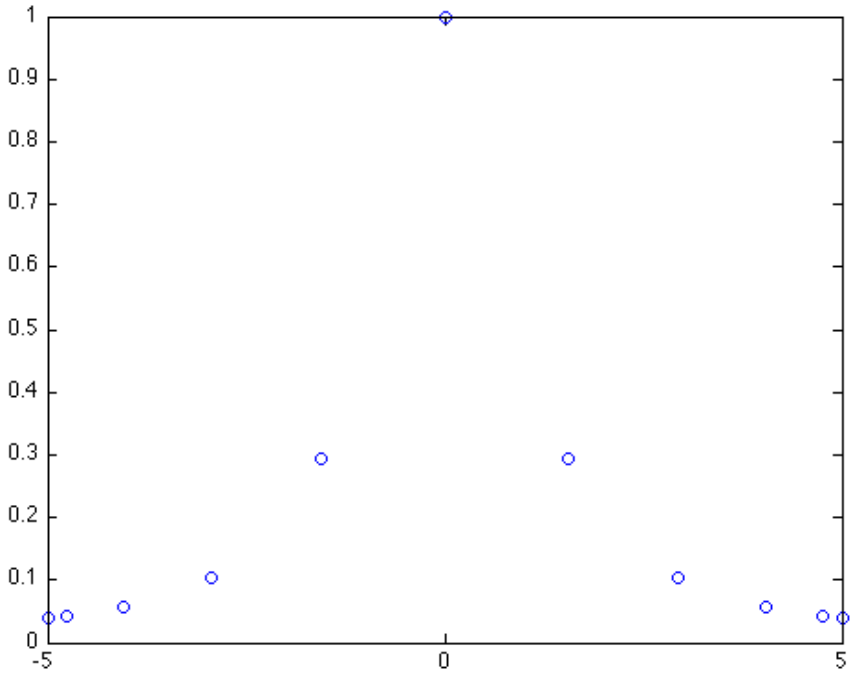
Interpolation and fitting are closely related. However, there are important differences that influence how and when they are used. Today, we will focus on how to use interpolation and fitting with discrete data. Roughly, interpolation is an effort to find reasonable data values between empirically derived discrete data points. Fitting is an effort to find some sort of fit, typically exponential or polynomial that attempts to describe the observed discrete points.

# 1. Interpolation

The simplest interpolation strategy is simply to do the equivalent of drawing a straight line between each point. This is called linear interpolation. For example:

```
>> n = 11;
>> jdata = linspace(0, pi, n);
>> xdata = 5*cos(jdata);
>> f = @(t) 1./(1+t.^2);
>> ydata = feval(f, xdata);
>> plot (xdata, ydata, 'o');
```

Review the above commands:
**jdata** = vector of length 11 equally spaced points between 0 and *pi*
**xdata** = vector of length 11 with values spread between -5 and 5 via a Chebyshev spacing (the x-axis values of 11 evenly spaced points around a circle)
**f** = anonymous function that takes a vector **t** as input (NOTE the **./** for element by element matrix division)
**ydata** = vector containing the evaluation of function **f** at each **xdata** point.
This plot creates:

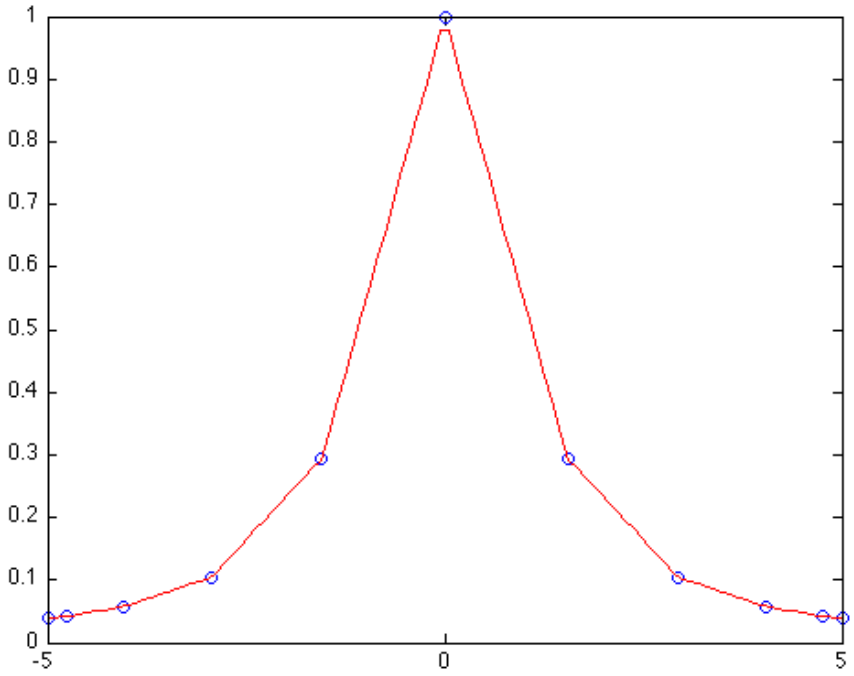



Now we will use Matlab's **interp1** function to linearly interpolate between these points:

```
>> interpx = linspace(-5, 5, 100);
>> popLinear = interp1 (xdata, ydata, interpx, 'linear');
>> hold on;
>> plot (interpx, popLinear, 'r');
```

These commands perform:
**interpx** = vector of length 100 with values between -5 and 5
**popLinear** = a vector which is the output of Matlab's **interp1** function. Notice the inputs:
**xdata, ydata** represent the discrete data
**interpx** is the set of xvalues that the **interp1** function uses to find the associated y-values which is the function's output.
**'linear'** is the property (notice it is in literal form) that tells this function that a linear interpolation is to be performed.

