B1 Numerical Algorithms

4 Lectures, MT 2023

Lecture two – Approximate representations of data.

Wes Armour

24th October 2023

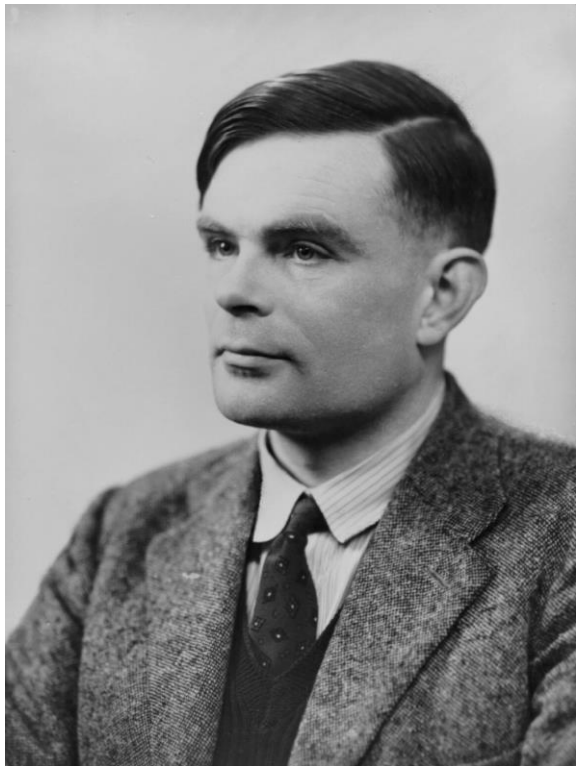
Errors/Typos/Questions to: wes.armour@eng.ox.ac.uk

Learning outcomes

- Function fitting and approximation
  - Linear regression,
  - polynomial regression,
  - condition number,
  - differentiation and noise.

