Fitting Data with *Mathematica*

In a number of experiments this semester, you will be asked to fit experimental data to a theoretical curve. This notebook walks through an example of doing that in *Mathematica*.

For this notebook, we will look at fitting a simple oscillating function to some data.

Entering Data

There are several ways to enter data in *Mathematica*. If you only have a small number of data points, you can just enter them directly. You can use the Palettes -> Classroom Assistant -> Typesetting -> Matrix entry, or you can just enter in the data by hand.

Entering by hand

If you enter it by hand, note that each data point is a list, or pair: { t, x } . As always in *Mathematica*, lists are enclosed in curly braces. The complete data set is just a list of those data pairs: { { t1, x1 }, { t2, x2 } }, etc. See below for an example data set. You can also enter data tables with the Matrix tool from the Classroom pallette.

Importing from a file

First, tell *Mathematica* where the data can be found. You will have to change this for your own set-up.

```
In[19]:= SetDirectory["E:/218/2011/lab-svn/labs/torsional"]
```

```
SetDirectory : Cannot set current directory to E:/218/2011/lab-svn/labs/torsional .
```

```
Out[19]= $Failed
```

If your *Mathematica* notebook and your data file are in the same directory, this next trick sometimes works. However, you have to save the notebook first (even if it’s just an empty notebook).

```
In[20]:= SetDirectory[NotebookDirectory[]]
Out[20]= /home/doughera/notes/Physics-Labs-git/intermediate/Phys218/labs
```

Importing from a plain text file.

If your data is a simple text file with two columns of numbers, you may import it as a "Table".

```
In[21]:= data = Import["curvefit-data.txt", "Table"]

Out[21]= {{0., -1.438}, {0.1, -1.321}, {0.2, -0.891}, {0.3, -0.271}, {0.4, 0.403}, {0.5, 0.989},
          {0.6, 1.355}, {0.7, 1.433}, {0.8, 1.165}, {0.9, 0.652}, {1., -0.007}, {1.1, -0.662},
          {1.2, -1.175}, {1.3, -1.433}, {1.4, -1.35}, {1.5, -0.965}, {1.6, -0.359}, {1.7, 0.31},
          {1.8, 0.921}, {1.9, 1.321}, {2., 1.429}, {2.1, 1.223}, {2.2, 0.725}, {2.3, 0.09},
          {2.4, -0.569}, {2.5, -1.116}, {2.6, -1.404}, {2.7, -1.375}, {2.8, -1.023}, {2.9, -0.442}}
```

Importing from a CSV file.

If you have used LoggerPro and exported the data to a CSV file (Comma-separated-values) then *Mathematica* can easily import the data.

```
In[22]:= data = Import["curvefit-data.csv"]

Out[22]= {{Time, Potential}, {0., -1.438}, {0.1, -1.321}, {0.2, -0.891}, {0.3, -0.271}, {0.4, 0.403},
          {0.5, 0.989}, {0.6, 1.355}, {0.7, 1.433}, {0.8, 1.165}, {0.9, 0.652}, {1., -0.007},
          {1.1, -0.662}, {1.2, -1.175}, {1.3, -1.433}, {1.4, -1.35}, {1.5, -0.965}, {1.6, -0.359},
          {1.7, 0.31}, {1.8, 0.921}, {1.9, 1.321}, {2., 1.429}, {2.1, 1.223}, {2.2, 0.725}, {2.3, 0.09},
          {2.4, -0.569}, {2.5, -1.116}, {2.6, -1.404}, {2.7, -1.375}, {2.8, -1.023}, {2.9, -0.442}}
```

LoggerPro included titles for the two columns of data ("Time" and "Potential", in this case). You need to remove them from the data before you can use it. There are many, many ways to do this in *Mathematica*. In the on-line help, look for the tutorial on "Getting Pieces of Lists." (Of course you can also simply edit the original text file as well.)