Now, the plot is:

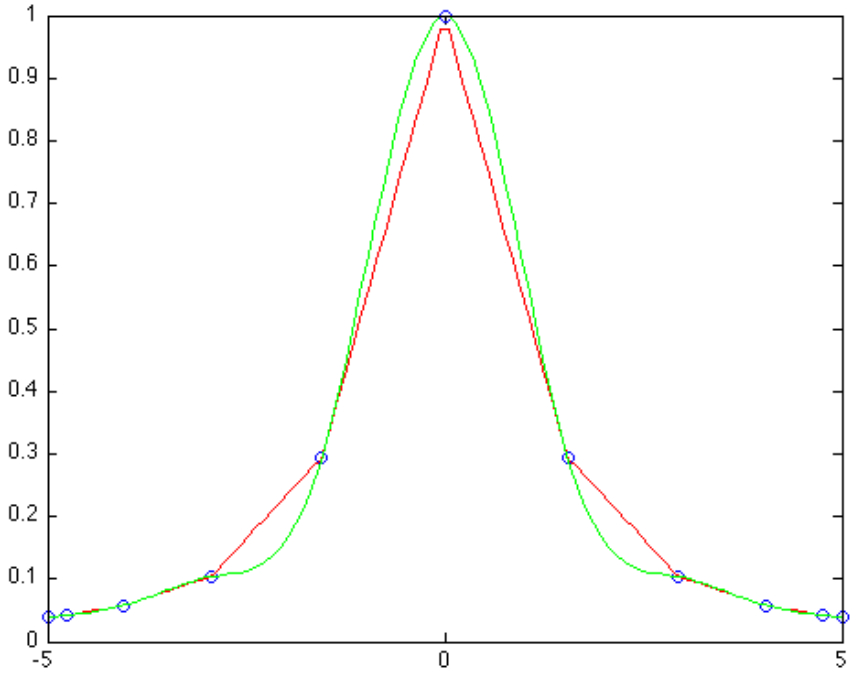



Notice that this looks like line segments were drawn between each discrete data point which is basically what linear interpolation is.

Now, we will add a spline interpolation:

```
>> popSpline = interp1 (xdata, ydata, interpx, 'spline');
>> plot (interpx, popSpline, 'g');
```

Notice that the keyword **'spline'** (in quotes) was used so that the **interp1** function performed a spline interpolation. Again, this function's output (which I named **popSpline**) consists of a vector of same length as **interpx** - the 100 x-values and contains the spline function's image values for each of these x-values.
The plot:




Why spline vs. linear?
A typical spline interpolation will fit a 3rd degree polynomial between each successive set of 4 points. Sometimes this "looks" better, sometimes not. One must be careful to include what might be known about the phenomena before blindly using the "superior" spline interpolation instead of a linear one. For example, maybe the experimentalist knows that the data, if anything, should go on the other side of the linear fit in those places where the green spline line is placed.

One nice thing about the spline is that the results have a smoothly changing first derivative which is not the case of a linear fit. This is often advantageous when using the interpolated data as inputs to other applications.

# 2. Curve Fitting

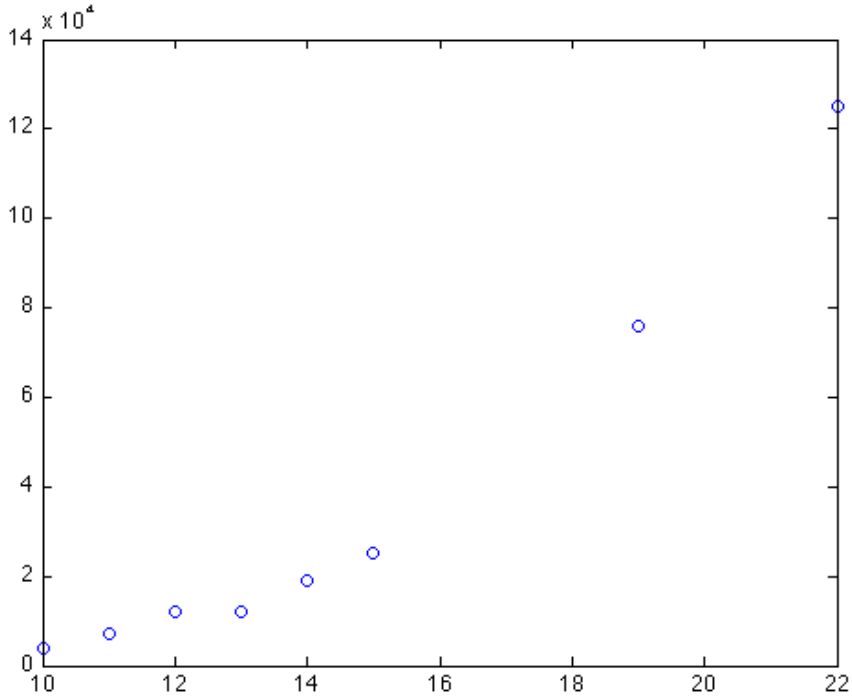
Curve fitting involves the attempt to find some sort of function that roughly mirrors (describes) the discrete data. For example, if a temperature reading at a single place was done every hour for months and then these points were graphed, there would be, at least in a rough sense, a rise and fall in values based on a 24-hour cycle. There would be many exceptions, but over the long term, a basic trend would be seen. This cyclic nature might suggest some sort of trigonometric function could describe this behavior, at least roughly. If the data sharply deviates from the expected behavior, it could be that temperature at that location behaves differently than what was expected. Alternatively, it could be due to some bias associated with that particular location, for example, if the thermometer was heavily shaded until late afternoon. Curve fitting is a form of looking at the data and seeing if it is

consistent with expectations, or, if the fitting attempts provide more understanding of the mechanisms that influence the data.

Today, we will look at fitting discrete data exponentially and polynomially. Our example will involve population counts of bacteria in a medium over time. Population growth is often exponential in nature, at least for a short time and given certain assumptions such as no restrictions on resources and no external input or shock to the system. Our example starts with 8 days (some days are skipped) of counting bacteria colonies in our growth medium, once per day at the same time each day:

```
>> figure;
>> dataDays       = [10 11 12 13 14 15 19 22];
>> bacterialCount = [4  7  12 12 19 25 76 125]*10^3;
>> plot (dataDays, bacterialCount, 'o');
```

**dataDays** represents the day number in which the observation was made
**bacterialCount** represents the number of colonies that were observed on each of the above days. Notice that each value is multiplied by a factor of 1000.
Note that while the time spans from day number 10 to 22, the bacterial count increases from 4 to 125 (times 1000).
Also note that most of the gains in bacterial count occur near the end of this time period. Such large changes near one end of sampling often (not necessarily) is indicative of an exponential nature of change.
The plot is:




Notice the scale of the y-axis.

Now let's find a "best" exponential fit. Remember, the basic form of an exponential function based on some variable has an exponent which is linear:

```
base^(ax+b)
```

Notice the exponent **ax+b** is the basic form of a linear equation. Usually, the **base** used is the natural logarithm **e**.

