Fitting Experimental Data

Chris Brown, Randal Nelson

July 2010, March 2011

Contents

1 Introduction and Motivation 2

2 Polynomial Fitting: The Approach 4

2.1 Fitting as a Linear System . . . 4

2.2 Using More Data: Least Squares . . . 5

2.3 Quantifying a Fit . . . 7

2.3.1 Introduction: Mean and Variance . . . 8

2.3.2 Standard Error of a Fit . . . 9

2.3.3 Practical Approach to Choosing a Model . . . 10

2.4 Noise . . . 11

3 Some Variations 12

3.1 Exponential Functions . . . 13

3.2 Power Law Functions . . . 14

3.3 Interpolation Polynomial . . . 15

3.4 Robust Fitting . . . 15

4 Calculators and Short Cuts: Matlab and Excel 16

5 References and Further Reading 17

A Fun with Convolution and the Central Limit Theorem 17

1 Introduction and Motivation

There are no new mathematical ideas here. We shall just use what you already know about solving systems of linear equations, which for us means you'll use the Gaussian Elimination program you've written.

Data analysis is to let us make sense of data we (or someone else) collected in an experiment. In what follows, we usually assume the measured variable is *dependent* on the other, which is *independent*ly accurately known. We shall only deal with the (common) one-dimensional case in which the dependent variable is a function of only one independent variable. Our goal is to detect and quantify the relationship between the two variables.

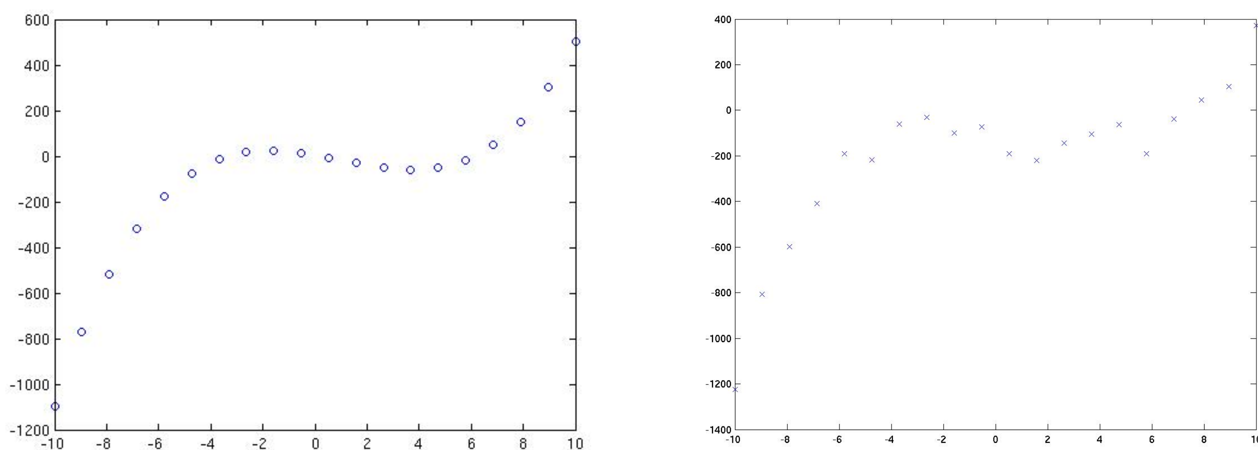

Figure 1: **Left: Points from a polynomial "law" (in fact $y = x^3 - 3x^2 - 20x + 5$). Right: what rather noisy experimental data might look like, taken from a measurement of a value obeying that law.**

In Fig. 1 we see a plot of a particular 3rd-degree *constant-coefficient polynomial* in x (the normal quadratic equation is a 2nd-degree polynomial and the equation for a straight line is 1st-degree: if the word polynomial is new to you, look it up).

Suppose there is a natural law (from physics, say) predicting the behavior of a quantity (voltage? pressure?) as a function of another quantity (time? temperature?). Say the left-hand plot of Fig. 1 shows how the law works. In the laboratory we use several values of the independent variable and measure and record corresponding values of the dependent variable. If our technique is lousy or for any number of better reasons, we might get data that looks like the right-hand plot.

Given this intellectual set-up we have several possible problems and opportunities.

1. We have no idea of the law but we have the data. Can we explain the data with an elegant relation? Given that, does the relation suggest or extend a basic theory (physics? chemistry?) Given that, can the relation be used to explain, predict, and control nature? If so, we've discovered a law, which people will doubtless name after us, and we join the ranks of Boyle, Newton, Ohm, Moore, Kepler, Poiseuille...

2. For reasons of theory we expect the law to be of a particular form (say a polynomial like the speed of a falling object through time), or exponential (like bacterial growth or temperature as a hot object cools), or a power law (like the force of gravity as a function of distance, or the volume that a quantity of gas takes up as a function of pressure). But we don't know the coefficients that determine the exact law. For instance, we might know that $v = gt^2 + c$ but for this particular falling-object experiment we don't know g or c and we want to find them.

3. Even if we do believe the law and the constant(s) are 'known', we may be able, using our data, to add to the current state of knowledge (our basic goal as scientists) by improving their accuracy. Part of scientific progress is improving this knowledge, even for such things as the gravitational constant.

4. The 'law' may just be a simple matter of applying existing theory to our particular problem, and we look for its manifestation in our experiment. A good example is predicting what the brightness of a star would do as a planet occults it (orbits in front of it). A small effect even up close, and very small observed from earth. However, if we know what we are looking for we are in a position to find it even in noisy data (see below).