The most direct is with the Drop[ ] command. The following drops the first element from the 'data' array and assigns it back to the data array.

```
In[23]:= data = Drop[data, 1]

Out[23]= {{0., -1.438}, {0.1, -1.321}, {0.2, -0.891}, {0.3, -0.271}, {0.4, 0.403}, {0.5, 0.989},
          {0.6, 1.355}, {0.7, 1.433}, {0.8, 1.165}, {0.9, 0.652}, {1., -0.007}, {1.1, -0.662},
          {1.2, -1.175}, {1.3, -1.433}, {1.4, -1.35}, {1.5, -0.965}, {1.6, -0.359}, {1.7, 0.31},
          {1.8, 0.921}, {1.9, 1.321}, {2., 1.429}, {2.1, 1.223}, {2.2, 0.725}, {2.3, 0.09},
          {2.4, -0.569}, {2.5, -1.116}, {2.6, -1.404}, {2.7, -1.375}, {2.8, -1.023}, {2.9, -0.442}}
```

Another more general way to select parts of a list is to use the Select[ ] function. This allows you to select elements from a list that meet some criterion. See the tutorial on Finding Expressions that Match a Pattern" for more on the Select[ ] function. The syntax is likely unfamiliar, but the following looks at all the elements of data, and selects only those for which all the elements are numbers.

The Select function loops through our data and assigns the symbol ‘#’ to each element in turn. It then selects only the elements for which our test is True. Since each element is a pair, or list of items, we need to look at both of them. The function VectorQ does that. It applies the “NumberQ” test to each of the elements of “#” and returns True only if all of them are numbers. (Be sure to include the '&'. Look up the online help for Function[ ] to learn more about what it is doing.)

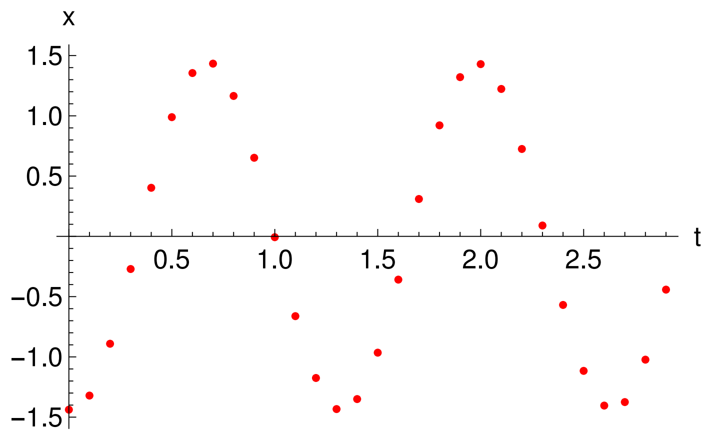
```
In[24]:= data = Import["curvefit-data.csv"]
         data = Select[data, VectorQ[#, NumberQ] &]

Out[24]= {{Time, Potential}, {0., -1.438}, {0.1, -1.321}, {0.2, -0.891}, {0.3, -0.271}, {0.4, 0.403},
          {0.5, 0.989}, {0.6, 1.355}, {0.7, 1.433}, {0.8, 1.165}, {0.9, 0.652}, {1., -0.007},
          {1.1, -0.662}, {1.2, -1.175}, {1.3, -1.433}, {1.4, -1.35}, {1.5, -0.965}, {1.6, -0.359},
          {1.7, 0.31}, {1.8, 0.921}, {1.9, 1.321}, {2., 1.429}, {2.1, 1.223}, {2.2, 0.725}, {2.3, 0.09},
          {2.4, -0.569}, {2.5, -1.116}, {2.6, -1.404}, {2.7, -1.375}, {2.8, -1.023}, {2.9, -0.442}}

Out[25]= {{0., -1.438}, {0.1, -1.321}, {0.2, -0.891}, {0.3, -0.271}, {0.4, 0.403}, {0.5, 0.989},
          {0.6, 1.355}, {0.7, 1.433}, {0.8, 1.165}, {0.9, 0.652}, {1., -0.007}, {1.1, -0.662},
          {1.2, -1.175}, {1.3, -1.433}, {1.4, -1.35}, {1.5, -0.965}, {1.6, -0.359}, {1.7, 0.31},
          {1.8, 0.921}, {1.9, 1.321}, {2., 1.429}, {2.1, 1.223}, {2.2, 0.725}, {2.3, 0.09},
          {2.4, -0.569}, {2.5, -1.116}, {2.6, -1.404}, {2.7, -1.375}, {2.8, -1.023}, {2.9, -0.442}}
```