We will use Matlab's **polyfit** function to attempt to find a the best coefficients of the above equation for **a** and **b**:

```
>> p = polyfit(dataDays, log(bacterialCount), 1);
```

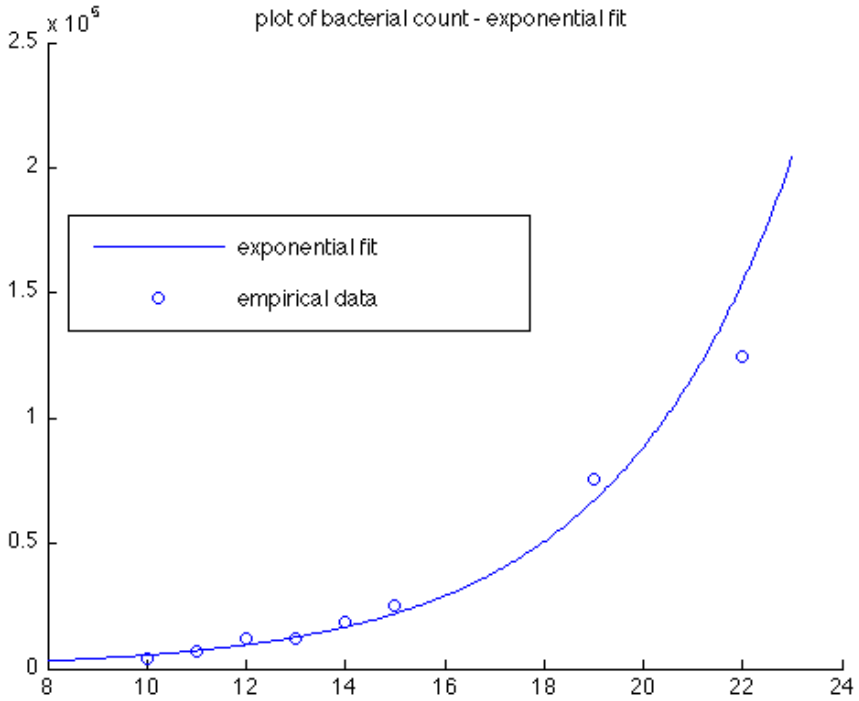
The inputs to this particular use of **polyfit** are: **dataDays** - the discrete data's x-values
**log(bacterialCount):** This "parameter" is actually a call to Matlab's **log** function to take the natural logarithm of the **bacterialCount** set of values. This is a good example of a parameter of one function is actually the output of yet another function - a usage of a **function function**.
The usage of the **log** function as a parameter to **polyfit** tells **polyfit** to fit an exponential function instead of a polynomial function.
The last parameter **1** is the degree of polynomial (in this case the degree of the exponent's polynomial) to be created.
The output is a 2 element vector which I called **p**. **p(1)** represents **a** and **p(2)** represents **b** in the above exponential equation. So when one uses this output appropriately, the output of this usage of **polyfit** is the exponential equation:

```
e^( p(1)x + p(2) )
```

Now let's create a denser array of time (x-values) and also an array of y-values representing the above exponential equation and plot this against our empirically derived data:

```
>> timeVals = linspace (8, 23, 360);
>> expFit = exp(p(2)) .* exp(p(1).*timeVals);
>> hold on
>> plot(timeVals, expFit);
```

The vector **timeVals** has 360 equally spaced elements from 8 to 23 which roughly represents each hour between day 8 and day 23.
The vector **expFit** contains the output of the **inline** function defined on this line.
Notice the usage of the vector **p** which is the output of the above **polyfit** function. Also notice that the **timeVals** vector was used as the x-values of this function - another usage of element by element vector multiplication.
The plot of this exponential fit is placed on the original plot of the discrete data:




Notice that the fitted curve does not exactly match the data values. But it is likely (if **polyfit** worked properly), this is the best possible exponential fit.
Now, we can use this fit to determine what the bacterial count would have been on a day not represented by our original data. For practice, we will use an anonymous function to compute this as opposed to the above inline function usage:

```
>> a = p(1);
>> b = p(2);
>> expFitFn = @(x, a, b) exp(a*x + b);
>> day25 = expFitFn(25, a, b);
>> fprintf ('According to this exponential fit, the bacterial count on day\n');
>> fprintf ('25 was %7.0f\n', day25);
```

The output of the **fprintf** statements are:

```
According to this exponential fit, the bacterial count on day
25 was  356407
```

IMPORTANT: This number is how many bacteria there would have been on day 25 IF THIS FUNCTION ACCURATELY DESCRIBED THIS EXPERIMENT'S BEHAVIOR.
If, on day 24, there was a fire or if a lot of new bacteria were introduced or any other reason why things changed significantly, the actual number would be much different.
ALSO, it is possible that our preconceived notion that this particular exponential function (or any exponential function) being a good descriptor of the growth phenomena is fundamentally flawed. The mathematics is correct; the assumption that this particular

mathematical model is a good description of our phenomena may be incorrect.

Now let's fit this same data with a 3rd degree polynomial. Again, we will use Matlab's **polyfit** function:

```
>> figure;
>> plot (dataDays, bacterialCount, 'o');
>> hold on
>> plot(timeVals, expFit);
>> p = polyfit(dataDays, bacterialCount, 3);

>> polyFit = p(1).*timeVals.^3 +...
>>                 p(2).*timeVals.^2 +...
>>                 p(3).*timeVals + p(4);

>> plot(timeVals, polyFit, 'r');
>> title('plot of bacterial count - exp, poly fits');
>> leg = legend('selected day''s data','exponential fit','3rd degree polyfit');
>> rect = [0.15, 0.55, .45, .15];
>> set(leg, 'Position', rect);
>> xlabel (' Day number');
```

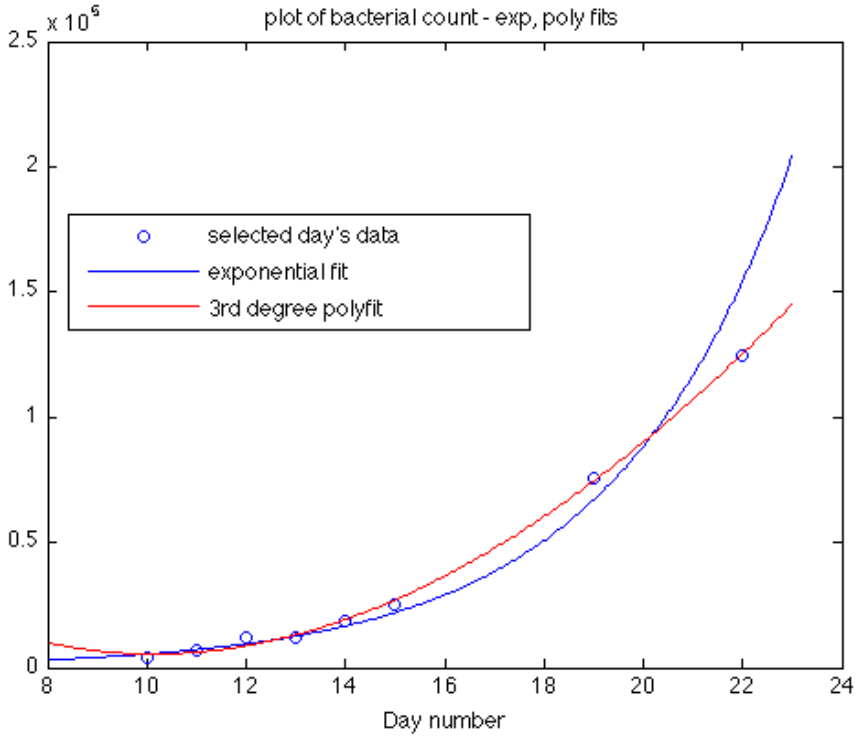
Here, the **polyfit** function is called using the same **dataDays** vector (the original day number each observation was made). But the **bacterialCount** vector is used as is (as opposed to calling the **log** function as part of the **polyfit** parameter). This tells **polyfit** to create a polynomial (instead of an exponential) fit. The last parameter **3** tells the function to create a 3rd degree polynomial fit.
NOTE: That a 3rd degree polynomial fit is different than the cubic spline we did above - both use 3rd degree polynomials but different definitions for fitting. (Basically a 3rd degree fit finds the best 3rd degree polynomial fit for all N data values vs. cubic spline uses a set of 4 points to fit a 3rd degree polynomial then, the vector is incremented by one so the next set of 4 points which includes 3 of the previous set of 4 points are used for the next 3rd degree polynomial. So there are N - 3 different 3rd degree polynomials used to fit the data.)