5. We have no reason to expect there is an elegant law governing the relation between independent and dependent variables. There presumably is one unless "randomness" is involved in the underlying theory, but that's beyond the scope of this discussion. But the true behavior is governed by causes and relations that are messy, complex, and maybe practically impossible to predict. It might depend on the individual component we are measuring as well as on its type, as in sensor calibration. What we want to do is predict the dependent variables for values of the independent variable between those of our data without claiming to know any laws at all.

6. In all these scenaria, the data is by definition "noisy"; it is inaccurate due to uncontrollable (or unknown) processes acting on the dependent variable or on our sensor or on both. How do we define and find the best solution to our problems in the presence of noise?

Further to item 4 above, see Fig. 2 and the references.

Now lots is known about these basic problems, but we'll only explore one solution here. We'll avoid mathematics as much as possible, use what we already know, grasp one basic approach, and show how to exploit it in some common situations.

An important topic we are avoiding here is statistics. If we see a dependent variable rising as the independent variable rises, we can say the two are *correlated*, and we can quantify how much correlation there is. The correlation exists independent of any laws and neither assumes nor needs any underlying model of the relation. Statistical analysis is a huge field, but we're not going there. (There are some simple observations about noise in Section 2.4).

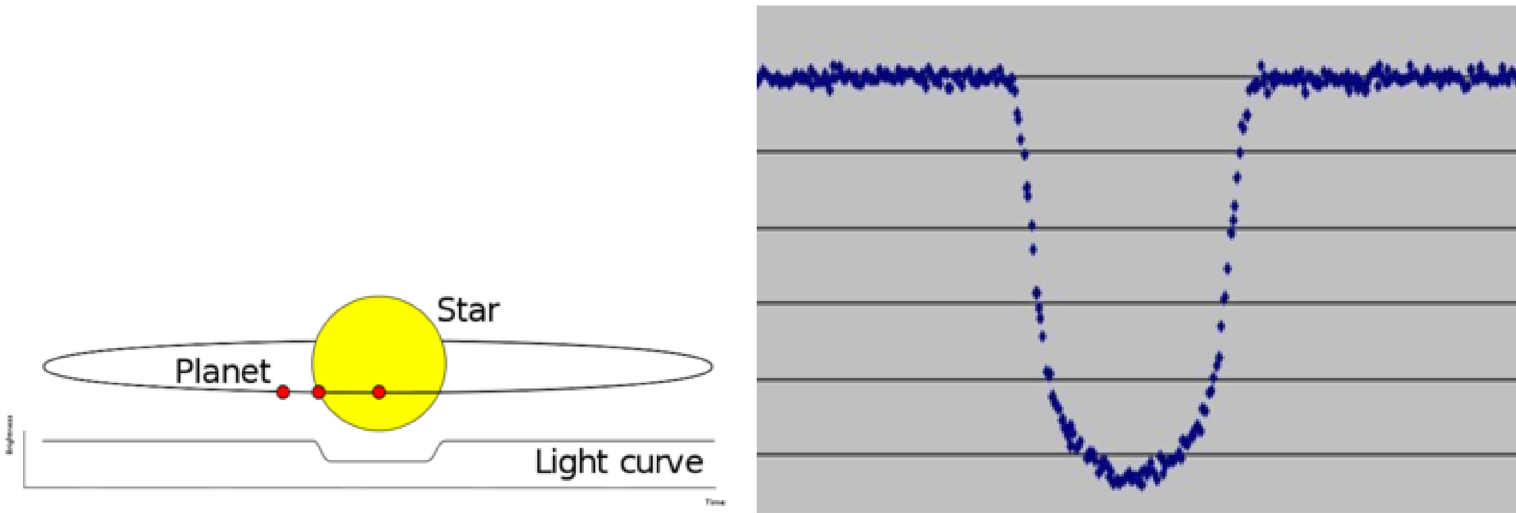

Figure 2: **Left: Predicting how planetary occultation affects a star's brightness is simple geometry. Finding it in real data is the problem (Right).**

2 Polynomial Fitting: The Approach

2.1 Fitting as a Linear System

Our approach is known as regression analysis, curve-fitting, least-squares, or sometimes trend-lines.

Say we take some data: it's a vector of (x_i, y_i) pairs, where x is the independent variable, y the dependent. If, for instance, we have reason to expect that the law governing the process that produced this data is a polynomial equation of order n, then we need to find the coefficients c_i such that then for every (x_i, y_i) pair (ideally)

$$y_i = c_0 + c_1 x_i + c_2 x_i^2 + \cdots + c_{n-1} x_i^{n-1} + c_n x_i^n.$$

This is an equation in $n + 1$ unknowns. It has constant coefficients c_i. It arises from one (x, y) data point and our hypothesis about the order of the polynomial governing the process. Our job now is to find values for those coefficients c_i.

We know that if we have $n + 1$ unknowns we need $n + 1$ equations, each arising from an (x, y), (independent, dependent) pair. If we know the law or the model for the data is nth order, then what we're looking for is the unknowns in the above equation; the coefficients c_i.

We know that to solve for the $n + 1$ coefficients of an nth -order equation, we need $n + 1$ equations (e.g. two points determine a line, three a quadratic curve, etc.). But we can make an equation like the above for every (x, y) data point we have. So we need to collect enough (x, y) data points to *fit the model to the data*. If our model is an n-th degree polynomial, we need $n + 1$ data points (hence equations).

Here the model is "an n-th order polynomial", the data is the (x, y) pairs from the experiment, and the fit is the coefficients c_i that determine the n-th order polynomial that (we

hope) predicts or explains the variation of our dependent effect (y) with some other variable (x).