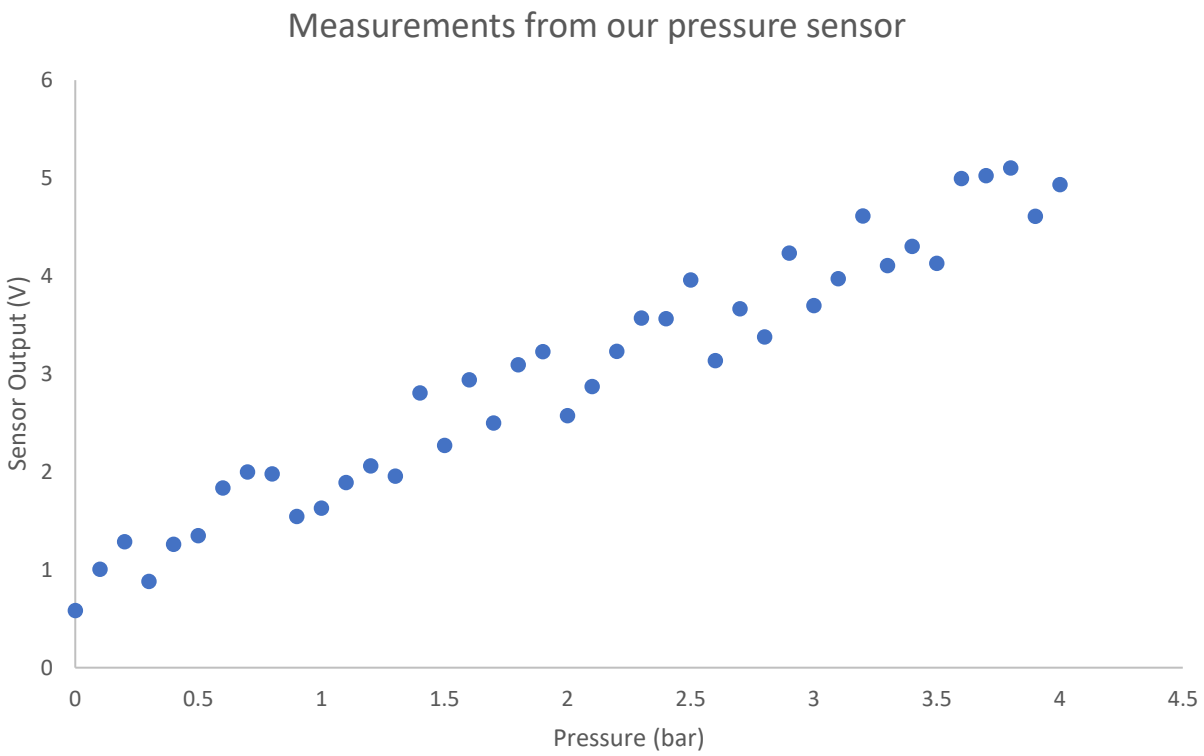
Adrien-Marie Legendre
1752 - 1833

Lewis Fry Richardson
1881 - 1953



Alan Turing
1912 - 1954

An overview of Curve Fitting



Let's return to our barometric pressure sensor (BMP180). We have some real-world data that we have collected from it.

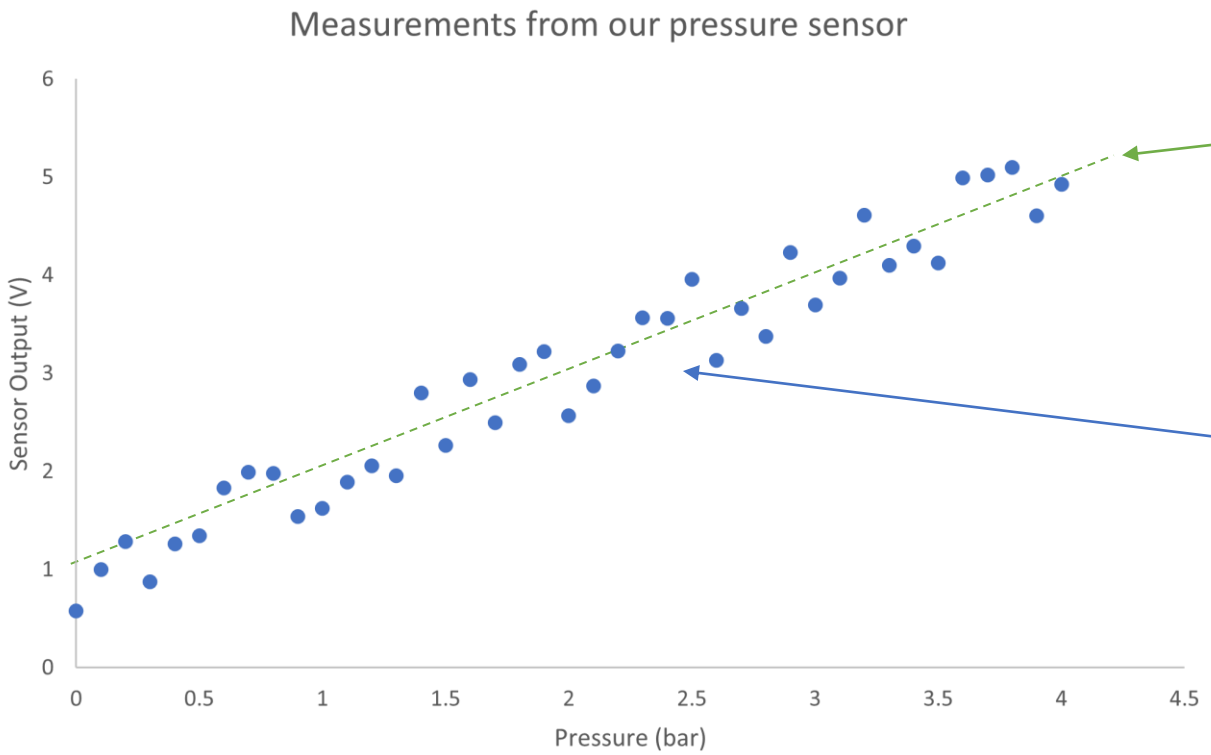
We can see some scatter in the measurement.

Do we believe that the scatter we see is real? Or is it noise, error?

Does the sensor have an underlying characteristic functional form?

In other words is there a curve (more properly a *transfer function*) that represents the sensor output?

