4. Fitting models to data

By now you have seen a variety of examples in which the answer to some problem is given by the solution to a system of linear equations. In each case we relied upon the fact that a certain system of equations did have a solution in all of the situations of interest.

In this case study we turn to a situation where there are many more equations than variables, with the result that there is almost certainly no exact solution at all. Instead we look for a solution that comes closest to solving the equations approximately. This approach is common where various mathematical *models* are to be tested for their ability to explain relationships among uncertain data measurements. Each model gives rise to one equation per measurement. Thus the large number of measurements together with the approximate nature of the data make the solution of any exact equation systems impossible. Instead, each model is judged on the basis of the best approximate solution that can be found to the equations that it implies. The computation of the approximate solution can be viewed as *fitting the model* to the data.

This case study gives two examples in which models are fit by means of the widely-used "least squares" criterion for approximate solution of the equations. The first seeks a model for the effect of gravity on a falling object, while the second models the computation time for inverting a matrix.

An example from physics. To measure the force of gravity experimentally, you can throw an object out of your window and have someone take a series of photos (or a video) of the object's trajectory. By analyzing the resulting pictures, you can compile a table of observations of height versus time. Using a camera having a motor drive that takes four photos a second, for example, you might come up with something like this:

TIME *(seconds)*	HEIGHT *(meters)*
0.25	11.75
0.50	13.00
0.75	13.50
1.00	13.25
1.25	12.50
1.50	11.50
1.75	9.50
2.00	7.00
2.25	3.75
2.50	0.00

You know (or soon will learn), however, that the definitions of velocity and acceleration imply a relationship between height of the rock and time of the form

$$h = -\tfrac{1}{2}x_2t^2 - x_1t + x_0$$

where t is the elapsed time, h is the height at that time, and

x_2 is the acceleration of gravity,
x_1 is the rock's initial velocity (toward the ground),
x_0 is your window's height.

You can think of this as a *model* of the effect of gravity on the stone, given perfect conditions and exact observations. For any elapsed time t, the model *predicts* a height h. To make best use of the model, you want to choose values for (x_2, x_1, x_0) that cause the observed heights to be as close as possible to the heights that the model predicts.

When you substitute the values $t = 0.25$ and $h = 11.75$ into the model you get an equation $11.75 = 0.03125x_2 - 0.25x_1 + x_0$ in the three unknown parameters. Each subsequent observation gives another such equation. The 10 observations give rise to 10 equations in all:

$$\begin{aligned}
11.75 &= -0.03125x_2 - 0.25x_1 + x_0 \\
13.00 &= -0.12500x_2 - 0.50x_1 + x_0 \\
13.50 &= -0.28125x_2 - 0.75x_1 + x_0 \\
13.25 &= -0.50000x_2 - 1.00x_1 + x_0 \\
12.50 &= -0.78125x_2 - 1.25x_1 + x_0 \\
11.50 &= -1.12500x_2 - 1.50x_1 + x_0 \\
9.50 &= -1.53125x_2 - 1.75x_1 + x_0 \\
7.00 &= -2.00000x_2 - 2.00x_1 + x_0 \\
3.75 &= -2.53125x_2 - 2.25x_1 + x_0 \\
0.00 &= -3.12500x_2 - 2.50x_1 + x_0
\end{aligned}$$

Or, to say the same thing in matrix terms,

$$\begin{bmatrix}
-0.03125 & -0.25 & 1.0 \\
-0.12500 & -0.50 & 1.0 \\
-0.28125 & -0.75 & 1.0 \\
-0.50000 & -1.00 & 1.0 \\
-0.78125 & -1.25 & 1.0 \\
-1.12500 & -1.50 & 1.0 \\
-1.53125 & -1.75 & 1.0 \\
-2.00000 & -2.00 & 1.0 \\
-2.53125 & -2.25 & 1.0 \\
-3.12500 & -2.50 & 1.0
\end{bmatrix} \times \begin{bmatrix} x_2 \\ x_1 \\ x_0 \end{bmatrix} = \begin{bmatrix}
11.75 \\ 13.00 \\ 13.50 \\ 13.25 \\ 12.50 \\ 11.50 \\ 9.50 \\ 7.00 \\ 3.75 \\ 0.00
\end{bmatrix}$$

You wouldn't expect these 10 equations in 3 variables to have an exact solution, so it makes sense to look for the values of the variables that come closest to solving the equations in the least-squares sense.

