Data Analysis and Model Fitting

Data and The Assignment

Described below (after a load of preliminary material) are four model-fitting problems. Please do at least two of the problems; you pick. In each case, the idea is to argue for the form of the best fit from among a specified set of models (linear, quadratic, cubic, quartic, exponential, and power law). The underlying theory, if you can find one, can help in model selection. There are established theories for predicting the behavior in the Water Flow & Gas Pressure, Primes, and Computer Science problems. There's a lot of prose (90% of it "how-to") in this assignment. Give yourself enough time.

The first two problems, (Thermocouple, and Fluid Flow & Gas Pressure use data in the Data Directory, courtesy of Prof. Roger Gans and his ME 241 course. You should be able to grab a file from this directory using the file names mentioned in the problems (PRTdata.xls, Thermo.xls, Boyle.xls, Flow.xls).

The files are excel files. Excel (by Microsoft) is what is called a "Spreadsheet" program, which basically implements smart tables. It is commonly used for for passing around data. To import an excel file into Matlab, go to the Matlab's File menu and halfway down you'll see "import data...". Some of the files have extra stuff besides numbers in them (labels etc.) and you may have to clean up the data. Once you have got it, you might store it in more convenient for Matlab .mat files.

These files all have an independent variable as the first column and a dependent variable in the second. (Yes, the Flow.xls file has a column A, but you may have to mess with the slider bar to see it). The label information may give you a hint about what the quantities actually are, which is crucial knowledge in a real engineering problem, but we can experiment with data-fitting with just the numbers. In general, the $i^{th}$ row contains first, $x_i$, and then $y_i$, for one data point.

In the "computer science" problem you get to make your own data. You will write an instrumented sorting function that produces an integer output from an integer input.

\[ \text{input } x \rightarrow \text{SORT } \rightarrow \text{output } y. \]

The input $x$ is the length of a vector of random numbers that will be created, and $y$ the number of comparisons (or the CPU time) needed to "bubble-sort" the vector of numbers. You are invited to run the experiment for a range of $x$ values (lengths from 1 to 100) and fit the $(x,y)$ data with a model.

All problems ask you to find a model ('law') for data and discuss evidence for (and against) it. Finding the appropriate model may involve: eyeballing (a plot of) the data, physical intuition, knowledge of underlying theory, and actually fitting and analyzing various forms.

The work could involve research, (e.g. looking for the right law in the literature, either before or after you find your favorite). For now we will use the standard deviation of the residuals (defined below) as our quantitative measure of the quality of fit. The tricky bit is trading off numeric fit quality against less objective measures such as complexity of the model (generally simpler is better) agreement with theoretically expected form.

Fitting Algorithms Are So Important That...

Matlab has built many of them in: poly, polyval, polyfit, roots, lscov, mean, std etc. Don't even think about using them (unless you want to check your results). Even that is dangerous since it turns out that the built-in std is the wrong error measure for most of the problems you will run. One of the goals of this project is to write your own versions of some of these built-ins (polyfit() in particular). There are good reasons for this: to understand the mathematical meaning, to get more programming experience, and to have control of your own experimental code.

Organizing your Code
We're somewhat reluctantly dictating a code organization here; you should first figure out why it makes sense, and consider its advantages and disadvantages (if any). There are other approaches, but let's stick with this for now. If you have questions, ask. **Summary:**

Functions to write (more are welcome).

```matlab
function Coefs = FitPoly(XYVals, N)
function [a, k] = FitExp(XYVals)
function [a, p] = FitPower(XYVals)
function VDM = VanderMonde(XYVals, N)

function YVals = GenPolyVals(Xvals, N, Coefs)
function YVals = GenExpVals(Xvals, a, k) % computes a * exp(k * x)
function YVals = GenPowerVals(Xvals, a, p) % computes a * x^p
function StdError = StdDev(ModelYVals, DataYVals, ModelDOF)
```