We're really done now. We've reduced the problem of finding an nth order polynomial to solving a system of linear equations, for which we use our Gaussian elimination program.

Putting this into matrix form, we get:

$$\begin{vmatrix} 1 & x_0 & x_0^2 & x_0^3 & ... & x_0^n \\ 1 & x_1 & x_1^2 & x_1^3 & ... & x_1^n \\ .. & .. & .. & .. & ... & .. \\ 1 & x_n & x_n^2 & x_n^3 & ... & x_n^n \end{vmatrix} \cdot \begin{vmatrix} c_0 \\ c_1 \\ .. \\ c_n \end{vmatrix} = \begin{vmatrix} y_0 \\ y_1 \\ .. \\ y_n \end{vmatrix}$$

or, for short,

$$y = Xc,$$

with y the dependent variable vector, c the coefficient vector we seek, and X is known as a *Vandermonde matrix*. Solving this system gets the desired values for the coefficients of our model and we're done.

Matlab factoid: We don't need to know this or use it, but Matlab can represent polynomials and interpret them (as in the `roots` command that finds the roots of a polynomial.) Matlab represents a polynomial, as we do here, by the vector of its coefficients. Our treatment above is standard and what you see in the literature – in Matlab, however, the coefficients are in *descending* order in powers of x. So [1 2 3] represents $x^2 + 2x + 3$.

Garbage in, garbage out? We might worry about those x_i^ns; the exponentiation will amplify any noise. However, x is the *independent* variable here and we seem entitled to assume it is accurate.

2.2 Using More Data: Least Squares

Only one cubic curve goes through four points. If the points are affected by noise, we get the wrong curve. Our intuition is that if we had more points, we could 'take an average' or something and get a better estimate (Fig. 3.)

Looking back at the formalization of the last section, we're talking about putting more rows in the Vandermonde matrix, one for each data point. Then somehow we need to solve this "overdetermined" system (e.g. we can put a line through any 2 points but not through any three points!).

It turns out, rather wonderfully, that we are well on the way with our current straightforward approach to finding a very useful form of answer. We formalize our 'averaging' intuition as "least-squared error" fitting, and we solve it as a classic calculus minimization problem: get the derivative of what we're trying to minimize and find the values that set it to zero.

We won't go into details here, but there's a very straightforward derivation at
`http://mathworld.wolfram.com/LeastSquaresFittingPolynomial.html`
that says it all succinctly and clearly.

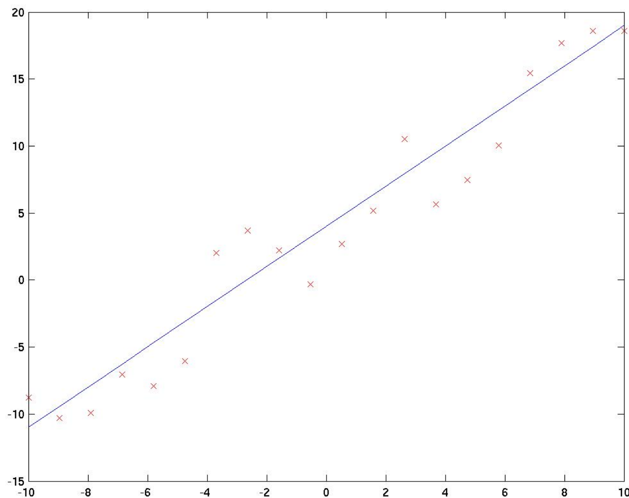

Figure 3: **Fitting a straight line (linear regression). We define the best fit as the line that minimizes the sum of the squared vertical distances to the line from the data points. Each distance is a *residual*.**

For example, least-squared-error fitting wants to find the straight line in Fig. 3 that minimizes the sum of the squared residuals (defined in the caption). That intuitively seems like a reasonable way to formalize the problem (but you could be praised for wondering "why not the absolute value of the residuals", or "why not the perpendicular distance to the line?").

The punch line (and one answer to your first question above) is that by pursuing a completely different-seeming problem (minimization of the sum of the squared values of all the residuals) we wind up quickly with a solution *in terms of things we've already done.* Instead of (basically) getting the inverse of a square Vandermonde matrix and multiplying by it, we can easily get a so-called *pseudo-inverse* of our non-square Vandermonde matrix: that is, instead of

$$c = X^{-1}y$$

for vectors c and y and square matrix X, the least-squares derivation leads to this:

$$c = [(X^TX)^{-1}X^T]y,$$

where X^T is the transpose of the Vandermonde matrix. You can verify that the matrix in [] has the appropriate dimensions.

The pseudo-inverse is of interest because it extends the concept of matrix inverse and because it allows us to express the least-squared-error solution in a familiar form. There is much mathematical lore about generalized inverses (check out "generalized inverse" and "Moore-Penrose pseudoinverse" in Wikipedia).

Elegant as the pseudo-inverse is, we can simplify the situation further to facilitate computation. Multiply both sides of
$c = [(X^TX)^{-1}X^T]y$ on the left by (X^TX), which cancels with its inverse and leaves

$(X^TX)c = X^Ty$. But the right hand side is a vector and we know both X and y, so we can compute a new vector $y' = X^Ty$. Also, the (X^TX) on the left side is a matrix product a matrix we know and its transpose, so we can write $A = (X^TX)$. Thus we derive, using the new A and y', $Ac = y'$, which should be familiar. It's the general form of a system of linear equations with unknown vector c (the coefficients we're seeking), with the Vandermonde matrix as the "coefficient" matrix in the system and the transformed y' vector as the constant vector. So, after a little matrix multiplication and renaming, all we need is Gaussian Elimination to solve the Least Squared Error estimation problem for any number of input data points for polynomial models.