The output of this call to **polyfit** is a 4 element vector that I called **p**. There are 4 elements because it represents the coefficients of the 3rd degree polynomial it found. To interpret this vector appropriately, use as:

```
p(1)*x^3 + p(2)*x^2 + p(3)*x + p(4)
```

The next line produces a vector called **polyFit** which is the same length as the vector **timeVals**. Each element of **timeVals** is used as an x-value to a 3rd degree polynomial using the 4 elements of **polyfit** output as it's coefficients.

The above code produces:




At first glance, the polynomial fit (red line) looks a little closer than the exponential fit (blue line) to the original data (blue circles). But upon closer inspection, the polynomial fit implies that before day 10, the bacterial count was higher. Even though this is possible (the experimenter may have killed some before day 10), there is no way for any fitting scheme to "know" this. If the days prior to day 10 were unremarkable, then each successive day would have more bacteria than the day before.

IMPORTANT LESSON: even though one can eventually find a polynomial that fits a given set of data better than an exponential fit, this does not necessarily mean that it is better a better description of the phenomena. This is especially evident in a case when shortly before the first observation was made, bacteria counts should be increasing all the time, even if slowly.

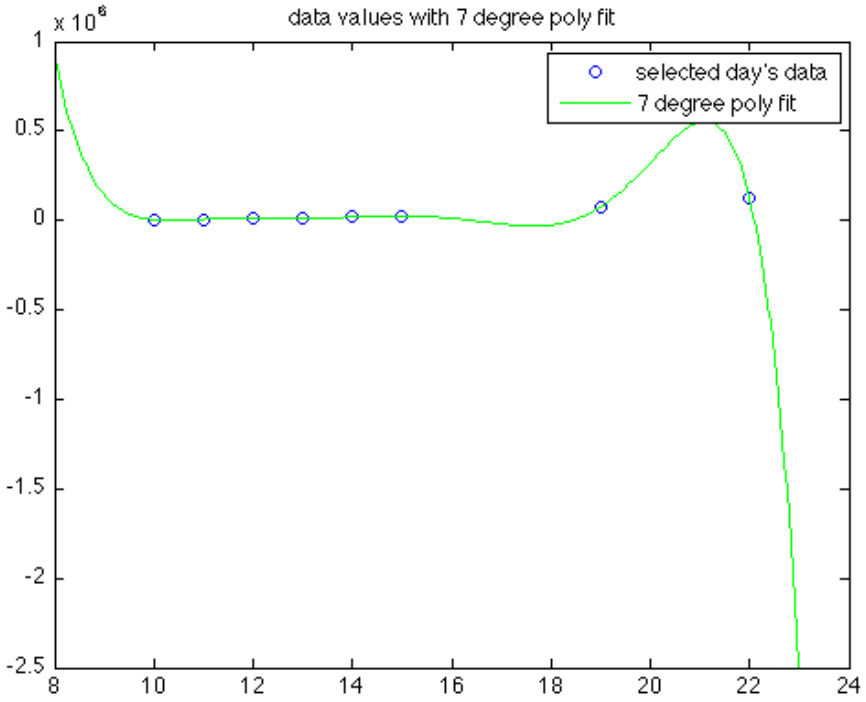
In fact, there is a mathematical theorem that states that for n-data points, there exists a polynomial of degree n that will exactly "fit" the data. But this polynomial might also behave very different for values in between each data point than one would expect the actual data would be if observed during those in between time spots.

The following fits a 7 degree polynomial to the original data and creates a new figure with a graph of this 7 degree polynomial superimposed on the original data:

```
>> figure
>> plot (dataDays, bacterialCount, 'o');
>> p = polyfit(dataDays, bacterialCount, 7);

>> polyFit = p(1).*timeVals.^7 +...
```

```
>>           p(2).*timeVals.^6 +...
>>           p(3).*timeVals.^5 +...
>>           p(4).*timeVals.^4 +...
>>           p(5).*timeVals.^3 +...
>>           p(6).*timeVals.^2 +...
>>           p(7).*timeVals + p(8);
>> hold on
>> plot(timeVals, polyFit, 'g');
>> leg = legend('selected day''s data','7 degree poly fit');
>> title (' data values with 7 degree poly fit');
```

Notice that the fit goes exactly through each data point - in a way, an "exact fit" to the original data. However, one would be hard pressed to defend this polynomial as a good representation of the behavior of this experiment's bacterial growth. Indeed, the observed count on day 22 was 125000. This "perfect fit" then predicts that on day 23, there would be minus 2.5 million bacteria.




# 3. Fun With Curve Fitting

(and a powerful way to improve a fit without increasing sampling)

We will now attempt to "harvest" data points from an image. This will also illustrate some of Matlab's input capabilities. We will attempt to click a number of points on the profile of Mt. Mckinley (do at least 21 of them, more is better), then use **polyfit** to create a polynomial of degree 21 that fits the points that were obtained by the user's cursor position on each click.