**The Fitting Functions**

In each of the functions `XYVals` is an M x 2 matrix in which each row is an `(x_i, y_i)` pair from the data set we're fitting. `N` is the degree of the polynomial (1 for linear, 2 for quadratic,...). `XVals` is the left column of `XYVals`, that is the vector of `x_i` data-point values. The `GenXXX` functions generate the vector `YVals`, the vector of y values that results from substituting the `XVals` vector into a model: exponential and power-law models have two parameters `(a, k)` and `(a, p)` respectively. Polynomial models have `N+1` parameters, the coefficients of the `N`th-degree polynomial determined by the `N+1` elements in the `Coefs` argument. `StdError` takes the `ModelYVals` vector returned as `Yvals` from one of the `GenXXX` functions, and the `DataYVals`, which is the second (`y_i`) column of the `XYVals` data-points matrix (obviously these `y` values should correspond to the `XVals`). By looking at their size or length, the function knows how many pairs there are: let's call that number DOF. `StdError` also needs `ModelDOF`, the number of degrees of freedom to subtract from DOF due to the model being tested; that is, the number of model parameters (see above). The sum of squared errors is divided by DOF - `ModelDOF`, and the square root of that is our standard error.

The first function, `FitPoly()`, pretty much does all the fitting work. `FitExp()` and `FitPower()` for exponential and power laws basically do some data preprocessing, and then call `FitPoly()` to fit a line to the transformed data. See the tutorial... `FitPoly()` returns the coefficients of the best-fitting `N`th degree polynomial for the input data `XYVals` (our M-row, 2-column vector of `(x,y)` data points). If M < (N+1), `FitPoly()` should report an error, since we don't have enough points to fit an `N`th degree polynomial;

`FitPoly` uses the theory presented in the readings and lectures to solve a system of linear equations whose unknowns are the coefficients of an `N`th order polynomial. You could use your own Gaussian elimination program if you want to feel really empowered (be sure to mention it in your writeup if you do). However, in this particular situation, you may use the Matlab built-in solver by employing the backslash operator as discussed in the lecture notes.

To implement the theory, `FitPoly` calls the function `VanderMonde()` to create the necessary VanderMonde Matrix. Recall that the VanderMonde matrix for an `N`th-order polynomial fit, is an M x `(N+1)` array, whose `i`th row is 

\[
[ x_i^0 \ x_i^1 \ x_i^2 \ ... \ x_i^N ]
\]

So the first column is always 1's.

As mentioned above, `FitExp` uses to fit a line to logarithmically transformed `ydata` and obtain the parameters `a` and `k` of a `y(x) = a e^{k x}` model fit. Similarly, `FitPower` uses to fit a line to logarithmically transformed `x` and `y` data and obtain the parameters `a` and `p` of a `y(x) = a x^p` model fit.

**More possibly helpful verbiage**

**though you should be able to proceed on your own at this point...**

Calling the VanderMonde function could look like this

```matlab
VDM = VanderMonde(XYVals, N)
```

and for an M-row input `XYVals`, it ignores the second (Y) column and produces an M x `N+1` VanderMonde matrix.
in which the \( i \)th row looks like 
\[ [x_i^0, x_i^1, x_i^2, ..., x_i^N]. \]

Now at this stage we all have our own style. The above algorithm is clearly possible with a couple of for-loops, but if you'd rather you should feel free to unleash the vectorizing power of Matlab. With \( X \) the \( M \)-long column vector of independent variable values, notice that the first column is always \( X.^0 = 1 \), the second is just \( X.^1 \). Recall \( ^\cdot \) exponentiates each element in \( X \) to a power. The third is \( X.^2 \), then \( X.^3 \), etc. To be even more explicit, you could make an \( M \times N+1 \) matrix of zeroes (here called VDM), and then use a single for-loop setting (say) \( k \) from 0:N. Inside that loop you use the \( : \) operator to exponentiate the whole \( X \) (first, presumably) column of XYVals to the \( k \)th power, for \( k = 0, 1, ..., N \) and stick that whole column into your (initially zero) VanderMonde matrix's \( k+1 \)st column. That means there's only one line inside the for-loop.

