# MATH 2650 - INTRO TO COMPUTATIONAL MATH
Fall 2012

---

## Lab 7. Curve Fitting and Interpolation in MATLAB (Chapter 8)

---

### Content
- Linear Interpolation
- Spline Interpolation
- Polynomial Fitting
- Exponential Fitting
- Review of function calls
- Review of anonymous function usage

• **Homework # 7**: Chapter 8, pg 290 # 20, 21 (Due Fri, Oct 12)

---

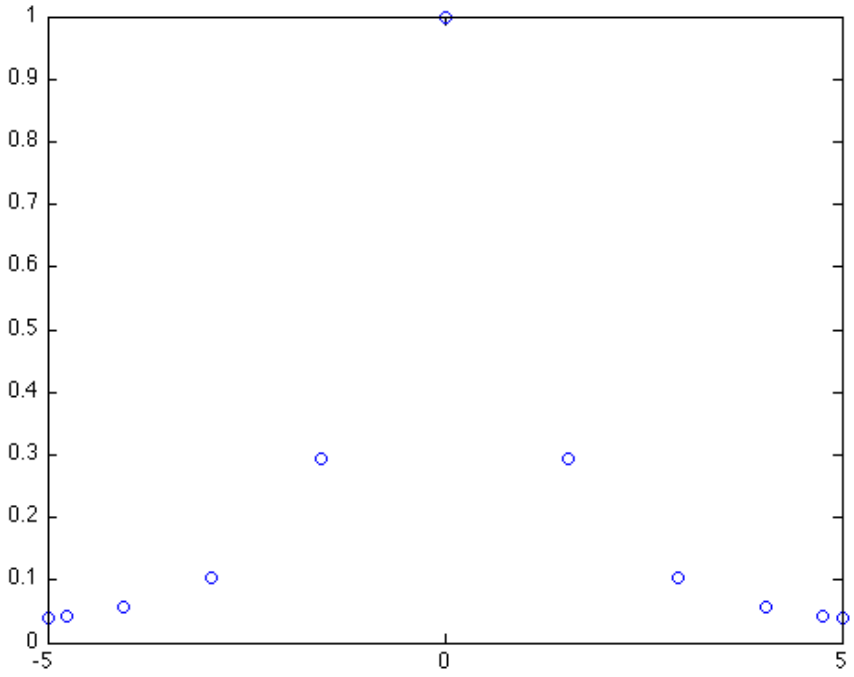
Today, we will introduce some powerful techniques for studying discreet data. Quite often, one performs a series of observations (such as counting bacterial growth each day) which is an example of discreet data. Sometimes there are pre conceived notions about how the observed phenomena is expected to 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:

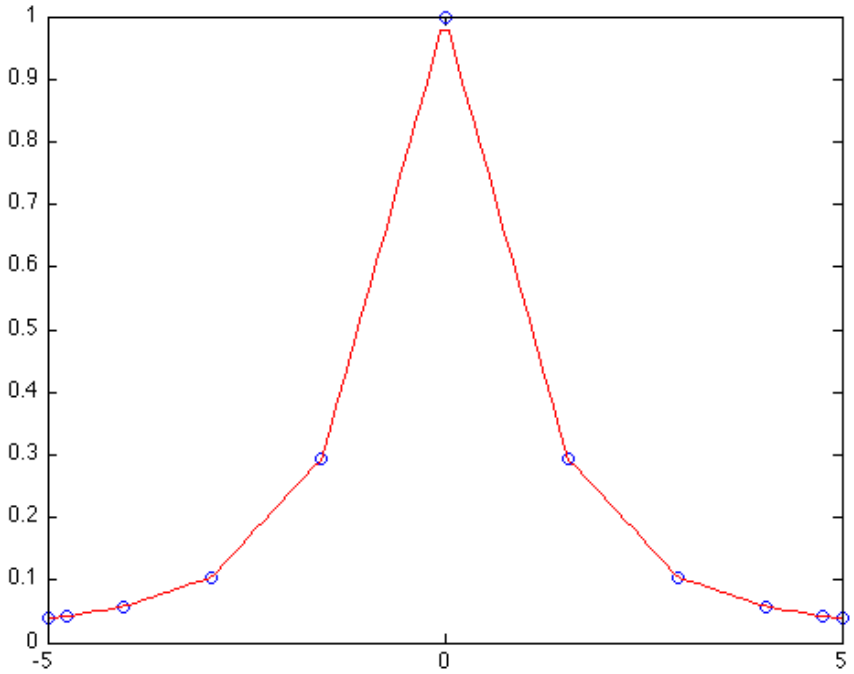



Now we will use Matlab's **interp1** function to linearly interpolate between these points:

```
>> 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:

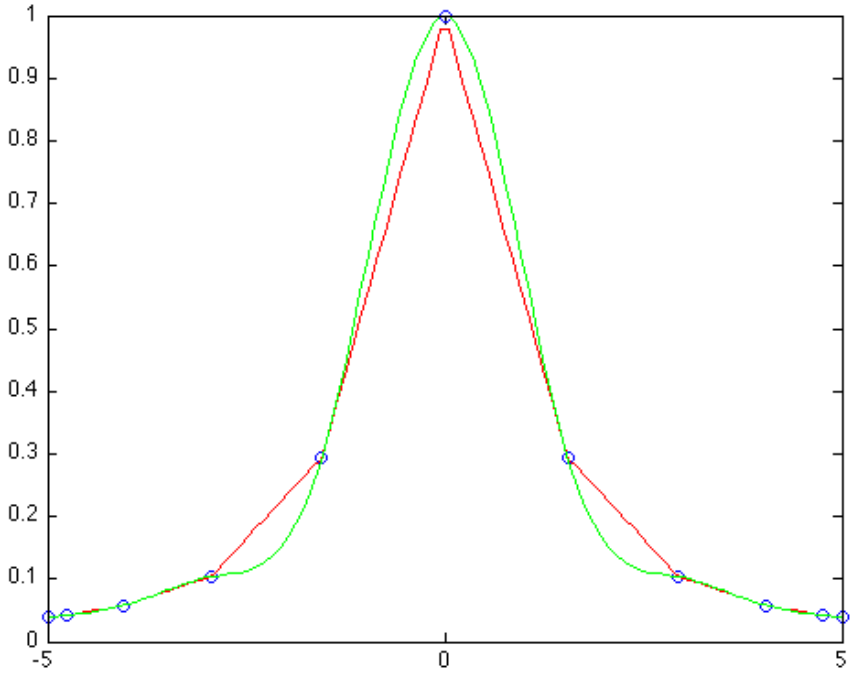



Notice that this looks like line segments were drawn between each discrete data point which is basically what linear interpolation is.

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

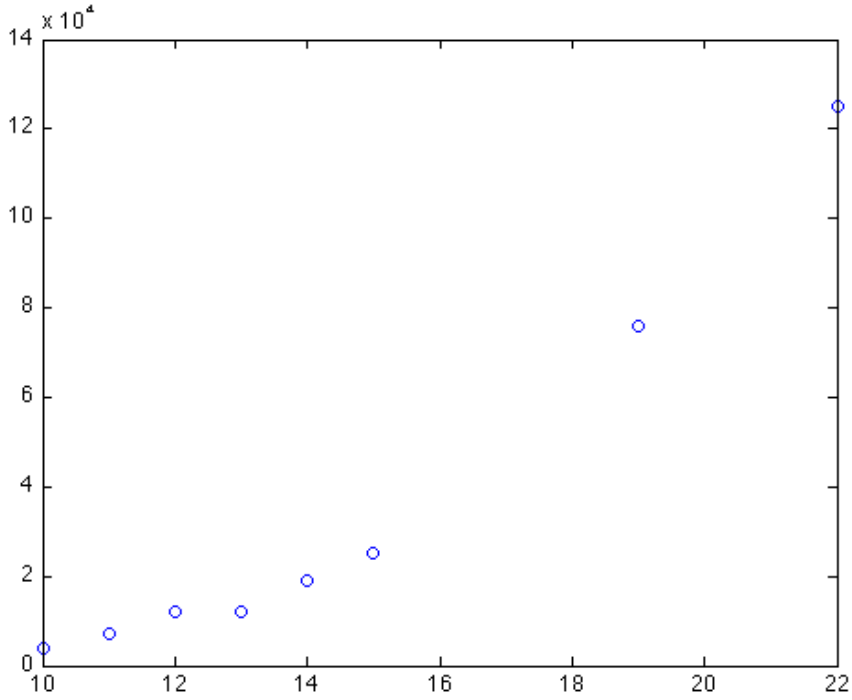
Curve fitting involves the attempt to find some sort of function that roughly mirrors (describes) the discrete data. For example, if a temperature reading at a single place was done every hour for months and then these points were graphed, there would be, at least in a rough sense, a rise and fall in 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);
```