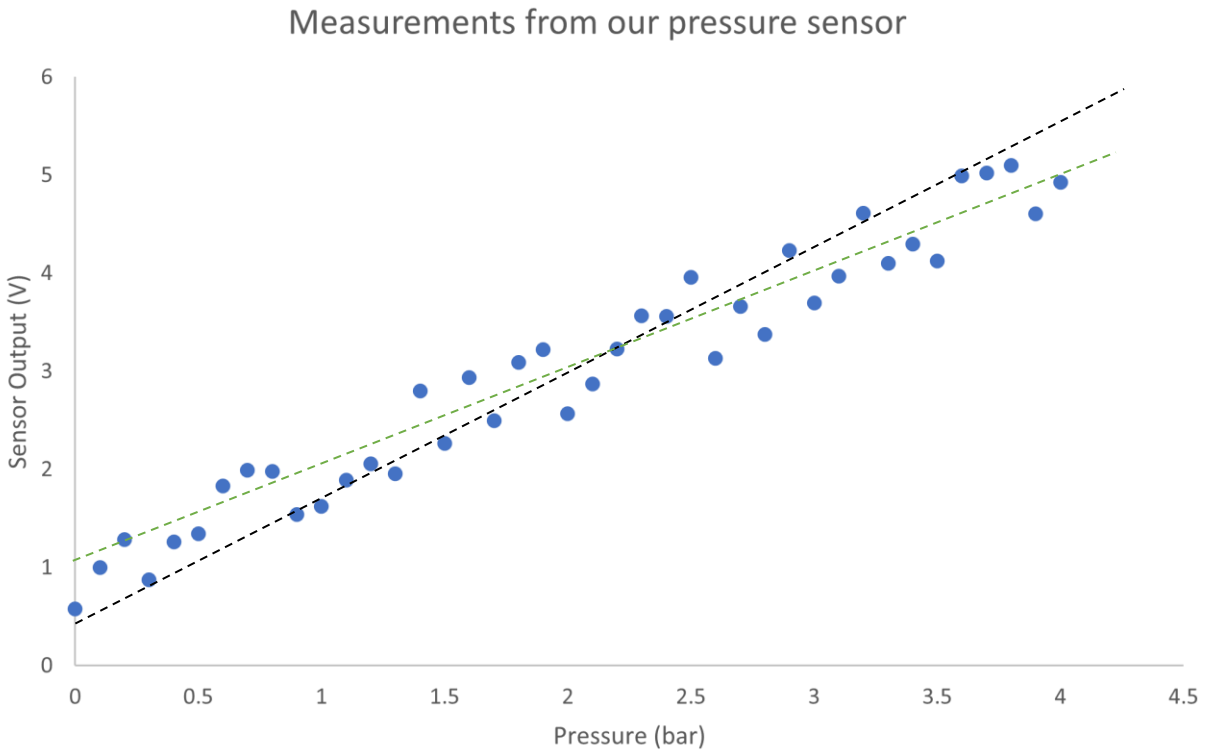
An overview of Curve Fitting



Ideal world – how the sensor should behave

Real world data

An overview of Curve Fitting



We can see that the data approximately sits along the same line or curve.

Given this:

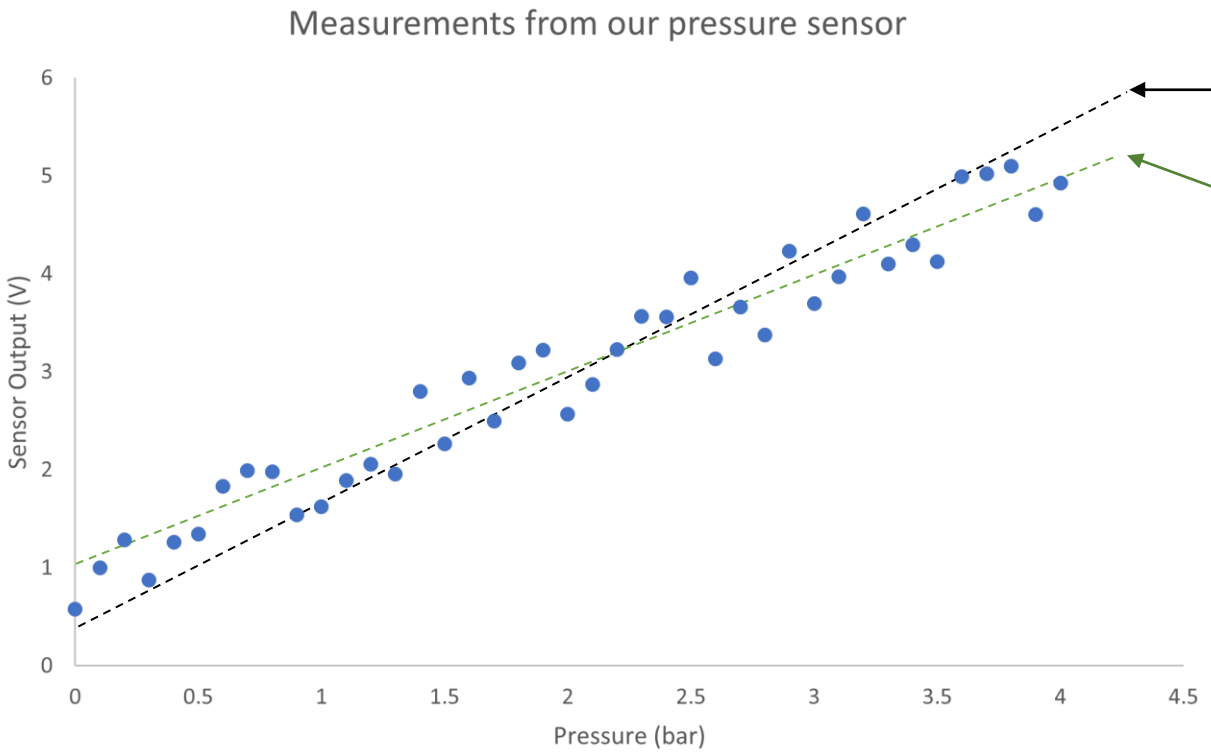
How do we find the line or curve that best represents the data?

$$y(x) = mx + c$$
$$f(x) = Ax + B$$
$$f(x) = \alpha_1 x + \alpha_0$$

But which gradient (m) and which constant (c) do we pick?