Now \( \text{FitExp()} \) and \( \text{FitPower()} \). For \( \text{FitExp()} \), you're investigating whether the logarithm of the \( Y \)'s is linear in \( X \), so you need an \( M \times 2 \) VanderMonde matrix with 1's down the first column and \( \log(Y) \) down the second. Everything else is as before and the only issue is how well that straight-line fit to \( \log(Y) \) works. Again, just use \( \text{Analyze()} \).

\( \text{FitPower()} \) is to investigate power-law relationships, and as the tutorial says, that's when the log of \( Y \) is linearly dependent on the log of \( X \). So the VanderMonde matrix is just as in \( \text{FitExp()} \), and the \( Y \)'s need to be replaced by \( \log(Y) \).

Finally, you might consider one function \( \text{CoefMat = FitAll(XYVals, MaxN)} \) that returns a \( \text{MaxN+2} \) -column matrix: \( \text{MaxN} \) columns of polynomial fit coefficients for all polynomial fits from \( N=1 \) through \( \text{MaxN} \), and the last two columns for \( \text{FitExp()} \) and \( \text{FitPower()} \) results. Just a thought!

### Analyzing the Fit

Our basic problem is that, in general, we don't know what model to choose! Should we fit a linear model? a power law? an exponential? or some higher order polynomial? If we know the physical law, we have a pretty good starting point, and we should have a good argument for what we are trying to do if we depart from it. (E.g., the physics model we have is linear, but we know this is only approximate, and we are trying to calibrate a sensor by adding a quadratic component to the basic linear law).

If we really have no idea, then we are stuck with trying to find a good guess. A starting point is to graph the data, and see if it appears to lie about a straight line. Your eye is remarkably good at this. If there is no apparent simple pattern (concave upwards or downwards, or clear pattern of maxima and minima), a linear fit is probably as good as you are going to get. If there appears to be some curved pattern we may need to try fitting different models and compare the fits.

This process is fraught with difficulty. A reasonable first step is to establish some measure of departure from the model. There are many formulas for doing this, but we will make use of the square root of the mean squared deviation from the model (aka the standard error). The computation of this is discussed in detail in the lectures and readings. For now, it provides us with a single positive number, and the lower that number, the better the "fit". If the standard error is 0, the model goes exactly through every data point. This is not necessarily a good thing, as it may imply a phenomenon referred to as "overfitting".

That is, we cannot just fit each of our models to all our data and take the one with the lowest numeric deviation. To see this, consider that, for a given set of data, a quadratic equation will ALWAYS fit at least as well as (and almost always better than) a line, because a line IS a special case of a quadratic equation. Similarly an \( N+1 \) degree polynomial will always fit as well or better than an \( N \) degree one. A polynomial will often fit better than a power law because integer power laws are special cases of polynomials.

An extreme example: if we have 1000 data points, we can find a 999-degree polynomial (thus 1000 coefficients) that goes through them all with no error. But there are no physical laws that involve 999-degree polynomials, and using 1000 parameters to fit 1000 data points does not count as an "elegant explanation". We have not condensed the input at all: our 'model' is really equivalent to the input data in complexity, so we have learned (said, proved, contributed, understood) nothing beyond the original data. That is overfitting. We've created a "law" that fits one particular set of data exactly, but any additional data points are completely ignored by our law and are likely to create severe, unacceptable errors. Better to have a more general (elegant, low-parameter) law with a little error on all data than a particular equation that exactly fits a given set of data that explains nothing, predicts
nothing useful, and will change violently if we add one more new data point.

So we need more help. The machine learning community runs into this problem all the time, and one common approach is to separate the data into two sets. One part is used to determine the model (the training data), and the other part is used to check the fit (the test data). This makes it less likely that we have just used the freedoms of our model to match random errors in our data. For one-dimensional problems like ours, a good version of this approach is to fit the model to, say, the central third or first half of the 'x' domain, and see how well it extrapolates to the outer (x,y) points. Most of the models we consider will interpolate pretty well if they actually apply to ('explain') the data, but if we have the wrong law, or have overfit the data, the extrapolation will likely diverge quickly and give big errors for points not in the initial dataset. Having picked the correct law, we could then go back and fit it with the full data set to refine its parameters (but not revolutionize it).

