Lecture 04.15 Regression

Suppose we have a sample with two measurands: (1) the force F through a spring and (2) its displacement X (not from equilibrium). We would like to determine an analytic function that relates the variables, perhaps for prediction of the force given another displacement.

There is some variation in the measurement. Let's say the following is the sample.

```
X_a = 1e-3*[10,21,30,41,49,50,61,71,80,92,100]'; % m
F_a = [50.1,50.4,53.2,55.9,57.2,59.9,61.0,63.9,67.0,67.9,70.3]'; % N
```

Let's take a look at the data. The result is Figure 04.23.

```
h = figure;
p = plot(X_a*1e3,F_a,'.b','MarkerSize',15);
xlabel('$X$ (mm)','interpreter','latex')
ylabel('$F$ (N)','interpreter','latex')
xlim([0,max(X_a*1e3)])
grid on
hgsave(h,'figures/temp');
```

How might we find an analytic function that agrees with the data? Broadly, our strategy will be to assume a general form of a function and use the data to set the parameters in the function such that the difference between the data and the function is minimal.

Let y be the analytic function that we would like to fit to the data. Let y_i denote the value of $y(x_i)$, where x_i is the ith value of the random variable X from the sample. Then we want to minimize the differences between the force measurements F_i and y_i.

From calculus, recall that we can minimize a function by differentiating it and solving for the zero-crossings (which correspond to local maxima or minima).

First, we need such a function to minimize. Perhaps the simplest, effective function D is constructed by squaring and summing the differences we want to minimize, for sample size N:

(recall that $y_i = y(x_i)$, which makes D a function of x).

Now the form of y must be chosen. We consider only mth-order polynomial functions y, but others can be used in a similar manner:

$$y(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m. \tag{04.24}$$

If we treat D as a function of the polynomial coefficients a_j, i.e.

$$D(a_0, a_1, \cdots, a_m), \tag{04.25}$$

and minimize D for each value of x_i, we must take the partial derivatives of D with respect to each a_j and set each equal to zero:

This gives us N equations and $m+1$ unknowns a_j. Writing the system in matrix form,

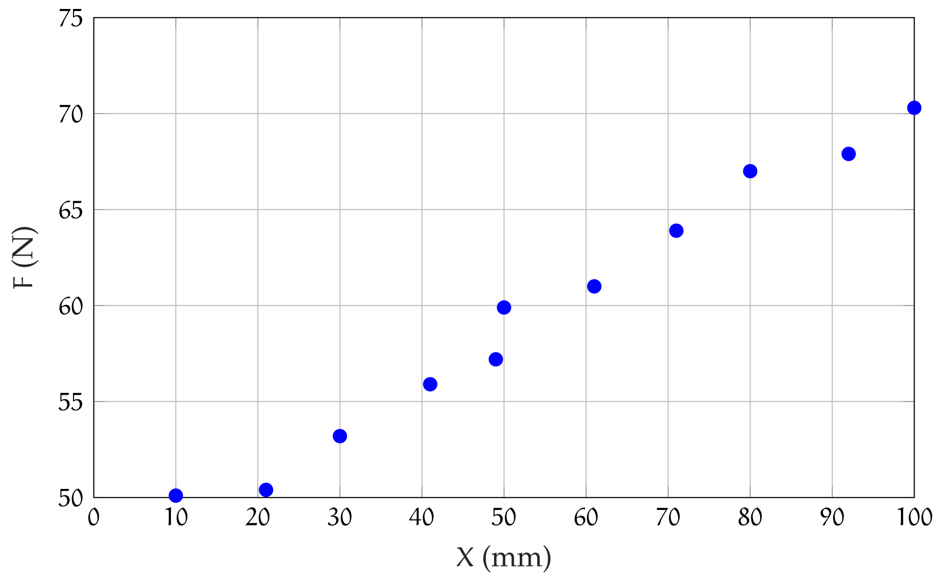

Figure 04.23: force-displacement data.

$$\underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^m \\ 1 & x_2 & x_2^2 & \cdots & x_2^m \\ 1 & x_3 & x_3^2 & \cdots & x_3^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^m \end{bmatrix}}_{A_{N\times(m+1)}} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}}_{a_{(m+1)\times 1}} = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}}_{b_{(m+1)\times 1}} . \tag{04.26}$$

Typically $N > m$ and this is an *overdetermined system*. Therefore, we usually can't solve by taking A^{-1} because A doesn't have an inverse!

