APSC 1001 MATLAB Curve Fitting Tools

Randy Schur

October 2, 2015

1 Sampling Data

An valuable part of your toolbox of engineering skills is testing, data collection, and analysis. Much of what scientists and engineers study was originally discovered by experimentation and the analysis of experimental data. The ability to decipher some meaning from the data you collect is something you will rely on throughout your engineering career.

1.1 Signal Analysis

Many textbooks have been written on the analysis of different types of signals, and we won't go into great detail in this class. Some common examples of applications that demand signal analysis are circuit design and radio frequency (RF) work in electrical engineering, image processing in computer science, wind or water tunnel experiments in mechanical or aerospace engineering, and earthquake movement in civil engineering.
Often, the signals we see are repeating. We call this *periodic*, meaning that the signal is repetitive and has a measurable period, or length of time between cycles. The simplest example of a periodic signal is a sine wave. In fact, any periodic signal may be replicated using only sine (or cosine) waves of different frequencies! This is a surprising and extremely important result in many fields of mathematics and engineering. Breaking a signal up into its component frequencies is called a Fourier Transform, named after Joseph Fourier who (sort of) discovered that periodic, or *sinusoidal*, functions can be approximated by sine waves. It is something that you will likely discuss in several classes during your studies.

1.2 Statistical Approximations

Not all data that we collect is from a periodic signal. Sometimes we have a single process, and we test it multiple times while changing one or more variables. An example of a data collection task like this might be testing for when a particular component fail. You might vary the loading on that component, and run the test multiple times. Once you have collected data from multiple runs of the experiment, you might try to find the how the failure of the component depends on the loading. You are looking for a *function* for failure in terms of loading. Because we can't perfectly control any experiment, there will be some element of randomness to the data you collect. However, using some statistical tools, it it often possible to find a relationship between the two (or more) variables that you observing. This process is called *curve fitting*.

2 Curve Fitting in MATLAB

MATLAB provides several built in functions that can help with curve fitting. We are going to use a tool with a nice GUI called cftool. See the example below to learn how to use this tool.
One way we can test the tool is to make up a function, inject some randomness in order to model 'experimental' data, and then see if we can find the original function.
Type the following into MATLAB.

```
>> x = linspace(0,100,101); %create initial data
>> y_actual = 2.5*x + 17; %function of x
>> rnums = 15*randn(1, 101); %random numbers with a zero mean (no bias)
>> y_test = y_actual + rnums; %create 'test' data.
```

Now we'll use the curve fitting toolbox to see if we can get the correct function from our test data.

```
>> cftool(x, y_test)
```

This brings up the curve fitting GUI. The screen shot below has several important values highlighted, which MATLAB often fills out automatically. The first is the type of curve that we are looking for, which in this case is a first degree polynomial. The next important part is the selection of data that we are using - it is good to confirm the correct data is being used for the x and y axis. Below the variables is the results section, where you can find the function that MATLAB has generated, and the coefficients to go along with that function. Also located in that box is a section titled Goodness of Fit. The value we are interested in is the R-square value, which can act as an estimate of how well the generated curve fits the data set. A value closer to 1 means a better fit. Finally, there is a plot of the data and the of the curve. Notice that the data is plotted as discrete points, while the function is plotted as a continuous line. This is conventional; test data should always be plotted discretely and functions should be plotted as continuous.