A rule-of-thumb procedure for the sort of data we'll be seeing is, if it is concave upward, first try a quadratic, then integer power laws up to 4, then an exponential, and then various rational power laws > 1 to 3rd or 4th roots (look at close rational fits to the slope on the log-log linear fit). If concave downward, first try a square root, roots to 1/4, (these are power laws) then a logarithmic model, and possibly rational power laws < 1, to 3rd or 4th roots. If the curve has a few maxima and minima, try a polynomial with degree equal to the number of maxima and minima plus 1. If it has a LOT of maxima and minima, we are probably not using the right "toolkit" of models (low-degree polynomials, exponentials, and power laws), and we might need to drag in sinusoids (as in the signal processing techniques later in the course).

When computing the standard error for a power law or exponential or log model obtained using a linear fit to log-remapped data, the fit should be computed in the original space, using the standard error of the data from the actual power law (or exponential or log) model, rather than from the line in the transformed space. This ensures that the different models are compared using the same measure of error.

In any case, for this lab we will try different degrees (N's) of polynomials, and exponential and power laws as well, and quantify the goodness of fit for each using the Analyze() function (next section). You get to look at the results, argue which model is most appropriate, report the coefficients that provide the good fit, and make some graphs as well.

**Error analysis functions**

We need to quantify the goodness of, and to make clear graphical illustrations of, our results.

To do this, it will be useful to have functions that take the parameters of a specified model, and a vector of input X values, and return a vector of the model Y values. The functions

```matlab
function YVals = GenPolyVals(Xvals, N, Coefs)
function YVals = GenExpVals(Xvals, a, k)   % computes a * exp(k * x)
function YVals = GenPowerVals(Xvals, a, p) % computes a * x^p
```

accomplish this for polynomial, exponential, and power-law models.

It will also be useful to have a function that takes a vector of ideal (model) values, and a vector of corresponding measured (data) values, and returns the square root of the sum of squared differences (the standard error). The function

```matlab
function StdError = StdDev(ModelYVals, DataYVals, ModelDOF)
```

accomplishes this. In the simplest case (not applicable to any of the data-fitting in this module) the model is simply that there is one number that explains the data, then that number is the mean of the data. We "expect" all the data to be the mean, but there is some variation. In this simple case, the standard error is equal to the standard deviation of the population, given in Attaway or any statistics texts, with ModelDOF equal to 1. If we fit a line, polynomial, exponential, etc, the DOF will be at least 2.

You will also want to generate plots that show the data points, and curves for one or more models, to illustrate various good and bad fits. When you are comparing extrapolated models to the data, you will probably want to distinguish the data points used to generate the model parameters, from those which the extrapolated model does or does not fit. You will probably find various custom plotting functions useful.
Your writeup should have a table of results for each fit to bring together all the numbers in an easily readable form. Thus Analyze() and power-law fit might look something like:

<table>
<thead>
<tr>
<th>Models</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>p5</th>
<th>exp</th>
<th>power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Error</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

Here the models are polynomial (described by their degrees, here 1 through 5, plus exponential and power-law models) The various e’s will be used to quantify and justify your choice of model, along with other arguments.

In each case you’ll want to give the relevant coefficients for your winning model: the coefficients of the polynomial, or the exponents and constants for the exponential and power-law models.

1. Thermocouple Calibration

Use the Thermo.xls data from a type E thermocouple (a common lab choice, big temperature range, good linearity, cost about $30, only possible downside is the low-voltage output). Find 1st through 4th -order polynomial functions (that is find their coefficients) f() such that µ volts = f(Temp): use only the temperature range (0, 100) degrees. The data looks pretty linear, no? Does the standard error decrease as the degree increases? Would you expect that? Discuss.