Matlab makes it easy to set up as well as solve the least squares problem. The matrix `A` of coefficients can be computed from the vector `t` of times, which has a simple expression because the times are evenly spaced. Only the vector `h` of heights needs to be typed in explicitly:

```
>> t = (0.25 : 0.25 : 2.50)';
>> h = [11.75; 13.00; 13.50; 13.25; 12.50;
        11.50;  9.50;  7.00;  3.75;  0.00];
>> A = [-0.5*t.^2, -t, ones(10,1)];
>> x = A \ h
x =
    9.7879
   -8.2402
   10.0208
```

In computing the first column of `A`, we make use of Matlab's `.^` operator to individually square the elements of vector `t`. Matlab's \ operator automatically computes a least-squares approximate solution when applied to any coefficient matrix that is not square.

Recalling that we wrote $x = (x_2, x_1, x_0)$ in the statement of the equations above, we can conclude from the Matlab solution that your window is about 10 meters high, that you threw the rock upwards with an initial velocity of about $8\frac{1}{4}$ m/sec, and that the acceleration due to gravity was about $9\frac{3}{4}$ m/sec^2. Matlab also makes it easy to compare the model's predicted heights `A*x` with the actual heights `h`, and to compute the sum of squared differences between them:

```
>> [A*x, h]
ans =
   11.7750   11.7500
   12.9174   13.0000
   13.4481   13.5000
   13.3670   13.2500
   12.6742   12.5000
   11.3697   11.5000
    9.4534    9.5000
    6.9254    7.0000
    3.7856    3.7500
    0.0341         0
>> sumsquares = sum ((A*x-h).^2)
sumsquares =
    0.0813
```

The differences `A*x-h` are often referred to as the *errors* in the model's predictions, relative to the actual observations. The acceptable amount of error varies from one application to another; in this particular example the error would probably be seen as reasonable, considering the crudeness of the measurements.

We can gain some further confidence in the model by comparing its performance to a few alternatives. If the acceleration due to gravity were not a significant effect, for instance, then we could employ the simpler model

$$h = -x_1 t + x_0.$$

With the same data, this model gives rise to 10 equations as before, but with all

the terms involving x_2 deleted. The Matlab analysis can thus also proceed as before, but with the first column of the coefficient matrix A removed:

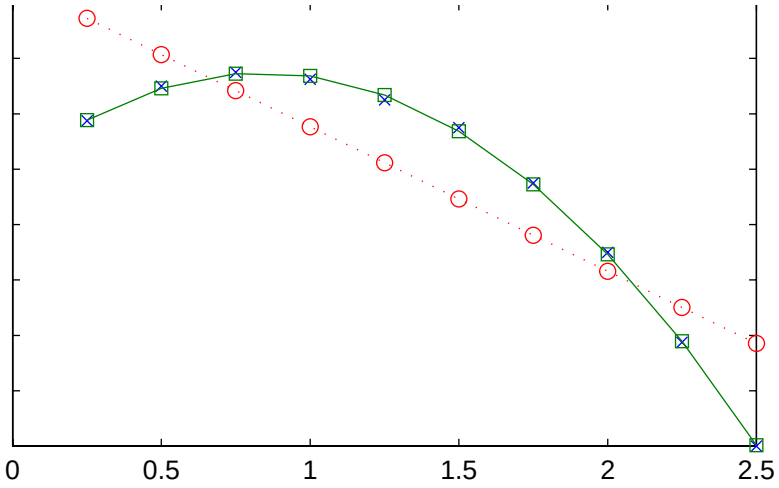
```
>> A2 =[-t, ones(10,1)];
>> x2 = A2 \ h
x2 =
    5.2182
   16.7500
>> sumsquares = sum ((A2*x2-h).^2)
sumsquares =
   49.4795
```

We see that the solution is much different here, with x_0 having changed from 10 to $16\frac{3}{4}$ and x_1 from less than -8 to more than 5. The sum of squares has ballooned, moreover, from a small fraction to almost 50! This suggests that the alternative model is greatly inferior to our original one.

Removing a variable from the model was bound, in fact, to result in some increase in the sum of squares. If it had been a small increase, however, we might have preferred the simpler model. You can see more clearly why the increase was so large by plotting the observed heights (h) against the heights predicted by the models (A * x and A2 * x2). The Matlab command

```
plot (t,h,'x', t,A3*x3,'s-', t,A2*x2,'o:')
```

does the trick, producing the following plot:



Each $\times$ marks an observed height, and each $\square$ a corresponding height predicted by our original model. The discrepancy between the two is only barely visible. Each $\bigcirc$ marks a height predicted by our second model; because it lacks the quadratic (t^2) term, its graph is just a line, which cannot possibly give a good fit to the object's curved trajectory.