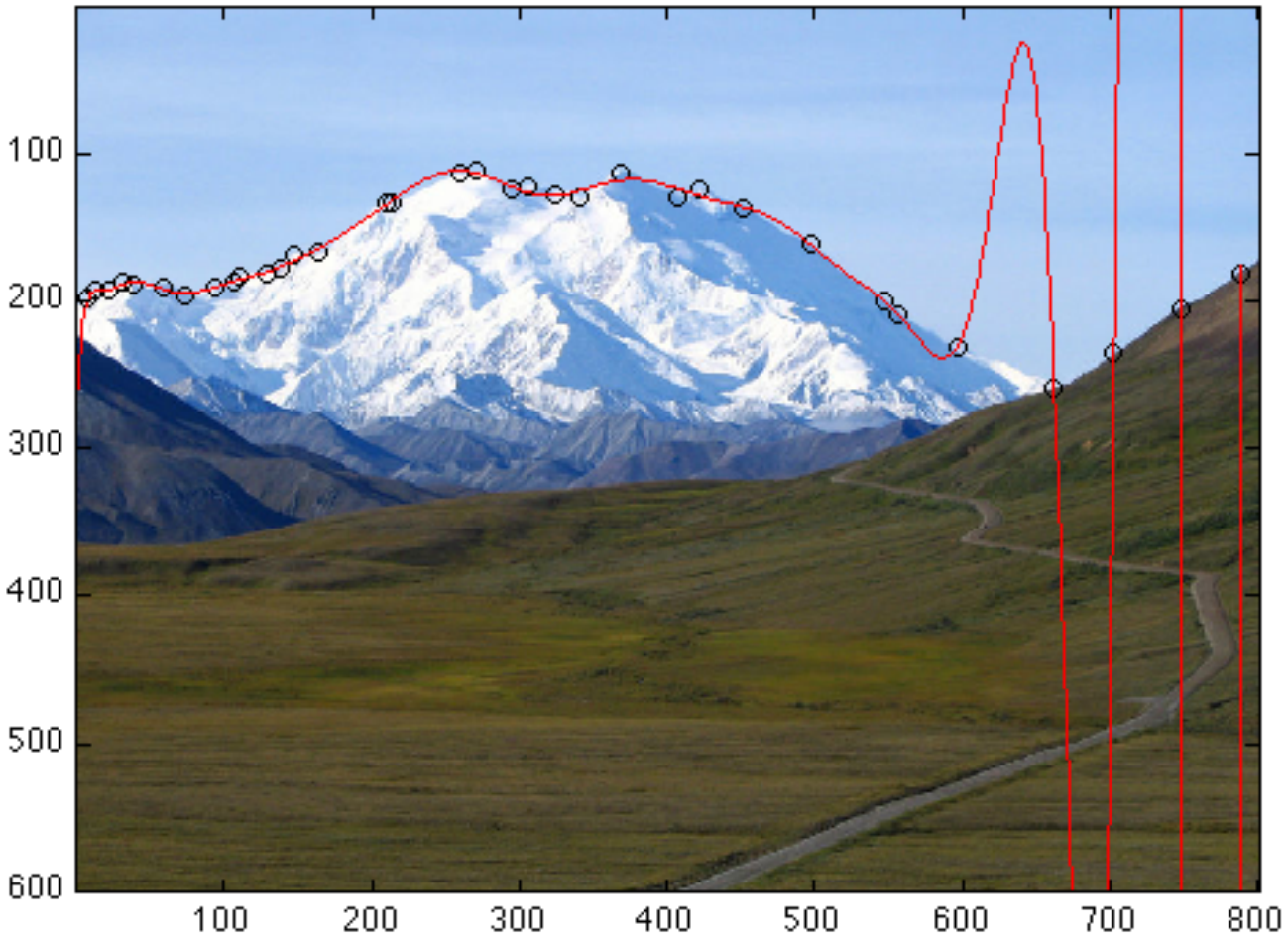
To do this:
- You need to have **mountain.jpg** in your working Matlab path (i.e. Matlab working directory).
- When the picture of the mountain appears, better results are obtained if the image is expanded by grabbing the lower right corner with the cursor and dragging it down and to the right.
- Click at least 25 points along the profile of the mountain. When finished, hit the enter key. Then click in Matlab's command window and hit any key (like space) to "un pause" the script execution.
- A red line should then appear on the image that goes through each clicked point (seen as circles). - More points = better results (usually). If you continue clicking up along the profile of the closer hillside on the right side of the image, the polynomial fit can get pretty crazy.

```
>> A=imread('mountain.jpg');
>> image(A), axis image, hold on
>> [X,Y]=ginput;
>> plot(X,Y,'o')
>> save('mount.mat','X','Y')
>> load('mount.mat')
>> fprintf('Number of points sampled (# of mouse clicks): %i\n', numel(X));
>> N=size(X); N=N(1);
>> plot(X,Y,'ok'), hold on
>> pause

>> Pfit=polyfit(X,Y,21);
>> xplot=linspace(0,max(X),1000);
>> yplot=polyval(Pfit, xplot);
>> plot(xplot, yplot, '-r');
>> hold off
```

The **fprintf** displays the number of points that were clicked:

```
Number of points sampled (# of mouse clicks): 34
```

Here is an image of an attempt I did at home:



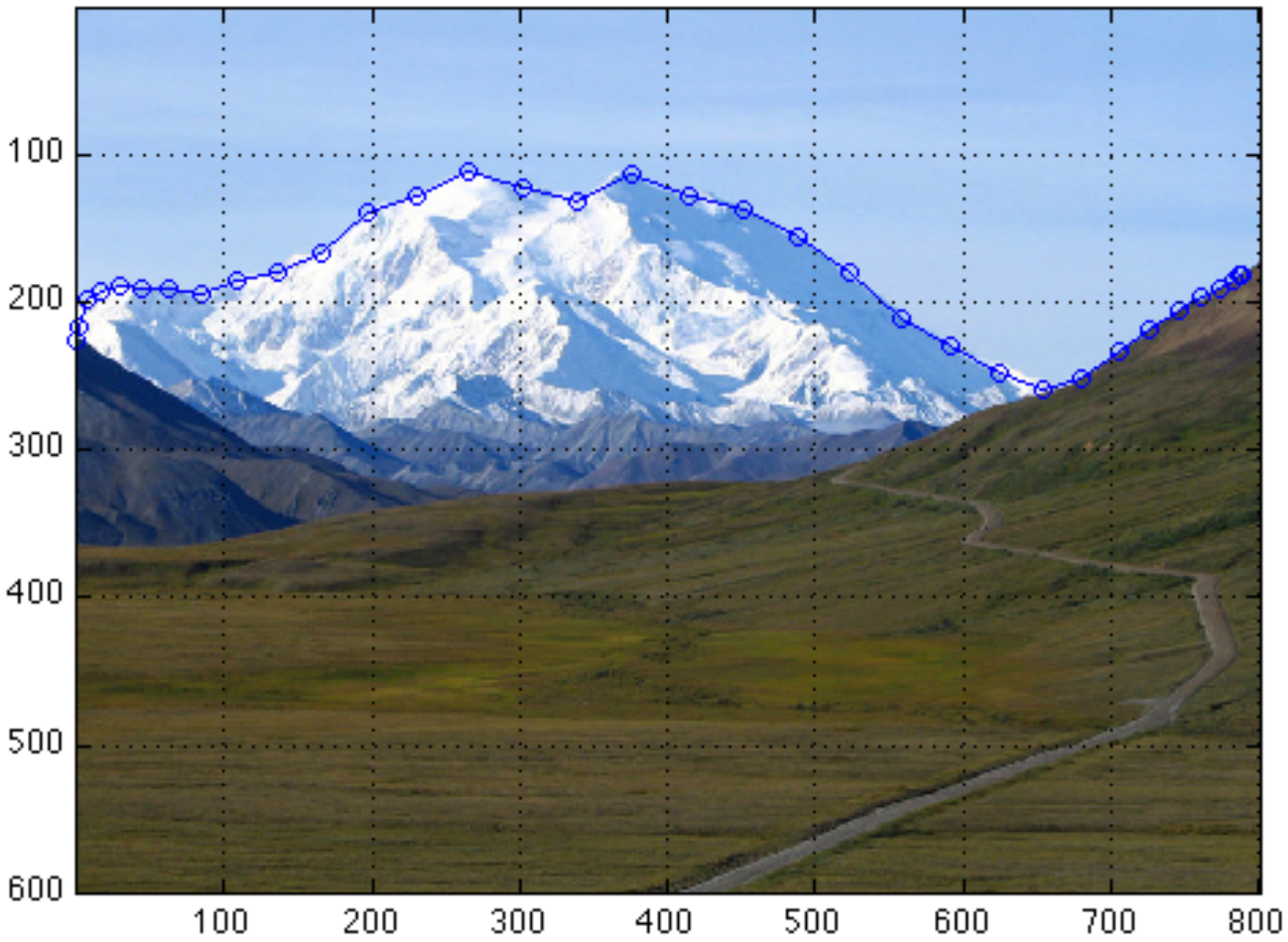

Now for a VERY GOOD way to greatly improve the fit WITHOUT increasing the number of points to sample:
This strategy basically involves sampling more often at the edges and where there is rapidly changing slopes while keeping the overall number of sampled points constant - i.e. if one is only allowed to use 34 points, this technique allows for spacing such that results are greatly improved.
Briefly, the previous attempt involved fitting a polynomial of degree 21 to the (in my case) 34 points sampled.
Now, the x-values of these same points will be modified via a Ç hebyshev spacing technique which can be graphically described as:
Imagine a circle where (in my case) 34 points are equally spaced around it. Then project these points down to the x-axis - these become the new x-values (after rescaling to fit the original picture).
Then the original (34) y-values are then modified via the spline method of **interp1**.

The below code snippet performs this and produces a new image of the fit. Notice that there are still only 34 sampled positions, however the fit is greatly improved:

```
figure
image(A), axis image, grid on, hold on
x=X/max(X);
j=linspace(0,pi,numel(X));
xi=cos(j);
yi=interp1(x,Y,(xi+1)/2,'spline');
fprintf('Number of points Chebyshev spaced points: %i\n', numel(xi));
plot((xi+1)/2*max(X),yi,'-ob')
```

```
Number of points Chebyshev spaced points: 34
```

Here is an image of the improved fit (without increasing number of points):



# Part II. Applications in Numerical Analysis

Numerical analysis is the field of mathematics that studies the effective use of computer algorithms for solving scientific problems. While this is a vast subject (which requires several courses just to get started in it), in the following two labs we highlight some of the applications of MATLAB, such as solving equations (finding zeros of functions), finding minima or maxima of a function, numerical integration, and solving ordinary differential equations.

---

# 4. Root finding

A central problem in numerical analysis is formulated as follows. Given a real-valued function $f : \mathbb{R} \to \mathbb{R}$, find a number $r$ so that $f(r) = 0$. The number $r$ is called the root or zero of $f$. We have already seen two ways to compute roots of polynomials and of symbolic functions. Let's briefly recall them.

## 4.1 Computing roots of polynomials

As we have already seen, polynomials are represented in MATLAB by their coefficients in the descending order of powers. For instance, the cubic polynomial $p(x) = 3x^3 + 2x^2 - 1$ is represented as

```
>> p = [3 2 0 1];
```

Its roots can be found using function roots

```
>> r = roots(p)
```

```
r= -1.0000
     0.1667 + 0.5528i
     0.1667 - 0.5528i
```

To check correctness of this result we evaluate $p(x)$ at $r$ using the function `polyval`.

```
>> err = polyval(p, r)
```

```
err = 1.0e-014 * - 0.6661
      - 0.0444 - 0.0722i
      - 0.0444 + 0.0722i
```

The above line of code called Matlab's **polyval** function which takes 2 parameters, the first being a vector of polynomial coefficients; the 2nd being a vector of values to evaluate. This vector of values that we use here are values returned by the **root** function that corresponds to the zeros of our polynomial function represented by the vector **p**. So the vector returned

by **polyval** which I chose to name **err** represents the polynomial evaluation for each element member of the vector **r**. If these 2 Matlab functions worked correctly, then the members of the vector **err** should be zero. Since the values of **err** are zero or close to the computer's **eps** value, we infer that the zeros returned by the **root** function call are indeed zeros of the polynomial.

To reconstruct a polynomial from its roots one can use the function `poly`. Using the roots $r$ computed earlier we obtain

```
>> poly(r)


ans = 1.0000    0.6667   -0.0000    0.3333
```

Note that these are the coefficients of $p(x)$ all divided by 3. The coefficients of $p(x)$ can be recovered easily

```
>> 3*ans


ans = 3.0000    2.0000   -0.0000    1.0000
```

## 4.2 Computing zeros of symbolic functions

A quick (but not always successful way) to evaluate zeros of functions is using the symbolic capabilities of MATLAB. For example, if we want to compute the roots of the same polynomial $p(x) = 3x^3 + 2x^2 - 1$, we can proceed as follows:

```
>> syms x
>> p=3*x^3 + 2*x^2- 1
>> r=solve(p);
```

We do not include the output of this operation since it would not fit on this handout. Typical of symbolic calculations, the zeros are displayed as very long (100 characters or so) of complex arithmetic. If you use the `double` command, then you can see the same roots in numeric format. But this procedure does have it's disadvantages, since not all equations can be solved symbolically, as the next example will show.