Now compute the standard error of your four best-fitting polynomials above, only evaluate the error over the whole range (not just the first 100 degrees) of data values. What do you observe? This is sort of an 'extrapolation polynomial' exercise... fit part of the data and see how well that fit works outside the range. One thing we expect is that the 'correct' model should extrapolate outside its range better than a 'wrong' model that may actually have lower error in the range for which it was constructed.

Repeat the whole exercise above for f() in ohms = f(Temp), using data for the platinum resistance thermometer, PRTdata.xls.

2. Water Flow and Gas Pressure

Flow.xls and Boyle.xls contain classic, actual data from original experiments.

Jean Louis Marie Poiseuille evidently was in competition with Gotthilf Heinrich Ludwig Hagen, and the law governing flow is often called the Hagen-Poiseuille law: it was derived experimentally in 1838.

The history of Boyle's law is complex, involving a correct conjecture by a couple of amateurs, confirmed by Boyle using apparatus built by Hooke (another big name). This was in 1662. The law was independently discovered 8 years later by Edme Mariotte, so it is sometimes (by the French anyway) called the Boyle-Mariotte law.

The Excel file for Boyle has extra columns (which give none-too subtle hints about the form of the law if you've forgotten it...) Treat Boyle's data like the Flow data: get best law, analyze, justify.

Use the data yourselves to find and justify the most elegant laws you can. As we've seen the best fit does not necessarily mean the best law, so you will probably want to go out into the literature to find out what these laws actually are, so you can use your results to verify or raise questions about the laws. Also you like doing this since it gives you references to include in your writeup, which makes you look like a responsible scholarly professional but more importantly can help your grade.

3. Avoiding Higher Math: Primes Again

How are prime numbers distributed amongst the integers? It turns out we (well, they) can prove the Prime Number Theorem; roughly, if you pick an integer near some other big integer N, the chance your number is prime is proportional to 1/ ln(N). That is, the average gap between prime numbers near N is about ln(N). Practically speaking there are lots of them (good for cryptography). By the way, there are thought to be infinitely many primes that are only 2 apart (twin prime conjecture), so the PNT is a statistical result.

Now the proof of the PNT is no joke (check it out on link above), but from our perspective the problem looks like
any other claim that a particular model fits some data. In fact, we can imagine (and implement) a function $P(x)$, which counts the number of primes less than or equal to $x$, and the PNT claims $P(x)$ is proportional to $x/\ln(x)$, or $P(x) \sim x / \ln(x)$.

There are a lot of prime number tables out there. Here for instance are the first 2950 or so primes. Your data set should be some subset of these primes. There's a practical, not intellectual, advantage to using the first $M$ of these primes, where you choose $M$. Imagine you load a vector, like the one above of the primes in order, into Matlab to give you a row vector. Think about that vector and its indices and their relation to $x$ and $P(x)$. Cute, eh? It takes about 9 lines for a function to plot out (in two ways) the relevant vector of primes of length $N$ (for my prime list, $N < 2950$) that tests the PNT hypotheses. CB's code uses only 200 or so primes, but if you want to do more...

You should do the usual: check the standard error for fitting the claimed model and also models of various polynomial degrees.

Also, for fun, fit some subset of your data (beginning? middle?) and compare the standard error for that fit computed for the whole data set. Be careful, the cute rule relating the prime vector and $P(x)$ needs modification if your data doesn't start at the beginning.

4. A Little Computer Science

The 'simple' problem of sorting a vector of numbers, putting it into ascending or descending order, is very common in the real world. Many high-level programming languages have a sort() built-in command, as do Matlab and Excel. Why is it there? We need it often. But also because "they" don't trust "us" to write it. Sorting is a MUCH-studied problem, and different methods have different efficiencies as the list gets longer....some methods just don't 'scale up'. Example: "rearrange the vector elements at random and see if the result is sorted". For $N$ elements there are $N!$ arrangements to check, and that number grows very fast. There are smart sorts as well as dumb ones, but their behavior as $N$ increases is always of interest.

To see some sorts in action, try these: Various Sorts, Heapsort, and even Shellsort.