Plotting the data

This function plots the data, but also saves a copy of the plot in the variable 'dataplot' for later use. I find the default labeling to be too small; this command makes it a bit larger.

```
In[26]:= dataplot = ListPlot[data, PlotRange → All,
           AxesLabel → {"t", "x"}, LabelStyle → Larger, PlotStyle → Red]
```

Out[26]=



Fitting a Model to the Data

For this exercise, we will try to fit this to a cosine function, allowing for the possibility that the average value might not be zero, due to some offset in the sensor.

```
In[27]:= x[A_, ω_, ϕ_, t_, xoff_] := A Cos[ω t + ϕ] + xoff
```

The basic function to use is NonlinearModelFit[ ]. Consult the online help for more details. This notebook will just illustrate the basic usage.

```
In[28]:= fit = NonlinearModelFit [data , x[A, ω, ϕ, t, xoff], {A, ω, ϕ, xoff}, t]
```

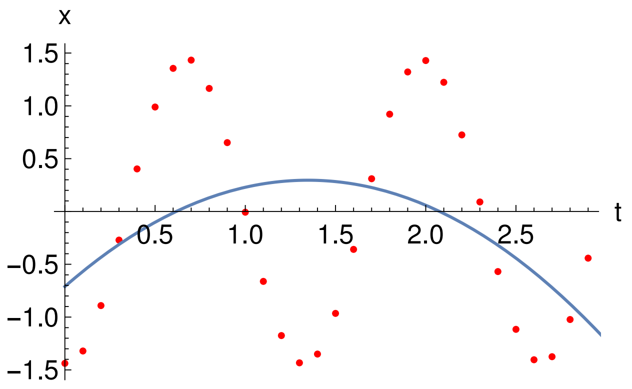
```
NonlinearModelFit : Failed to converge to the requested accuracy or precision within 100 iterations .
```

```
Out[28]= FittedModel[ -878.989 + 879.285 Cos[0.0478572 - 0.0355321 t] ]
```

It is also easy to generate a plot from the fit. This pair of lines generates the plot, gives it a name, and then combines it with the data plot. The Show[] command shows them both together. It is easy to make the graph larger -- the ImageSize -> Scaled[0.50] makes the image equal to 1/2 the screen width. Adjust as you see fit.

```
In[29]:= fitplot = Plot[fit[t], {t, 0, 3}, PlotRange → All];
         Show[{dataplot , fitplot}, ImageSize → Scaled[0.50]]
```

Out[30]=



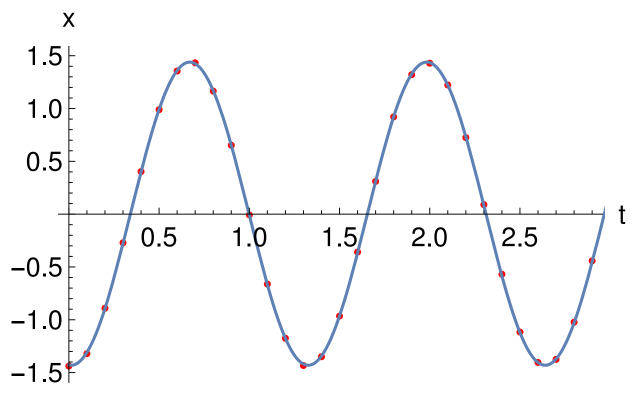
This fit is quite poor. *Mathematica* performs the fit by starting with an initial guess, and then seeing how the fit improves as it changes the guess. (LoggerPro works in a similar way.) You can give it a hand by suggesting a better initial guess for any or all of the parameters. For example, looking at the graph, it is clear that the period is a little over 1.0 seconds. Specifying an initial guess for ω of $2\pi/1.0$ is likely to be closer to the desired result.

```
In[31]:= fit = NonlinearModelFit [data , x[A, ω, ϕ, t, xoff], {A, {ω, 2 π / 1.0}, ϕ, xoff}, t]
```

```
Out[31]= FittedModel[ 0.00441981 + 1.43493 Cos[3.07873 + 4.7841 t] ]
```

```
In[32]:= fitplot = Plot[fit[t], {t, 0, 3}, PlotRange → All];
         Show[{dataplot , fitplot}, ImageSize → Scaled[0.50]]
```

Out[33]=



Fit Diagnostics

NonlinearModelFit returns a variety of measures for how "good" the fit is. The online help has more details, but three particularly useful ones are shown here.