To implement all this in Matlab, you could use your Gaussian elimination routine (that would be impressive, actually... you built a tool now demonstrated to be useful in later work). Or you could use Matlab's built-in vectorized "divides" operator: if $Ax = b$ then in Matlab

```
x = A \ b;
```

solves the system. Note that `x = inv(A)* b` would also work but we practical numerical analysts avoid computing the inverse when possible: a successful and more accurate answer is more likely using special system-solving methods.

2.3 Quantifying a Fit

Given the same (X,Y) data we can fit it with any model we choose. Our job is often to find the best model. Fig. 4 shows a linear and an exponential fit to the same data. The only difference was our choice of model. Both our eyes and the size the residuals indicate that the exponential fit is better (whether it is better than a quadratic or cubic remains to be seen!).

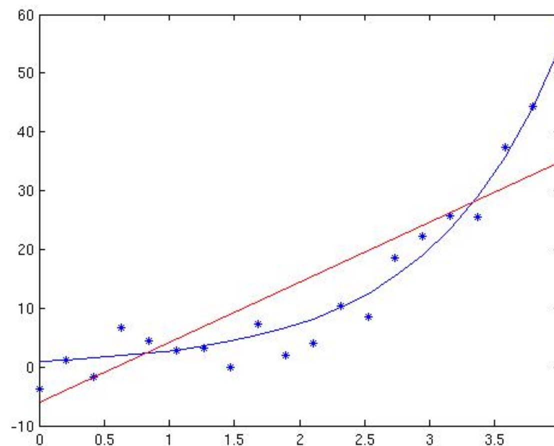

Figure 4: **A linear and an exponential model fit to the same X Y data.**

One generally accepted way to quantify the goodness (badness, actually) of a fit is its *standard error*. Its form is related to the standard deviation (square root of the variance)

we have seen before. It can be related rather easily to a *confidence interval* that has the nice intuitive interpretation its name implies, and reflects the fact that we are surer about our answers if we use more data. But that's just a little too much statistics for us.

In estimating statistical quantities like the standard error of a fit, the *degrees of freedom (DOF)* in an estimate is the number of data values we start with, minus the number of other estimated statistical quantities we have to use in order to compute the estimate. The DOF appears in rules for estimating population statistics from a number of samples, in particular it appears in the standard error formula.

2.3.1 Introduction: Mean and Variance

Let's warm up with a first example of DOFs that will explain Attaway's formula for variance (first formula in section 12.1.2).

We remember what the mean (average) is. For a data set X with N values of mean M, its *second moment* W is

$$W = \frac{\sum (X_i - M)^2}{N}$$

.

This second moment is one version of the statistical variance of the data, so let's look at the formula as programmers for a minute.

The above mathematical definition of W has a computational disadvantage: we need to look at all the data once to compute the mean M, and then look at it all over again to subtract each data point from M. But it turns out that if we also keep the sum of the x_i^2 when we are summing the x_i to calculate the mean, we can calculate the mean and variance as we read the data the first time (thus we never need to store it or read it more than once). Let E be the "expectation" or "mean" function (sum divided by N): so the expected value of any particular x_i is the mean: $E[x_i] = M$. The expectation of the sum is the sum of expectations, so

$$\begin{aligned} W &= E[(x_i - M)^2] \\ &= E[x_i^2 - 2Mx_i + M^2] \\ &= E[x_i^2] - 2ME[x_i] + M^2 \\ &= E[x_i^2] - 2M^2 + M^2 \\ &= E[x_i^2] - M^2 \\ &= E[x_i^2] - E[x_i]^2 \end{aligned}$$

which is the standard and accepted way to compute the second moment, and often used to compute the "variance". But we need to be a little careful...read on.

The second moment is the *sample variance* (of the *data itself*). It describes the "physical properties" of variability in the points: it is the same formula we use for the second moment of inertia in a physical system. It does not pretend to "look beyond" the data.

On the other hand, it is more usual to think of the data as a sample from a large population. The sample is what we can practically obtain, or we convince ourselves it's all we need to obtain, from the general population we want to describe.

In this case, the mean and variance are meant to describe properties *of the whole population*, which is subtly a different problem. We want to estimate the *unbiased population variance* (of the actual population of events that is being sampled over.) That is we want to prove our estimate of variance is neither systematically too high or too low.

We **know** the sample average and variance of the data – the latter is given by the equation for W above. What can we say about the estimated mean and variance of the population's distribution?

Trust your statistics prof. here, but a quickie version goes like this. First, it turns out that the estimated population mean is the sample mean — so far so good.

Now the bigger the sample, the better the estimate for the variance; that's why we divide (normalize) by N in the formula for W. If we have N data points in the sample, that is N independent numbers (if our sampling technique is good). All of them are used to compute the mean, which is thus dependent on them all. So with the mean we have $N + 1$ numbers now, any N of which will give us the remaining one. But in the first equation for W above, all of those numbers appear – all the x_i and the mean too, which we know depends on the x_i. So we're using the mean AND the N data points. This is equivalent to double-dipping, sort of like counting some number twice.

The point is that if we are estimating the population variance, we need the mean. Once we have that, we only have $N - 1$ more independent numbers (aka Degrees of Freedom) in our sample, and so we only get to normalize by *that*, namely $N-1$. This explains the $N-1$ as the traditional normalizing factor in variance definitions, including those in Attaway 12.1.2 and in Matlab's built-in `var, std` functions. Using N in this case would bias the estimate of variance to be too low. Clearly this effect is small for large N.