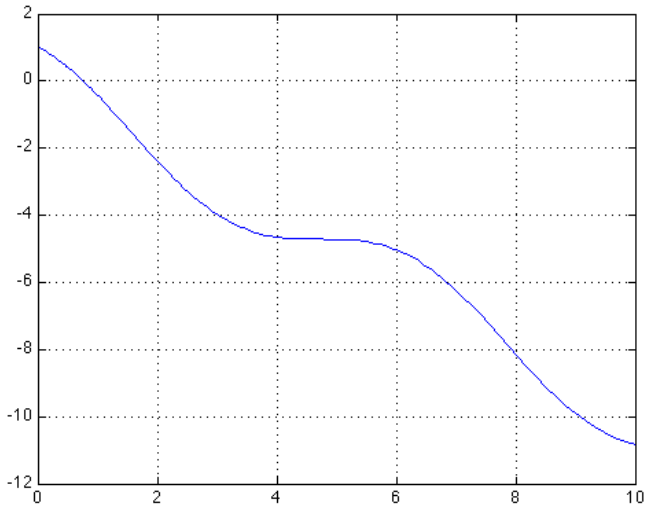
## 4.3 Computing zeros of any function

The MATLAB function `fzero` computes a zero of the function $f$ using user supplied initial guess of a zero sought. For example let $f(x) = \cos(x) - x$. To see how `fzero`works, let's define the function in anonymous format

```
>> f = @ (x) cos(x)-x ;
```

The graph below is obtained with the `fplot` command

```
>> fplot(f, [0 10])
```




To compute its zero we use the MATLAB function `fzero`:

```
>> format long
>> r = fzero(f, 0.5);


r =
   0.739085133215161
```

The first parameter of the function `fzero` is the name of the function whose zero is to be computed. The second argument of `fzero` is the initial approximation of $r$. One can check the last result using the function `feval`:

```
>> err = f(r)


err =
         0
```

In the case when a zero of a function is known to be within an interval, one can enter a two-element vector that designates this interval. In our example we choose $[0, 1]$ as this interval to obtain

```
>> r = fzero(f, [0 1])


r=
     0.739085133215161
```

However, this choice of the designated interval

```
>> fzero(f, [1 2])
```

generates the error message.

```
???? Error using ==> fzero The function values at the interval endpoints
must differ in sign.
```

The above error implies that **fzero** uses an aspect of the Mean Value Theorem: a zero can be numerically approximated if an interval is given such that the function evaluation at the extremes of that interval have opposite signs. If these evaluations are of opposite signs, a narrower interval can be constructed with this same behavior. This process can be continued until the interval is within some specified tolerance.

The function `fzero` can take several optional parameters (for a complete list, see the help for `fzero`.) below we illustrate one of them, where all the iterations are displayed.

```
>> options=optimset('Display','iter');
>> r=fzero(f, 0.5, options)


Search for an interval around 0.5 containing a sign change:
 Func-count    a          f(a)             b          f(b)        Procedure
     1             0.5      0.377583           0.5      0.377583   initial interval
     3        0.485858      0.398417      0.514142      0.356573   search
     5            0.48      0.406995          0.52      0.347819   search
     7        0.471716      0.419074      0.528284      0.335389   search
     9            0.46      0.436052          0.54      0.317709   search
    11        0.443431      0.459853      0.556569      0.292504   search
    13            0.42      0.493089          0.58      0.256463   search
    15        0.386863      0.539234      0.613137       0.20471   search
    17            0.34      0.602755          0.66      0.129992   search
    19        0.273726      0.689045      0.726274     0.0213797   search
    21            0.18      0.803844          0.82     -0.137779   search

Search for a zero in the interval [0.18, 0.82]:
 Func-count    x          f(x)             Procedure
    21            0.82     -0.137779        initial
    22        0.726355     0.0212455        interpolation
    23        0.738866    0.00036719        interpolation
    24        0.739085  -6.04288e-08        interpolation
    25        0.739085   2.92788e-12        interpolation
    26        0.739085             0        interpolation

Zero found in the interval [0.18, 0.82]

r =
   0.739085133215161
```

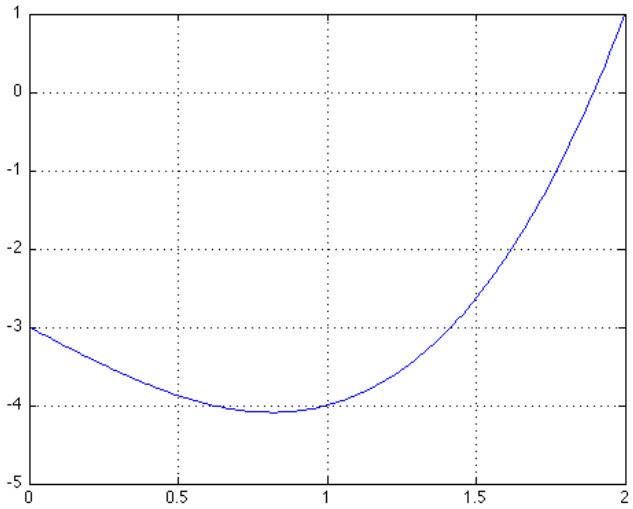
# 5. Finding a minimum or maximum of a function

A very common task in mathematics is to determine the minima and maxima of a function (either of one variable or several variables). MATLAB has a few powerful ways to do this without having to go through the derivative (or gradient) computations.

## 5.1 The `fminbnd` command

Finds minimum of single-variable function on fixed interval. Let's consider the following function.

```
>> f = @(x) x^3-2*x-3;
>> fplot(f, [0 2])
```




To compute the x-value corresponding to the minimum point of $f$ in the interval $[0, 2]$, invoke `fminbnd` with

```
xmin = fminbnd(f, 0, 2)
```

The result is

```
xmin =
    0.8165
```

which is the x-value corresponding to the minimum point and this minimum value is

```
fminval = f(xmin)
```

The result is

```
fminval =
   -4.0887
```

One can use this command also to find maximum of a function in a single interval.

## 5.2 The `fminsearch` command

Finds minimum of unconstrained multivariable function using derivative-free method. Again, we illustrate this on the following function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

which is called the Rosenbrock banana function.

```
>> banana = @(x) 100*(x(2)-x(1)^2)^2+(1-x(1))^2;
```

Pass the function handle to `fminsearch`:

```
[x,fval] = fminsearch(banana,[-1.2, 1])
```

This produces

```
x =
    1.0000    1.0000
fval =
    8.1777e-01
```

What's remarkable about `fminsearch` is that it uses an algorithm that does not require the computation of the derivative, in this case the Nelder – Mead method ( also known as downhill simplex method or amoeba method), which is very effective even in large dimensions.