Best Fit Parameters

```
In[34]:= fit["BestFitParameters "]
Out[34]= {A → 1.43493 , ω → 4.7841 , ϕ → 3.07873 , xoff → 0.00441981 }
```

Parameter Confidence Interval

This item gives both the uncertainty and the 95% confidence-level interval.

```
In[35]:= fit["ParameterConfidenceIntervalTable "]
```

Out[35]=

	Estimate	Standard Error	Confidence Interval
A	1.43493	0.00199071	{1.43084 , 1.43902 }
ω	4.7841	0.00169737	{4.78062 , 4.78759 }
ϕ	3.07873	0.00289789	{3.07277 , 3.08469 }
xoff	0.00441981	0.00144711	{0.00144523 , 0.00739438 }

Correlation Matrix

The "CorrelationMatrix" tells you about the correlations among the fit parameters. Low values mean the parameters are poorly correlated. This is good -- it means they are mostly independent. On the other hand, high values mean the parameters are highly correlated. This is often a sign of trouble in the proposed fit. For example, if you are trying to fit the parameters A and B in the function

$$f[x] = A\frac{x}{B}$$

then the fitted parameters for A and B will be highly correlated. You could double A and B, and still get the same answer.

```
In[36]:= fit["CorrelationMatrix "] // MatrixForm
Out[36]//MatrixForm=
```

$$\begin{pmatrix} 1. & 0.0313374 & -0.000416606 & 0.125407 \\ 0.0313374 & 1. & -0.872054 & 0.16148 \\ -0.000416606 & -0.872054 & 1. & -0.17947 \\ 0.125407 & 0.16148 & -0.17947 & 1. \end{pmatrix}$$

For example, the 3rd element on the first row (-0.000416606) means that changes in the first parameter (A) are quite weakly correlated with changes in the third parameter (ϕ). The 1's along the diagonals mean each parameter is perfectly correlated with itself, as expected. Checking the correlation matrix is a good way to check whether your proposed fitting function really has the independent parameters you thought it did.

The resulting fit object has many other “Properties” you can query. Here is a list of all of them:

```
In[37]:= fit["Properties "]
Out[37]= {AdjustedRSquared , AIC , AICc , ANOVATable , ANOVATableDegreesOfFreedom ,
          ANOVATableEntries , ANOVATableMeanSquares , ANOVATableSumsOfSquares ,
          BestFit , BestFitParameters , BIC , CorrelationMatrix , CovarianceMatrix ,
          CurvatureConfidenceRegion , Data , EstimatedVariance , FitCurvatureTable ,
          FitCurvatureTableEntries , FitResiduals , Function , HatDiagonal ,
          MaxIntrinsicCurvature , MaxParameterEffectsCurvature , MeanPredictionBands ,
          MeanPredictionConfidenceIntervals , MeanPredictionConfidenceIntervalTable ,
          MeanPredictionConfidenceIntervalTableEntries , MeanPredictionErrors ,
          ParameterBias , ParameterConfidenceIntervals , ParameterConfidenceIntervalTable ,
          ParameterConfidenceIntervalTableEntries , ParameterConfidenceRegion ,
          ParameterErrors , ParameterPValues , ParameterTable , ParameterTableEntries ,
          ParameterTStatistics , PredictedResponse , Properties , Response ,
          RSquared , SingleDeletionVariances , SinglePredictionBands ,
          SinglePredictionConfidenceIntervals , SinglePredictionConfidenceIntervalTable ,
          SinglePredictionConfidenceIntervalTableEntries ,
          SinglePredictionErrors , StandardizedResiduals , StudentizedResiduals }
```