So here is our formula for the estimated variance V and the standard deviation s of the *population*, given the mean M and the data vector X of length N:

$$V = s^2 = \frac{\sum(X_i - M)^2}{N-1}.$$

We say that the variance estimate has $N - 1$ *degrees of freedom.*

2.3.2 Standard Error of a Fit

Now in polynomial model-fitting, the fitting process estimates $n + 1$ parameters for an nth degree polynomial. *Given them*, we can calculate the standard error of the fit as follows (it looks very like the square root of a variance-like number).

$$S = \sqrt{\frac{\sum(R_i)^2}{N-p}}.$$

Here we sum over all the *residuals*. The residual of a data point (x_i, y_i) is $(y_i - P(x_i))$, where $P()$ is the $(p - 1)$-degree polynomial model we have fit with p parameters: for each term

(say the ith) in the sum we need to substitute x_i, x_i^2, etc as needed into $P()$ and evaluate the result. Notice that the more parameters we must estimate to get our model, the less evidence is available to estimate the error of fit.

For example, if our least squares fit is a cubic that predicts $y = 3x^3 + 4x^2 - 5x + 20$ and we get an (x_i, y_i) data point $(2, 42)$, then the model predicts that the y_i value should be 2*8 + 4*4 - 2 + 20 = 50, and thus the residual is -8 (we don't care about the sign, we're going to square it anyway).

Here, with 1000 data points, we'll normalize by 1000-4 = 996. Generally for a p-parameter system we normalize by $(N - p)$ for N data points, with $p = n+1$ for an nth order polynomial.)

Back to Matlab: `var, std` have an extra parameter to tell it to normalize by N or N-1. BUT Matlab does not deal with the general p-parameter case, so to get the right standard error of fit we have to write it ourselves.

2.3.3 Practical Approach to Choosing a Model

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 here we have chosen the square root of the mean squared deviation from the model (aka the standard error). 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 are likely to create severe, unacceptable errors.

It's 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. The latter explains nothing, predicts nothing useful, and will change violently if we add even one more new data point.

Errors, elegance, what to do? 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 paramete (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 (like 1/2, 1/3, 1/4, the square, cube, 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, expontials, 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 exponen tial 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.

2.4 Noise

"Noise" is any signal that interferes with the one we want. We can (try to) divide noise up into systematic noise and random noise. The former is due to some predictable process, the latter is due to some process we might know statistical facts about but cannot predict.

For instance, systematic noise could be a 'bias', where readings are just off by a constant everywhere. Or it could be "60-cycle hum", in which our experiment's wires are picking up the 60 Hz electromagnetic field caused by the AC current in the building. Or we could

even say the deviations of our sensor from linear behavior, which we could address with interpolation polynomials (Sec. 3.3), are systematic noise.

Random noise is often described statistically: E.g.: It has *mean* zero, or it is *distributed according to a Gaussian or Poisson distribution*, etc. Noise can either be independent of the signal (like background noise) or dependent on it (often the noise increases as the signal does). There could be more or fewer outliers (Sec. 3.4), etc.

The distribution of noise values is often taken to be Gaussian (also called a *normal distribution*, which is very mathematically convenient, but also it's more than that. The *central limit theorem* (CLT) of statistics states that the mean (thus sum) of a sufficiently large number of independent random variables (maybe different independent noise processes), each with finite mean and variance, *will be approximately normally distributed (!).* A powerful result (Fig. 5, and see the Appendix).

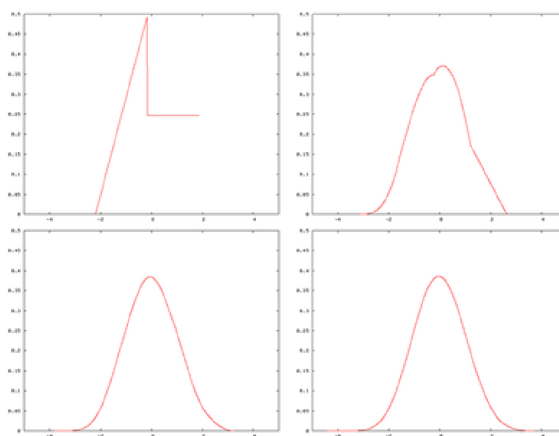

Figure 5: **The Central Limit Theorem: Three successive summations of a very non-Gaussian distribution.**

Finally and generally, knowledge about our experiments is power: always pays to know about the sources of noise in our data because then we are better prepared to explain, predict, and possibly control it.

3 Some Variations

We have seen how to fit an nth-degree polynomial to data, either exactly with just enough data or in a least-squares sense with more data.

We ideally should have some grasp of the physical processes producing our data: what law is likely to be at work? Getting the right law is important. We can fit any data with any model: our process doesn't care. However, the philosophical and practical status of a description based on an inappropriate law is very conjectural. Practically, serious deviations often appear if the wrong law is extrapolated beyond the data...e.g. we can fit the $sin(x)$ between 0.0 and 0.1 radians well by a straight line of slope one. The line goes from 0 to .1, the sine from 0 to 0.0998. Does this mean the sine of 100π is about 314? No, it's 0.

In the worst case we can plot the data and look at it, with the principles of Section 3 in mind.

For interpolation polynomials especially and some other cases, another move of desperation is to try fitting the data with polynomials of increasing order. Must minimize order to minimize weirdness arising from extrapolation, but may have to increase order to get a 'better' fit. An Nth-degree polynomial can fit any N+1 points but usually this is *over-fitting*, and the resulting fit has no predictive or generalization properties (can't interpolate, can't extrapolate).