A very interesting use of the `fminsearch` command is in finding solutions of any system of equations. To illustrate the method, consider a $2 \times 2$ system of the form

$$\begin{cases} f(x_1, x_2) = 0 \\ g(x_1, x_2) = 0 \end{cases}$$

Note that any solution of this system is a minimizer for

$$\min_{x_1, x_2} h(x_1, x_2) \text{ where } h(x_1, x_2) = f(x_1, x_2)^2 + g(x_1, x_2)^2$$

in other words we can replace solving a system of (possibly very) nonlinear equations with several variables into a minimization problem, for which the Nelder-Mead algorithm can be effective in finding a solution:

**Example**: Find the points of intersection between the implicit curve $(x_1^2+x_2^2-1)^3-x_1^2x_2^3 = 0$ and the line $x_1 + 2x_2 = 1$
We employ the `fminsearch` command with the initial guess [1,1]

```
>> h = @(x)  ((x(1)^2+x(2)^2-1)^3-x(1)^2*x(2)^3)^2+(x(1)+2*x(2)-1)^2;
>> fminsearch(h, [1,1])
```

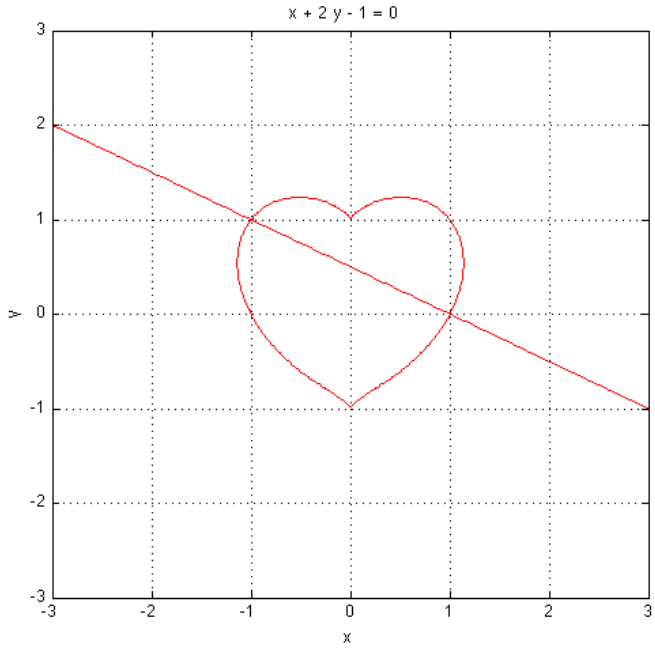
```
ans =
       1.0004   -0.0002
```

Note that the actual solution is (1, 0). However, by default, the `fminsearch` stops after a certain number of iterations. To get a better approximation, we can change the desired tolerance:

```
>> options=optimset('TolX', 10^(-10));
>> fminsearch(h, [1,1], options)


ans =
  1.0000   -0.0000
```

Is this the only solution? Numerically, it is impossible to tell how many solutions a nonlinear system will have, or how many local minimum points a function of two or more variables will have. Even mathematically this is a difficult problem to solve. In this case, we can get an idea of how many intersection points these two curves will have by plotting them (symbolically):

```
>> syms x1 x2
>> f = (x1^2+x2^2-1)^3-x1^2x2^3;
>> g = x1+2*x2-1;
>> ezplot(f), hold on, ezplot(g)
>> axis([-2 2 -2 2])
```




)

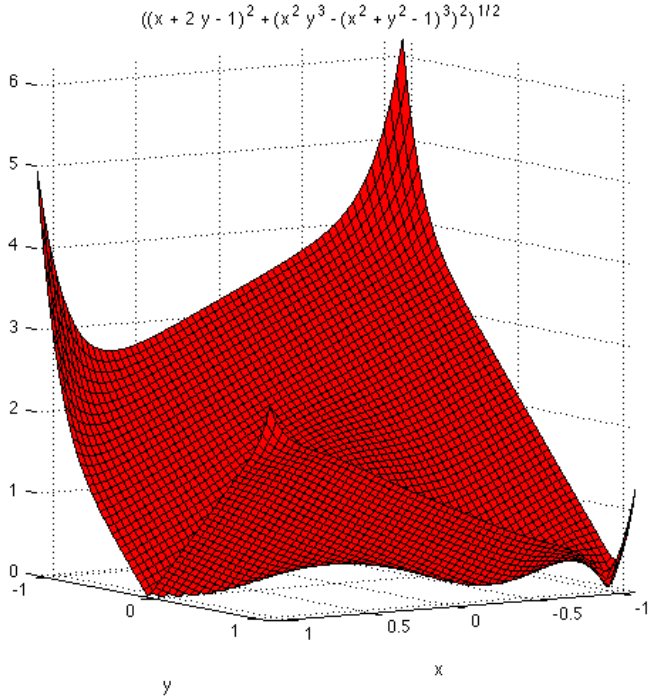
So obviously the only two solutions are $(x_1, x_2) = (1, 0)$ found above and $(x_1, x_2) = (-1, 1)$. How do we find the second solution? Simply choose a different guess point:

```
>> fminsearch(h, [-1,-1])


ans =
   -1.0000    1.0000
```

Note that you can also get extraneous solutions by this method, such as if you choose the initial guess to be $[0, 0]$:

```
>> fminsearch(h, [0,0])


ans =
   -0.4238    0.7151
```

This is because the function of two variables that gets minimized has more than two local minima, as seen from the graph below (more about plotting in 3D in Lab 10).




One could have employed the symbolic solver `solve` for seeking solutions of the system above, which would give the following results:

```
>> [x1star, x2star] = solve(f,g)


x1star =
                              1
                             -1
 - 54/125 + (3*51^(1/2)*i)/125
 - 54/125 - (3*51^(1/2)*i)/125
```

```
x2star =
                              0
                              1
 179/250 - (3*51^(1/2)*i)/250
 179/250 + (3*51^(1/2)*i)/250
```

which confirms again that there are two (real) intersection points, but the methods used in symbolic computations are quite different than those used in numerical methods.

---

## SUMMARY: Things to remember from this lab

Solving algebraic equations can be done efficiently using the `fzero` command, which employs a cocktail of numerical methods in finding zeros of a function. This only works for single equations, and it is in addition to the other methods (the symbolic `solve` or the the numerical `roots` functions).

Finding minima (or maxima) for a function there is the `fminbnd` , which also works only for functions of one variable. For functions of several variables, the `fminsearch` is effective since it employs a derivative-free method.

For solving systems of (nonlinear) equations there is no build in command in MATLAB. Algorithms such as the Newton's method, or steepest descent method can easily implemented (although they were not shown in this lab). More creatively, the `fminsearch` command can be used, although one has to be cautious to exclude the extraneous solutions (i.e. local minima that are not intended to be used as minima) that might be added by the equivalent minimization problem.

---