As stated up top, for us sorting is an experiment: we put in an input $X$ (how long a random vector to sort) and get back an output $Y$ (basically the count of primitive operations) saying 'how much we worked' on the sort. We want to characterize the shape of the $Y(X)$ function, as in all data-modelling problems. We use random numbers, so we can expect variations from any "law", and we are not repeating trials and averaging results (a very common and normally expected practice), so we won't see those variations smoothed out.

Normally a sorting function takes a vector as input and produces a sorted version of the vector as output. For experimental purposes we'll always be sorting a vector of random elements, so let's simplify things a bit.

Write a function bubble_sort($N$) that both creates and sorts (into ascending order) a vector of $N$ random numbers. It works like this:

1. make a vector of $N$ random numbers using $\text{rand}$.
2. compare the first element to the second. If the first is greater. swap them. If not, do nothing.
3. compare the third element to the second. If the 3rd > 2nd, do nothing and go on to fourth element. Otherwise swap them and then compare the new 2nd to the 1st. If new 2nd > 1st, do nothing and go on to fourth element, else swap elements 1 and 2.
4. For all elements (from 4 to $N$), repeat this operation of comparing and swapping the 4th, 5th, etc. element with all the lower elements until it's greater than the element below it.
5. You've now bubbled each element to its proper place and you should return the sorted vector to check that.

For example, with time running left to right, start with [4 3 1 2] and end with [1 2 3 4] as the first 2, the first 3, and finally all four elements are put into sorted order by bubbling the next element into its proper place.

<table>
<thead>
<tr>
<th>index</th>
<th>element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4 3 3 1</td>
</tr>
</tbody>
</table>
Your function should have two for loops, with comparisons happening in the inner one. You'll see that the "do nothing" operation could be implemented with an if command in the inner loop. You may decide to use a break along with the if, which abandons the inner loop and goes on to execute the next case in the outer loop. I don't think it's needed though.

Now add a counter that is initially 0 (set when you enter the function) and that is incremented in the inner loop every time you compare two elements. That number of comparisons is the data we really want: we don't care about a sorted random vector. So now you can change bubble_sort() to return that number of comparisons. Also you're ready to answer these questions, which you should do in your writeup. You don't need to compute to figure these out: just think about what's going on.

1. If all the random numbers happen to be generated in sorted order, (or if you sort the vector 1:N, say), how many comparisons would you get (as a function of N)?

2. If the random numbers happened to be generated reverse (backwards) order (or if you sorted the vector N:-1:1, say) how many comparisons would you get? (as a function of N)?

Now, write a function or script that calls bubble_sort() for values of N from 1 to 100, and stores the returned comparison-counts in a vector. Plot that vector and fit the data with a polynomial. (NOTE that the answer to the 2nd question is a hint on what order of polynomial is appropriate, and you should also do some research if you can't generalize from the above questions to the general situation. Your question is "how complex is bubblesort?").

It would be interesting also to look at the CPU times for this problem. These sorts will probably run too fast for reliable timing, so you'll want to slow them down (or use rather larger lengths, like 100, 200, 300... or even 1000, 2000, 3000... Or, easier, you could put a do-nothing for loop in the inner loop. Then every time you do a comparison you might also execute something like

```matlab
for k = 1:1000 % or 100 or 10000, I don't know...
    x = sqrt(exp(sin(k)));
end;
```

That should add a constant time to each comparison and so give you some more precise numbers. Use tic, toc to find the time for the function call and return that time, not the comparison count.

Again, make a script or function that runs the sort for a range of values of N and saves them in a vector. Plot the CPU times against N and fit that data with a polynomial.

Again again, use some subset of your data to find the 'best' 1st-order, 2nd-order,... 4th-order, exponential, power-law models for the subset, and then compute the standard error for the full data set to see which model(s) hold(s) up when extrapolated.

**What to Hand In**

See the [Universal Hand-In Guide](#).

Briefly, for the Code component of this assignment, you'll need a .zip (not .rar) archive with code files and README.

Submit the writeup as a single, non-zipped .PDF file. Submit before the drop-dead date for any credit and before the due date for partial-to-full (or extra!) credit.

Check immediately to see BB got what you sent!