We have a bit of a problem in general, though. Many natural laws are not polynomials at all. Now what?

3.1 Exponential Functions

If we think our process is like bacterial populations growing or the declining radiation of an isotope as it decays or the temperature of a cooling body, it could be obeying a two-parameter law (here, the parameters are a, t) like this:

$$y(t) = ae^{kt}.$$

With negative k, y decays to zero and with positive k it explodes (Fig. 6).

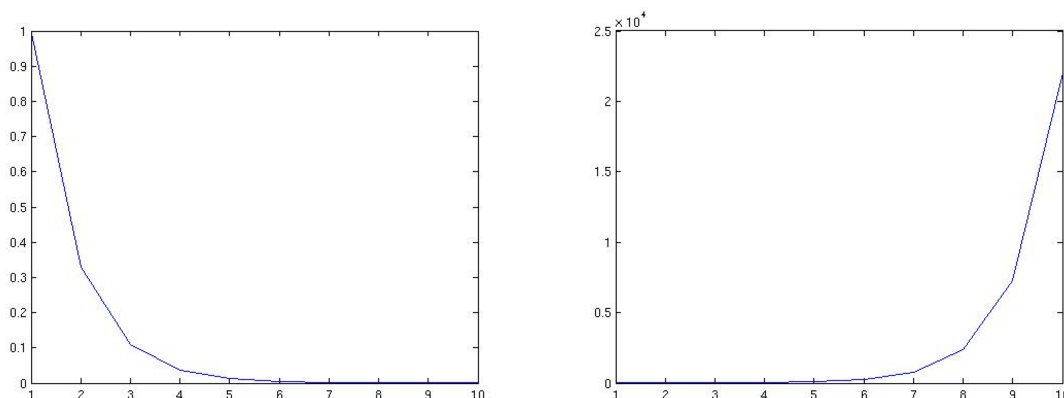

Figure 6: **Exponential decay and growth.**

Remember semi-log graph paper? It plots the logarithm of y against t. If we take the log of the RHS of our exponential law we get

$$y(t) = \ln(a) + kt,$$

Which is not just a polynomial: it's a line (Fig 7).

Thus if we suspect an exponential law, take the log of the data values and see if the log data points fall into (and can be fit best with) a straight line.

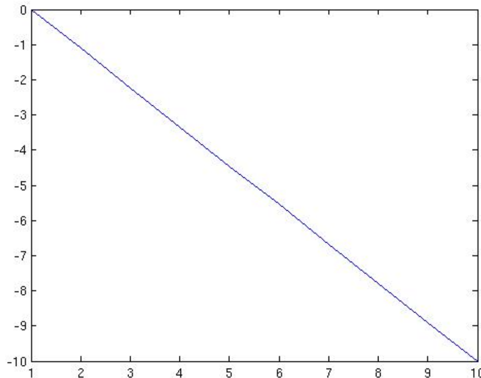

Figure 7: **We can use polynomial tools on the logarithm of exponential data (here of the previous decay function).**

A law of the form

$$y(t) = b + ae^{kt}$$

is perfectly reasonable but our simple log-taking approach doesn't work. Possibly a preliminary step would be to estimate b as the value of the first y_i data point?

3.2 Power Law Functions

If we think our process should have $y = 0$ for $x = 0$ and y increases without bound along with x, or that y starts at some value for $x = 0$ and decreases to 0 as x increases, it could indeed be a general polynomial but we might get lucky and it is obeying a *power law*. Two familiar examples are Boyle's law, which says that the volume of a gas in a system is inversely proportional to its pressure:

$$V = k/P,$$

for some constant k. This is a power law:

$$V = kP^{-1}$$

Newton's law of gravity is:

$$F = G\frac{m_1 m_2}{r^2},$$

with F the force, G the gravitational constant, r the distance between bodies of mass m_1 and m_2: this is an inverse square law. Under certain circumstances, the rate of flow of water through a pipe is proportional to the fourth power of the diameter of the pipe.

Generally, a power law is a two-parameter law (here in k, m) that looks like

$$y(t) = kt^m$$

for some positive or negative exponent m and constant k. Knowing (or hoping) that we only need the single coefficient and exponent keeps our fit more elegant (if it works).

Here we take the log of both sides of the equation:

$$\log(y(t)) = m \log t + \log k,$$

so now we have a linear relationship between the log of the dependent and the log of the independent variables.

Again we're only trying to fit a straight line. Thus if we suspect a power law, take the log of the x and y values and see if they fall into (and can be fit best with) a straight line.

Again again, if we suspect a third parameter that is added to the right hand side to offset the curve, we can estimate that offset somehow, apply it to the data and proceed as above, OR we can look for more powerful (nonlinear) fitting methods.

3.3 Interpolation Polynomial

It often happens that we have no idea about the law, or laws, governing our data: how they interact, which ones we don't know about or can't model, and so forth. A good example is the issue of *calibration*, in which we have a measuring device whose output correlates with what we are trying to measure but in a non-linear way that is hard to predict or understand: for a CCD sensor it might look like Fig. 8, and the neighboring CCD sensor in the mega-pixel array in your cellphone's camera may have a related but different response. Likewise for thermocouples, strain gauges, speedometers, fuel guages, or any real sensor: we must be able to interpret their output accurately.

What we can do is start with a plot of the response data, figure out what range of inputs we'll be working with, and then fit that range of data with some function or other (as discussed above). What we will have then is an *interpolation polynomial* $f(t)$ that, over the range of interest, will give us a reasonable value for our sensor's output for any t in the range of interest. It smoothly interpolates between our known calibration data points using a locally-appropriate function. For a different part of the sensor's input range we may want a different interpolation polynomial.