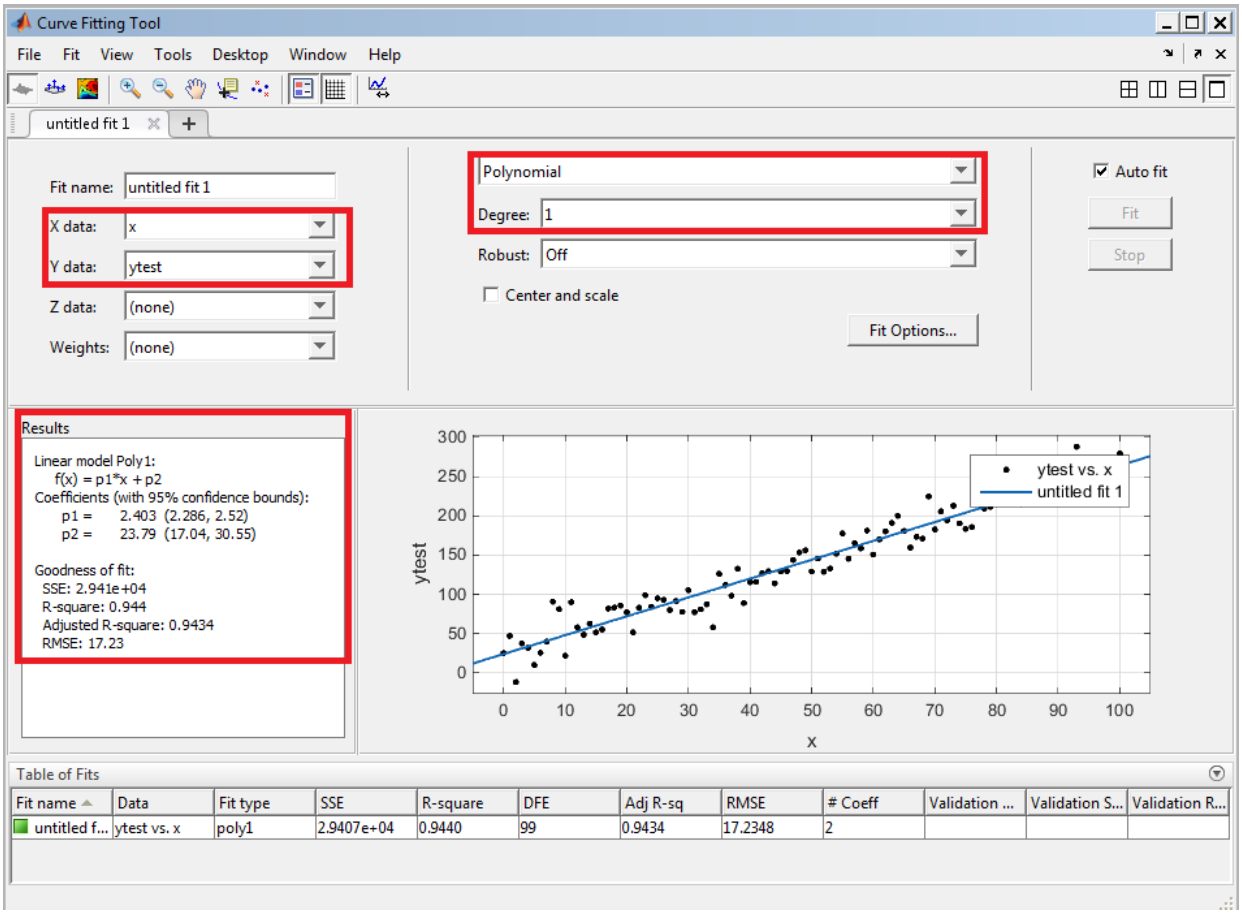

Figure 2.1: MATLAB's curve fitting toolbox GUI.

There are several ways to save the generated curve so that we can use it in a script or the command window. Here, we are going to save the coefficients as variables, and then plot the curve again on our own. So, in this case we would type into the command window:

```
>> p1 = 2.403;
>> p2 = 23.79;
```

For a first degree polynomial, this is all the information we will need!
Now, if we want to plot the generated curve there are two options. The first is to create an array containing a y value for each x value.

```
>> y_curve_fit = p1*x + p2;
```

Then, we can plot x vs. y_curve_fit on the same plot as x vs. y_test, and recreate the plot we saw in the curve fitting tool. Try this, and create a title, axis labels, and legend. Does your plot match the one in the tool? Why might it be useful to create your own plot rather than use the one in the toolbox?

2.1 Writing a function

There is another option for generating the values in y_curve_fit. MATLAB has many built in functions, but we can also write our own. Type the following line into the command window:

```
>> f = @(s) p1*s + p2;
```

We have just written our first MATLAB function! Now, we can do the following:

```
>> y_generated = f(x);
```

Our function, f, takes an argument of a vector and returns a vector representing our fitted curve. In addition to using x, we could create any array we want and run it through our function. The result would be a segment of the line generated by the curve fitting toolbox. Try creating a new x vector that has values from 0 to 1000, and use this function to generate y values. Then, plot the result on the same graph as the original data set. Why might this be useful?