Instead, we either find the *Moore-Penrose pseudo-inverse* $A^\dagger$ and have $\mathbf{a} = A^\dagger \mathbf{b}$ as the solution, which is *inefficient*—or we can approximate $\mathbf{b}$ with an algorithm such as that used by *Matlab*'s \ operator. In the latter case, `a_a = A\b_a`.

Let's use *Matlab*'s \ operator to find a good fit for the sample. Now, there's the sometimes-difficult question "what order should we fit?" Let's try out several and see what the squared-differences function D gives.

```
N = length(X_a); % sample size
m_a = 2:N; % all the order up to N

A = NaN*ones(length(m_a),max(m_a),N);
for k = 1:length(m_a) % each order
    for j = 1:N % each measurement
        for i = 1:( m_a(k) + 1 ) % each coef
            A(k,j,i) = X_a(j)^(i-1);
        end
    end
end
disp(squeeze(A(2,:,1:5)))
```

```
    1.0000    0.0100    0.0001    0.0000       NaN
    1.0000    0.0210    0.0004    0.0000       NaN
    1.0000    0.0300    0.0009    0.0000       NaN
    1.0000    0.0410    0.0017    0.0001       NaN
    1.0000    0.0490    0.0024    0.0001       NaN
    1.0000    0.0500    0.0025    0.0001       NaN
    1.0000    0.0610    0.0037    0.0002       NaN
    1.0000    0.0710    0.0050    0.0004       NaN
    1.0000    0.0800    0.0064    0.0005       NaN
    1.0000    0.0920    0.0085    0.0008       NaN
    1.0000    0.1000    0.0100    0.0010       NaN
```

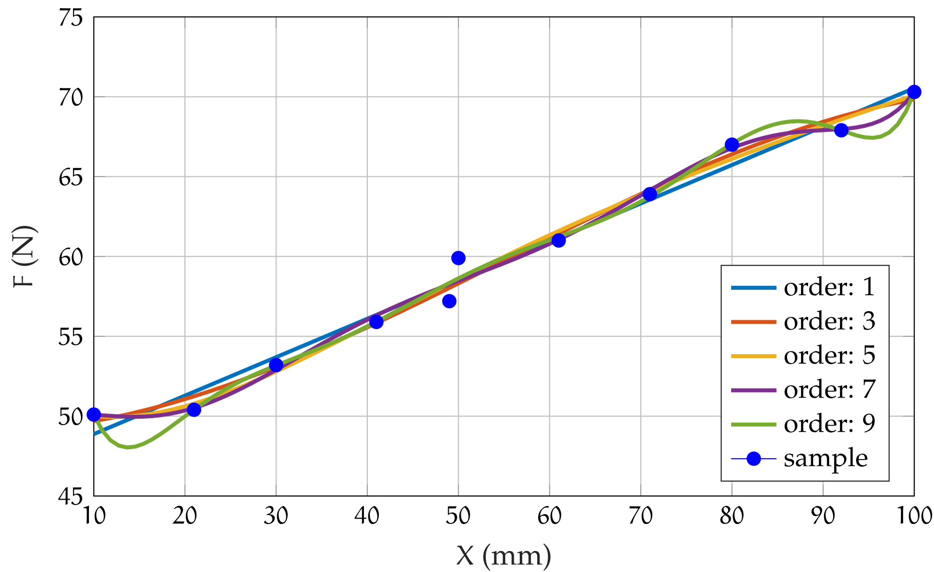

Figure 04.24: force-displacement data with curve fits.

We've printed the first five columns of the third-order matrix, which only has four columns, so `NaN`s fill in the rest.

Now we can use the \ operator to solve for the coefficients.

```
a = NaN*ones(length(m_a),max(m_a));

warning('off','all')
for i = 1:length(m_a)
    A_now = squeeze(A(i,:,1:m_a(i)));
    a(i,1:m_a(i)) = (A_now(:,1:m_a(i))\F_a)';
end
warning('on','all')
```

```
n_plot = 100;
x_plot = linspace(min(X_a),max(X_a),n_plot);
y = NaN*ones(n_plot,length(m_a)); % preallocate
for i = 1:length(m_a)
    y(:,i) = polyval(fliplr(a(i,1:m_a(i))),x_plot);
end
```