Thus an interpolation polynomial is a bit of a second-class citizen: it is often an *ad-hoc*, unprincipled fit of a subset of our data and has the connotation of not being based in physical law.

3.4 Robust Fitting

The Garbage In, Garbage Out problem can arise with data points that are politely called "outliers". They are points, often isolated, that are wildly outside expectations for a particular model. They could be caused by temporary sensor failure, electric glitch, or other vagaries of nature (or by a true part of the process that's messing up our theory and which

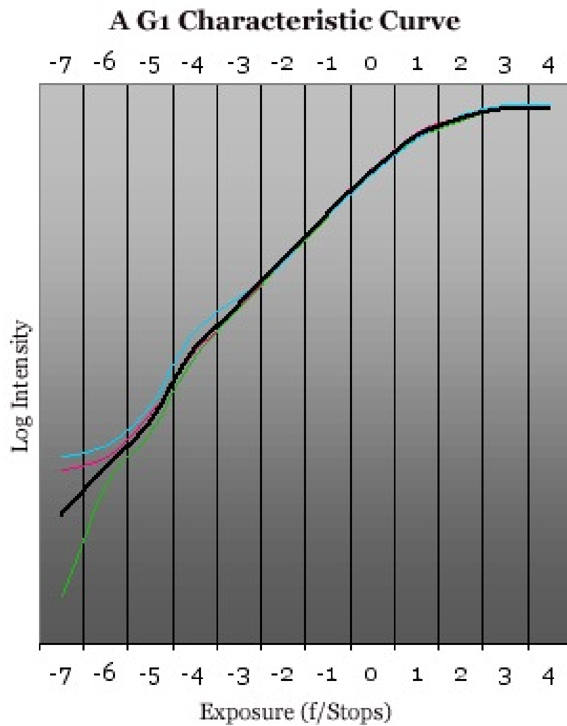

Figure 8: **Response of one type of CCD (digital camera sensor) to increased light flux: As is typical of real sensors, they are (sort of) linear over a range at best, with nonlinear rolloff and saturation effects.**

we'd love to ignore — careful over ethical issues here!). Since we are counting the sum of *squared residuals*, large residuals have large-squared influence (Fig 9).

There is a whole field of *robust statistics* that deals with outliers. We want a fit that is robust against outliers. For now, just notice that if we have enough "good data" we can eliminate "bad data" by noticing the residual of each data point and if it offends us, remove that data point and run the fit again. This process can be done iteratively, working down from the worst residuals. The resulting "cherry picked" data could be open to misinterpretation, and our cherry-picking could be labelled, at worst, fraud. So it would pay us to have a good grasp of the statistical issues before pitching data that doesn't "look right" (fit our preconceptions or current theory).

Cute idea: if we don't use an averaging approach (the fit minimizes the sum or mean of the squared error) but use a voting approach (the fit maximizes the number of points it goes through or 'explains'), we become completely indifferent to outliers. The fit we get explains the most data. This voting approach could be called *mode-based fitting*.

4 Calculators and Short Cuts: Matlab and Excel

Our text, Attaway Ch. 14.1, outlines the above material and also shows built-in Matlab commands like `polyfit, polyval`, which use the (easy) Matlab representation of poly-

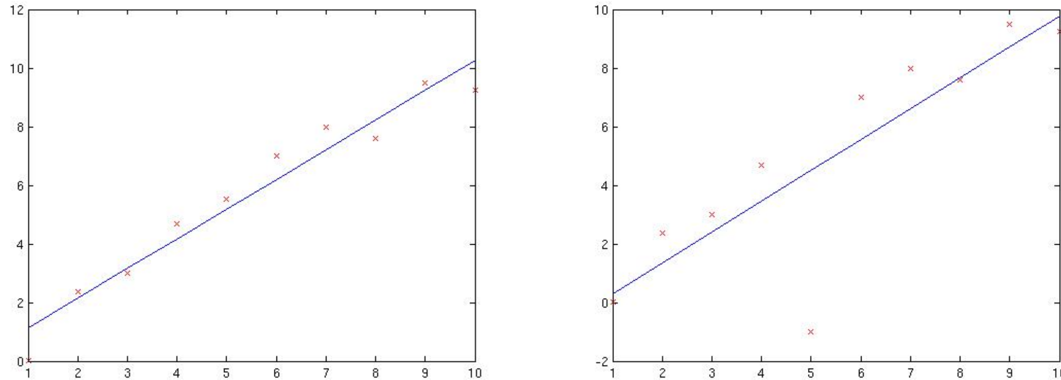

Figure 9: **Least-squared fits to data: Right has all the same data but for one outlier. The least-squares fit (being below almost all the data) is arguably no longer what we want.**

nomials mentioned in Section 2.1. Likewise you'll probably be expected some time in the future to use Excel, which has functions do the same thing. Attaway also takes you through the mathematics of least-square line fitting (linear regression). It's just a specialized version of our general treatment above using the pseudo-inverse for least-squares fitting of a polynomial.

What you use in the future (near-term, in the LVDT and LED projects later in this course) is up to you. It would be in the spirit of this course to keep your model-fitting code and use it later. Such re-use sounds efficient, but actually doing it is an interesting discipline, and is guaranteed to give you respect for extendable and flexible coding style, good documentation and comments.