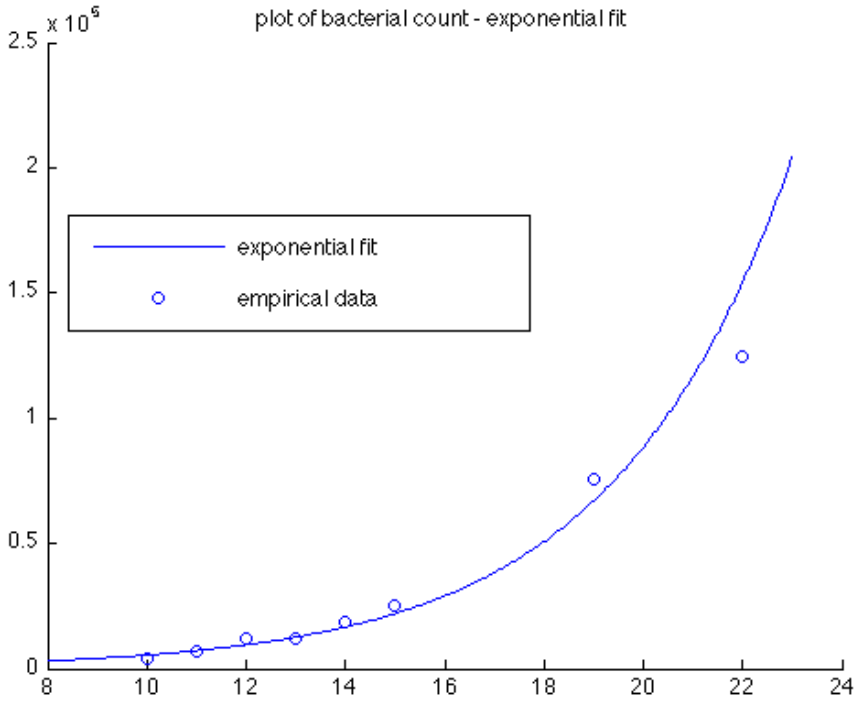
The inputs to this particular use of **polyfit** are: **dataDays** - the discrete data's x-values
**log(bacterialCount):** This "parameter" is actually a call to Matlab's **log** function to take the natural logarithm of the **bacterialCount** set of values. This is a good example of a parameter of one 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:

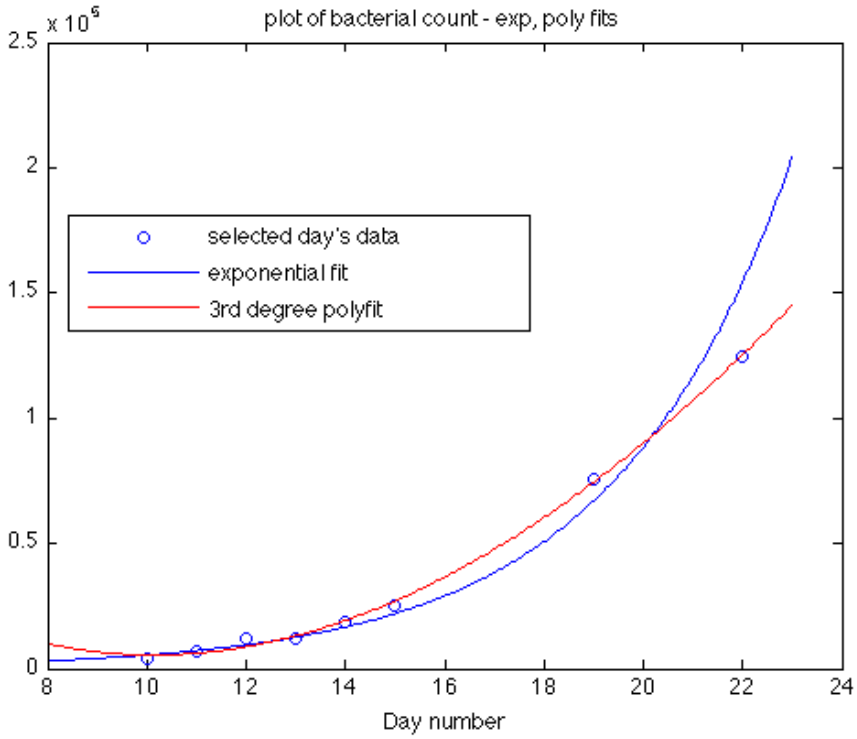
```
>> figure;
>> plot (dataDays, bacterialCount, 'o');
>> hold on
>> plot(timeVals, expFit);
>> p = polyfit(dataDays, bacterialCount, 3);

>> 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.

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. 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.

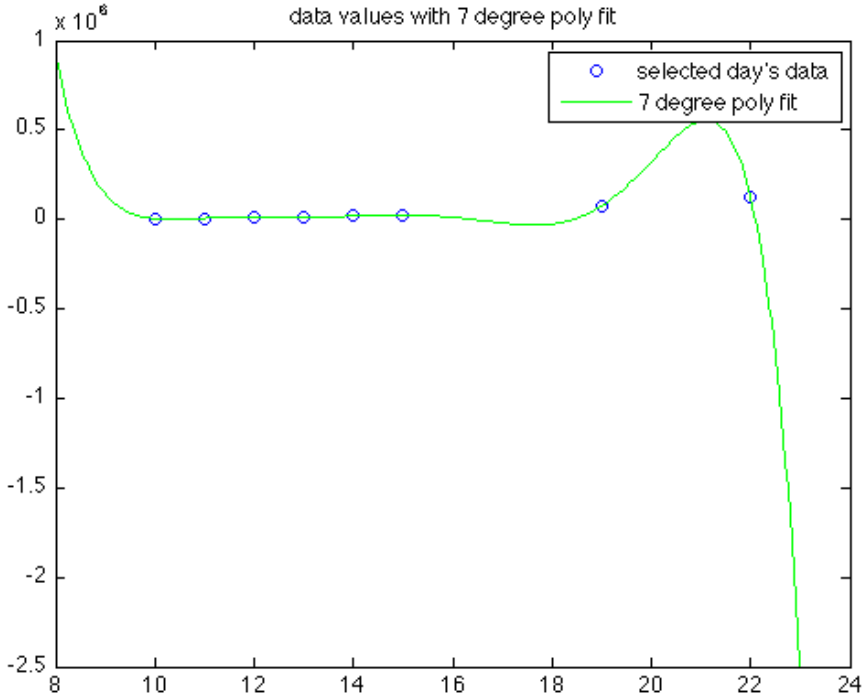
In fact, there is a mathematical theorem that states that for n-data points, there exists a polynomial of degree n will exactly "fit" the data. But this polynomial might also behave very different for values in between each data point than one would expect the actual data would be if observed during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