Just as removing a term from the model makes it less predictive, adding a term can only improve its predictive abilities, with a corresponding reduction

in the sum of squares. Suppose, for example, that we add a cubic term to the model like this:

$$h = \tfrac{1}{3}x_3t^3 - \tfrac{1}{2}x_2t^2 - x_1t + x_0$$

Then one more column must be added to the coefficient matrix, and the least squares analysis in Matlab proceeds as follows:

```
>> A3 = [(1/3)*t.^3, -(1/2)*t.^2, -t, ones(10,1)];
>> x3 = A3 \ h
x3 =
   -0.3077
    8.9417
   -7.7523
   10.1583
>> sumsquares = sum ((A3*x3-h).^2)
sumsquares =
    0.0734
```

The coefficient of the added term, -0.3077, is quite small, while the other coefficients have values not much different from those we found for our original model. The sum of squares has indeed gone down, but only from an already small value of 0.0813 to a slightly smaller value of 0.0734. These observations suggest that the effect of the cubic term is not significant, and that the original quadratic model is more likely to be the one that truly describes the motion of objects subject to gravity.

Our analysis of this example has used "small value" and "not significant" in an informal way. These and related terms can be made precise, however, by use of ideas developed in the field of statistics. Specifically, the subject known as *statistical regression* develops precise, numerical tests to help people decide which terms of a model like ours are significant.

An example from computational linear algebra. It's no surprise that finding the inverse takes an increasing amount of time for increasingly large matrices. Matlab's standard `inv` function takes about the same amount of time, moreover, for any two matrices of the same size, regardless of what particular numbers are in them.

Thus it makes sense to investigate inversion time by applying `inv` to randomly generated matrices of various sizes. The following short Matlab M-file, `invtime.m`, computes inverses from 100×100 to 1000×1000, in steps of 100:

```
for k = 1:10
   dim(k,1) = 100 * k;
   t = cputime;
   inverse = inv(rand(dim(k,1)));
   time(k,1) = cputime - t;
end;
```

The `cputime` function measures the "computer time" used by Matlab since its

current session began. Thus the time to compute each matrix inverse is determined by taking the difference between `cputime` readings before and after the `inv` function is executed. The matrix dimensions and inversion times are stored in vectors `dim` and `time`, respectively. On one moderately fast computer, the results come out like this:

```
>> invtime
>> format short g
>> [dim; time]
ans =
       100      0.08
       200      0.57
       300      1.96
       400      5.25
       500     10.91
       600     19.06
       700      30.7
       800     47.09
       900     67.13
      1000     93.05
```

What is the pattern here? Our challenge is to find a good model for inversion time as a function of matrix dimension.

The number of elements in an $n \times n$ matrix is n^2. The number of multiplications in computing the product of such a matrix with an n-vector is also n^2, since each matrix element is involved in exactly one multiplication — try a small example if you're not convinced. The product of two $n \times n$ matrices amounts to a product of one of the matrices by each of the n columns of the other, for a total of $(n)(n^2) = n^3$ multiplications. Facts like these suggest that we should try to fit a polynomial function to the inversion-time data. That means we should consider mathematical models of the form

$$t = x_0 + x_1 n + x_2 n^2 + x_3 n^3 + x_4 n^4 + \dots ,$$

where n is the dimension and t is the time for inverting an $n \times n$ matrix. The goal of our analysis will be to choose a particular model by deciding which and how many of the polynomial terms to use.

Just as in our physics example, each data point gives one equation when substituted into a model. For 10 observations, we get 10 equations, while the number of variables equals the number of terms that we include in the model. Thus we can expect to have significantly more equations than variables. The times are only approximate, moreover; they are reported only to two decimal places, and if you run `invtime.m` more than once you'll see that they come out somewhat different each time. Thus a least-squares appraoch makes sense in this situation.

This time we'll create an M-file to automate setting up and solving least-squares problems using different combinations of polynomial terms. To do this we first observe that if we use, say, the first 3 terms, then the equations have

the form

$$\begin{bmatrix} n_1^0 & n_1^1 & n_1^2 \\ n_2^0 & n_2^1 & n_2^2 \\ \vdots & \vdots & \vdots \\ n_{10}^0 & n_{10}^1 & n_{10}^2 \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_{10} \end{bmatrix}$$

where $n_1, n_2, \ldots, n_{10}$ are the matrix dimensions and $t_1, t_2, \ldots, t_{10}$ are the corresponding observed times. (Every n_j^0 equals 1, of course, and every n_j^1 is the same as n_j.) Translating to the terminology of the Matlab vectors that were produced by `invtime.m`, we want to consider equations of the form `A*x=time`, where the matrix `A` has a column `dim.^k` for each term `k` included in the model. The least-squares solution is given, as you have seen, by `x=A\time`. Our M-file automates this computation by taking as its arguments `dim`, `time`, and a row vector of nonnegative terms to be included in the model:

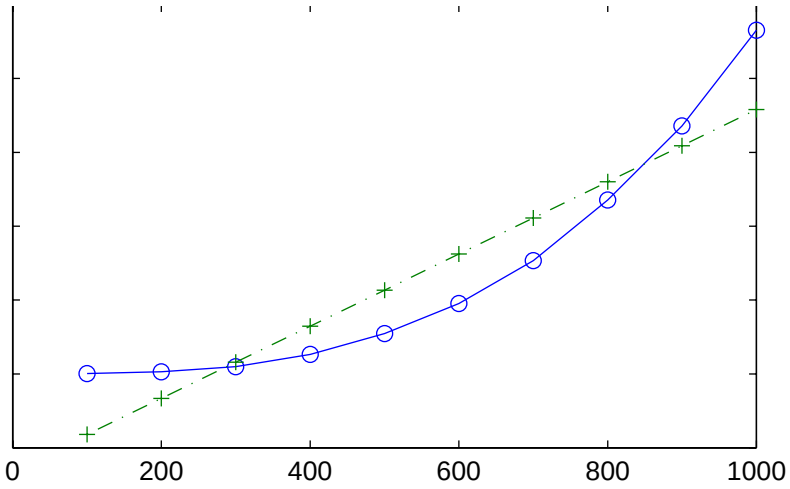
```
function polyfit (dim,time,terms)
   A = [];
   for k = terms
      A = [A dim.^k];
   end
   x = A \ time
   err = time - A * x;
   sumofsquares = sum (err.^2)
   plot (dim,time,'o-', dim,A*x,'+:')
```

The best-fit solution (in column vector `x`) and the corresponding sum of squared errors (in scalar `sumofsquares`) are displayed. Then the observed times (`time`) are plotted against the predicted times (`A*x`).

We start by fitting the data to a linear function, using only the first two terms of the polynomial model:

```
>> polyfit(dim, time, [0 1])
x =
       -26.18
     0.097745
sumofsquares =
         1350
```

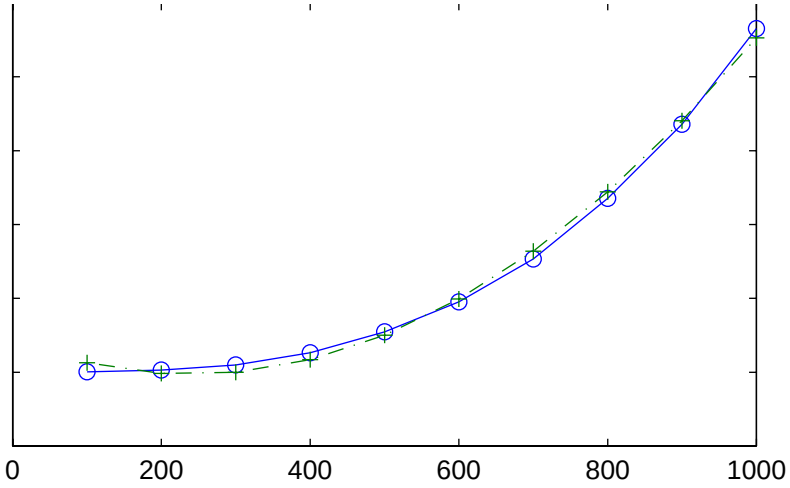
The sum of squares seems rather high relative to the times we are trying to model, and the plot confirms that this is a poor model:



We get a much better fit to a quadratic function, using the first three terms:

```
>> polyfit (dim, time, [0 1 2])
x =
       8.6033
    -0.076171
   0.00015811
sumofsquares =
       30.118
```

The sum of squares is more than 10 times smaller, and the plot shows the times predicted by the model coming much closer to the times observed:



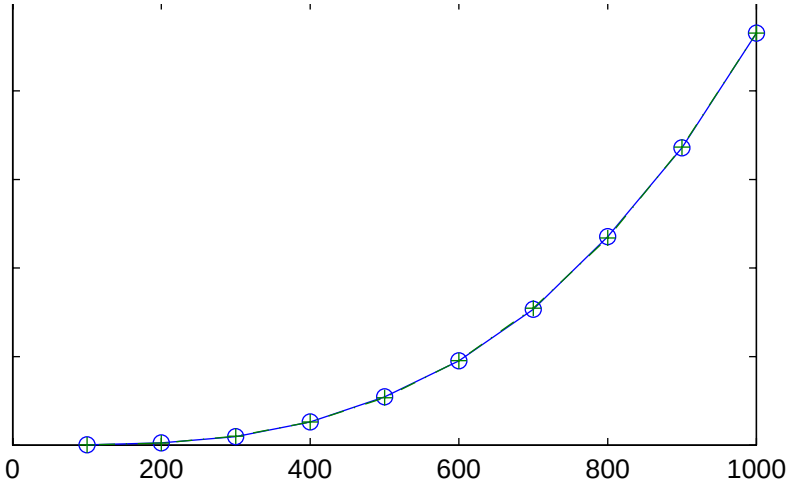
Even here, though, the plot does not look quite right. We would expect inverting a matrix of a certain size to take less time than inverting any larger matrix. As the dimension decreases toward zero, however, the plot shows the predicted

time falling but then starting to rise again. This is a strong sign that our model is still not fully capturing the relationship between inversion time and matrix dimension.

We thus continue the analysis, next fitting the cubic function given by the first four polynomial terms:

```
>> polyfit (dim, time, [0 1 2 3])
x =
         0.16
   -0.0012834
  -4.2657e-06
   9.8407e-08
sumofsquares =
      0.20591
```

The sum of squares is again more than 10 times smaller. What's more, the fit is now so close that the plots of the observed and predicted values are barely distinguishable:



Thus it would seem reasonable to conclude that inversion time is a cubic function of matrix dimension. Further evidence is provided by the results of fitting a quartic polynomial function:

```
>> polyfit (dim, time, [0 1 2 3 4])
x =
         0.18
   -0.0015398
  -3.3275e-06
   9.7125e-08
   5.8275e-13
sumofsquares =
      0.20586
```

The sum of squares is essentially the same as in the cubic case, strongly suggesting that there is nothing to be gained from adding a higher-order term.

We thus arrive at the following model for inversion time as a function of matrix dimension:

$$t = 0.16 - 0.0013n - 0.0000043n^2 + 0.000000098n^3.$$

(Since this is only an estimate based on approximate timings, we have rounded the Matlab output to two significant digits.) Looking at this formula, you might well wonder why the last two terms were needed, when their coefficients have such small magnitudes. For values of n in the range of 100 to 1000, however, the magnitudes of n^2 and n^3 are rather large, and hence make significant contributions to the formula even when multiplied by their small coefficients. You can observe that this is the case, by adding

```
termvals = A * diag(x)
```

to `polyfit.m`. The matrix `diag(x)` is square with `x` along its diagonal and zeros elsewhere:

```
 0.16            0            0            0
    0   -0.0012834            0            0
    0            0  -4.2657e-06            0
    0            0            0   9.8407e-08
```

Thus `A * diag(x)` is a matrix the same size as `A`, but with each column multiplied by its corresponding term in `x`:

```
termvals =
      0.16     -0.12834    -0.042657     0.098407
      0.16     -0.25667     -0.17063      0.78726
      0.16     -0.38501     -0.38392        2.657
      0.16     -0.51335     -0.68252       6.2981
      0.16     -0.64169      -1.0664       12.301
      0.16     -0.77002      -1.5357       21.256
      0.16     -0.89836      -2.0902       33.754
      0.16      -1.0267      -2.7301       50.384
      0.16       -1.155      -3.4552       71.739
      0.16      -1.2834      -4.2657       98.407
```

This shows that the third-order term actually makes the largest contribution to the predicted time, and that in fact it would make more sense to consider dropping the lower-order terms. As an example, here are the results from a model that uses only the n^2 and n^3 terms:

```
>> polyfit(dim, time, [2 3])
x =
  -6.8577e-06
   9.9902e-08
sumofsquares =
       0.2137
```

The sum of squared errors increases very little. Thus the model

$$t = -0.0000069n^2 + 0.00000010n^3.$$

(again rounded to 2 digits) would normally be preferred, for its simplicity, to the one previously given.

If you replace `cputime` with `flops` in the two places that it appears in `invtime.m`, you'll get estimates of the number of *floating-point operations* such as addition, multiplication, and division that were performed in computing each matrix inverse. You can then do a similar analysis to model operations as a polynomial function of matrix dimension. You should find that the number of operations needed is very nearly $2n^3$; together with the model for time above, this suggests that the computer used for these tests performed about 20 million operations per second.