5 References and Further Reading

`http://mathworld.wolfram.com/LeastSquaresFittingPolynomial.html`
`http://en.wikipedia.org/wiki/Methods_of_detecting_extrasolar_planets`, (section: Transit Method).
`http://www.ucar.edu/news/people/Brown/` (7 years out of date but nice picture).
`http://onlinestatbook.com/chapter8/df.html` (Clear explanation of population variance).

A Fun with Convolution and the Central Limit Theorem

There's only one integral that appears in this course, and it appears again in the signal processing component. It appears here in a rather natural way, so this little section is meant to introduce the convolution operation, which is nothing but a 'running, sliding, backwards dot-product'. It also illustrates the central limit theorem (CLT) doing its thing.

Consider the CLT in this context: we have two fair dice and we want to compute how many ways we can roll a 2, 3, ..., 11, 12. Let's consider the function measuring how many ways one die can yield 1 – 6 in one toss: the number of ways to get 1 – 6 is: 1 1 1 1 1 1, in order. Only one way for 2 dice (die 1, die 2) to yield two 1, or 'snake eyes': (1 1). Two ways to throw 3: (1 2) (2 1). Three for 4: (1 3) (2 2) (3 1). So the count for 2–12 looks like 1 2 3 4 5 6 5 4 3 2 1, which is the function showing how many ways to get all sums of yields from two dice in one toss.

Now since we're young and alert and love to put things together, we notice immediately that the triangular (sum of two dice) function is the auto-convolution of the flat (yield of one die) function. Coincidence? Sure, right... Let s be the yield of a toss of two dice, the $N_2(.)$ function count the number of ways that a yield can be achieved in a two-dice toss and $N_1(.)$ count the ways in a one-die toss, we see (mathematicizing the observations above)

$$N_2(s) = \sum_x N_1(x) \cdot N_1(s - x).$$

This is called the *convolution operation*. Nothing changes if we count probabilities instead of "ways to yield", and go to continuous distributions, so we get 160's only integral, the famous convolution integral:

$$N_2(s) = \int_x N_1(x) \cdot N_1(s - x) dx.$$

It's easy to see that sliding the pdf by itself reversed and doing the dot product counts the probability of all the ways the sum of the two is s. In our case if we compute $N_3(s)$ from $N_2(.)$ and $N_1(.)$, we get a quadratic function, which goes to cubic next time, and quickly looks Gaussian.

Fig. 10 was produced by plotting the results of Matlab's `conv()` built-in function:

```
Fig. dieprob = [1 1 1 1 1 1];
twodice = conv(dieprob, dieprob);
fourdice = conv(twodice,twodice);
```

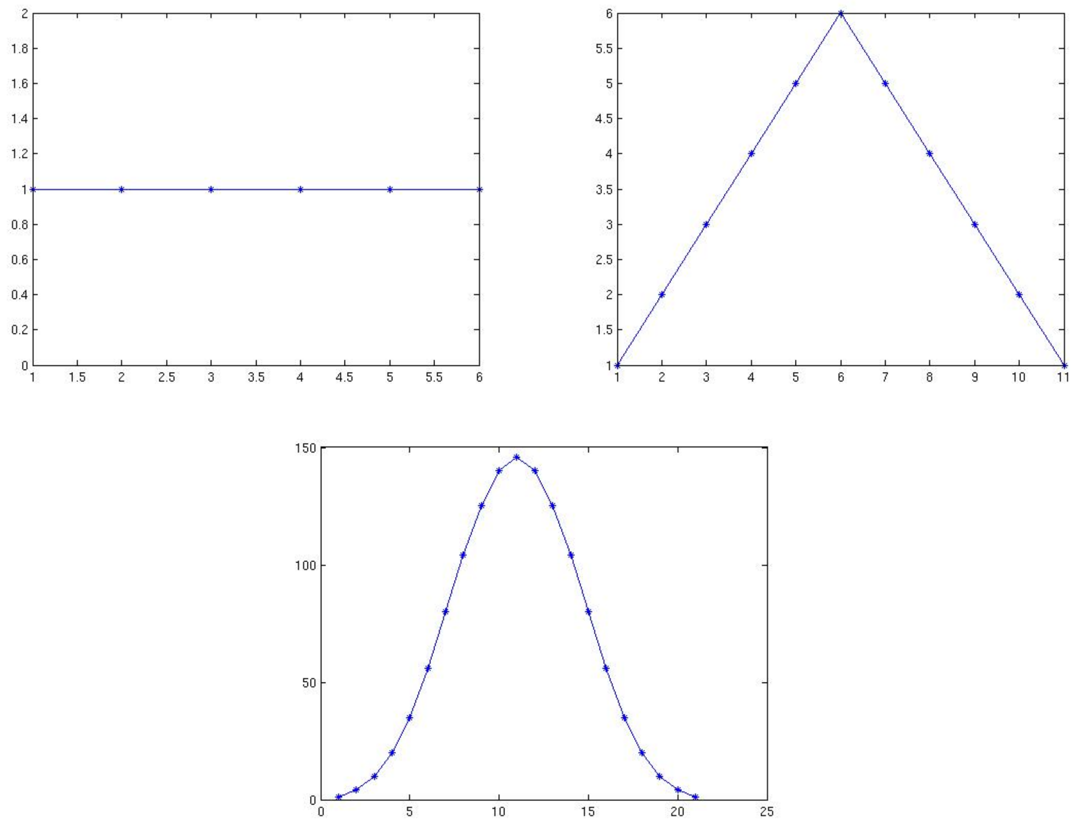

Figure 10: **Convolutions (proportional to probability density functions) for the values yielded by the sum of the values showing when tossing one, two, and four dice (see text). (The values may need shifting a bit).**