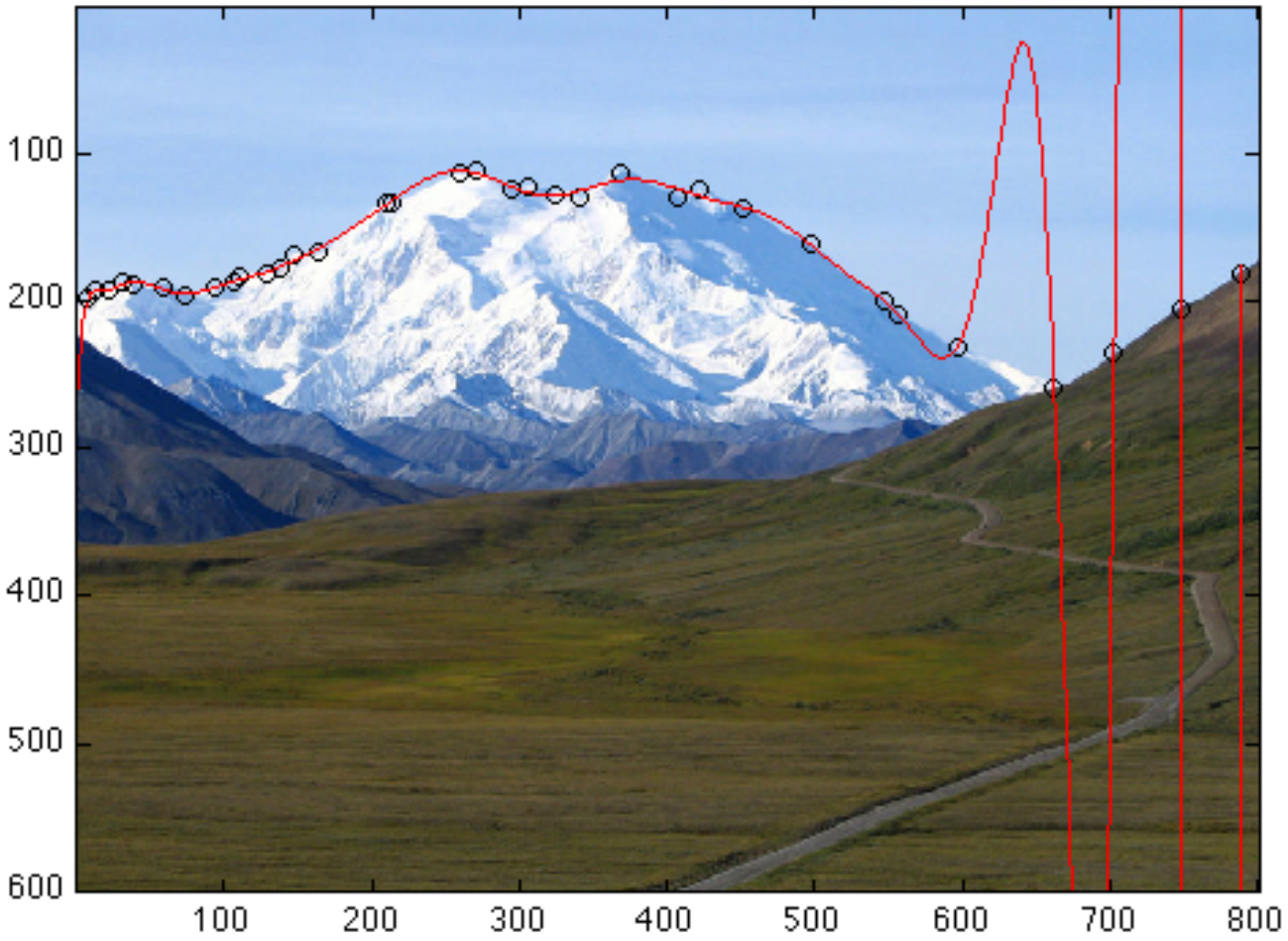
- You need to have **Mount.jpg** in your working Matlab path (i.e. Matlab working directory).
- When the picture of the mountain appears, better results are obtained if the image is expanded by grabbing the lower right corner with the cursor and dragging it down and to the right.
- Click at least 25 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('Mount.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:



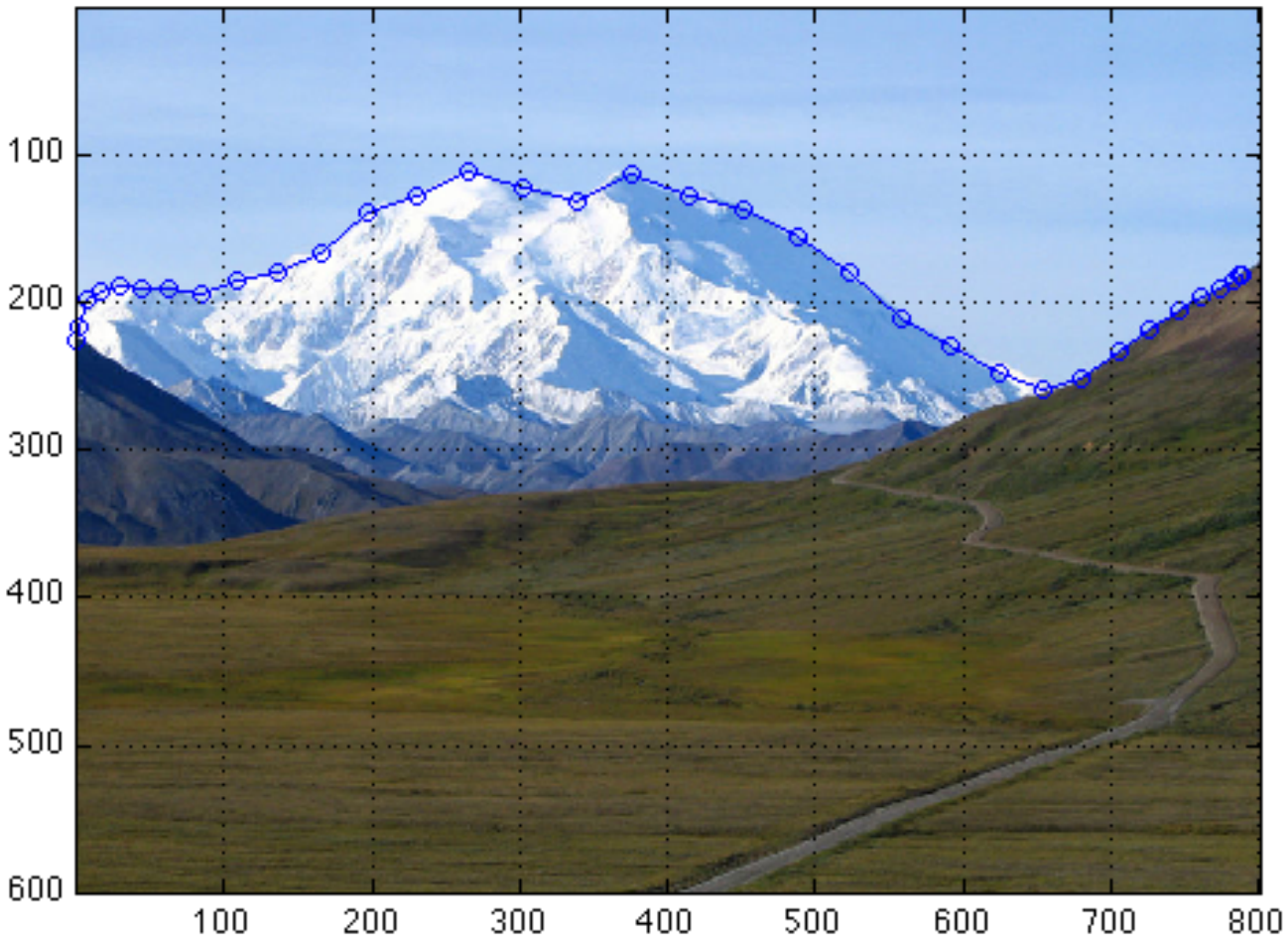

Now for a VERY GOOD way to greatly improve the fit WITHOUT increasing the number of points to sample:
This strategy basically involves sampling more often at the edges and where there is rapidly changing slopes while keeping the overall number of sampled points constant - i.e. if one is only allowed 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):



# 4. Home work number 7 hints:

**Problem 20:** Describes a classic ideal gas law relationship where it gives the formula **pressure =** (various items including volume **V**) which means that the pressure exerted by a closed container given various (given) inputs can be calculated.
However, this problem asks: given the pressure, compute the resultant volume. Mercifully, the text rearranges the equation into a 3rd degree polynomial such as:

```
a*V^3 + b*v^2 + c*V + d = 0;
```

Where **a b c d** are the coefficients. This 3rd degree polynomial can be easily solved, as explained in Chapter 8 by using Matlab's **root** function.
Briefly, set up a element vector consisting of these coefficients. For example:

```
a = 1                  % coefficient of V^3
b = -(n*b + n*R*T/p); % coefficient of V^2
c = ;                  % coefficient of V^1 ... follow the problem as I did for a, b
d = ;                  % coefficient of V^0 ... follow the problem as I did for a, b
coefficients = [a b c d];
V = roots(coefficients);
```

Since the above equation is cubic, there will be 3 solutions for **V**. You will be required to figure out which is the correct one. 2 of them are positive so you can't just pick the positive one. An easy way is to create an anonymous function of the original equation (given at the beginning of problem 20) such as:

```
pressure = @(n, R, T, tmpV, b, a) n*R*T/(tmpV - n*b) % fill in the rest
pressure(n, R, T, V(i), b, a);coefficients = [a b c d];
V = roots(coefficients);
```

Obviously, those variables in the anonymous function: **n, R, T, tmpV, b, a** need to be initialized - just use what the problem specifies. the **tmpV** variable is one of the **V** values returned by **roots**.

Then, for each value of **V** returned by the **roots** function, execute the above anonymous function to see what pressure is returned and then compare this number to the original pressure that you were asked to input to the cubic equation. The **V** value that is computes a pressure that is closest to the pressure that the problem asked to be used is the correct **V**. Then **fprintf** that particular V value.

To get full credit for this problem, the above needs to be done programatically (as opposed to just printing out which one you think it is).

**Problem 21:** This requires similar work as was done with the bacteria population problem described above (Curve Fitting), along with the discussion of interpolation described at

near the top of this document.

---

## SUMMARY: Things to remember from this lab

**Interpolation:**
- Linear interpolation is the simplest but results in discontinuous derivatives at each data point
- Spline interpolations are "smooth" (continuously differentiable) at each data point
- Depending on the nature of the data, one might be a more accurate representation compared to the other

**Curve Fitting:**
- exponential fit using Matlab's **polyfit** with the **log** of the function values used as input
- polynomial fit also using **polyfit** using the actual function values as input along with the degree of polynomial desired
- It is important to consider the nature of the phenomena before deciding whether exponential or polynomial fits (or what degree of polynomial) are used

**Important General Review Items:**
- Inline functions
- anonymous functions
- how to input vectors into functions and interpret the vectorized outputs
- ELEMENT BY ELEMENT multiplication, division, exponentiation is usually called for (using the dot before the operators)

## 5. HOMEWORK FORMAT (review from last week):

Organize this home work assignments as follows:
Put all problems into one .m file similar to below by making the first line identify the class, week number and YOUR NAME. Then, make a new section (demarcated by

%%

) for each new problem

```
%% Your Name, Math 265 WEEK 7 HOMEWORK #7 problems
%% Chapt. 8 #20

< all code for chapt. 8 #20>
```

```
%% Chapt. 8 #21

< all code for chapt. 8 #21>
```