Recall the gradient $m=A=\alpha_1=dy/dx$
and the intercept $c=B=\alpha_0=y(0)$

An overview of Curve Fitting



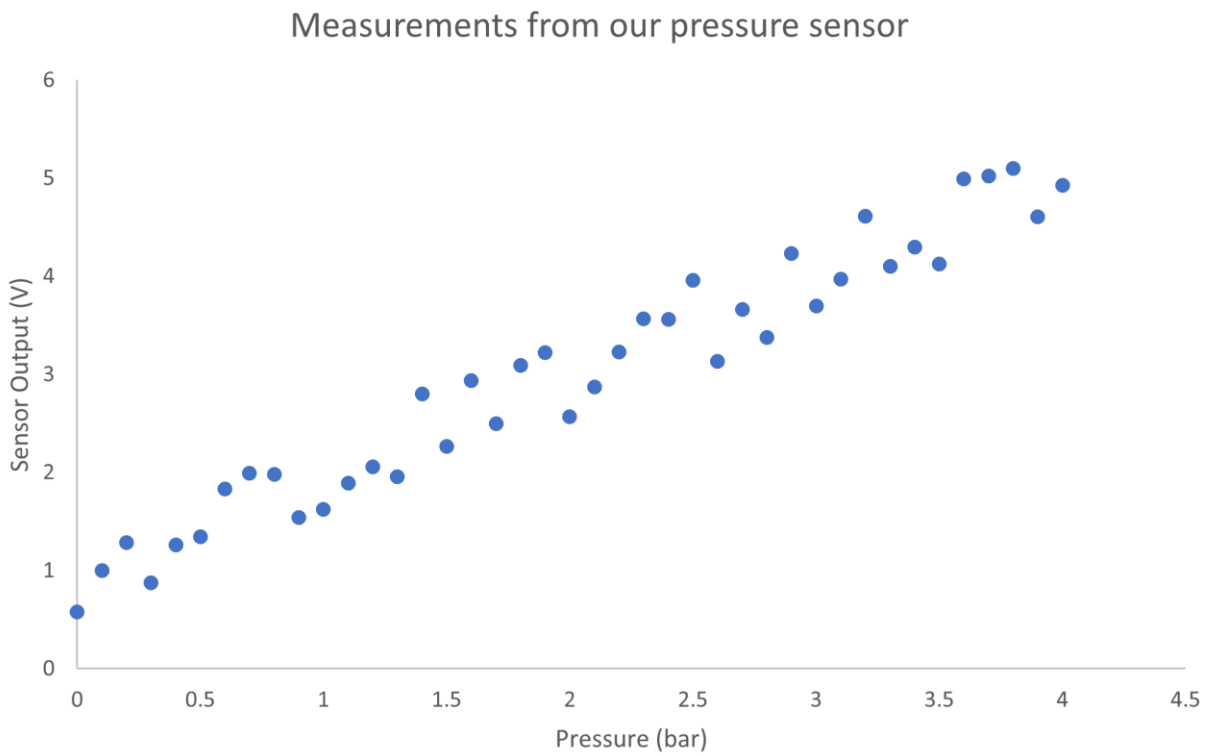
$f(x) = \beta_1 x + \beta_0$

$f(x) = \alpha_1 x + \alpha_0$

Something in between?

$f(x) = ? \, x + ?$

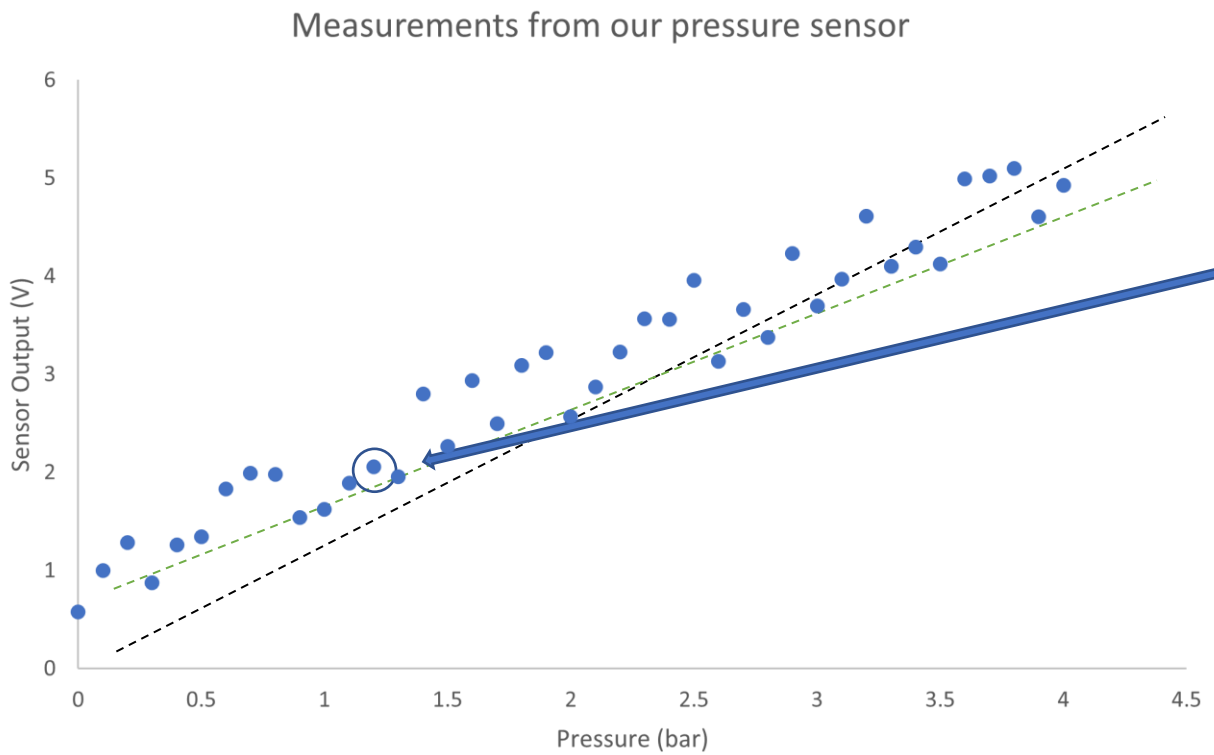
An overview of Curve Fitting



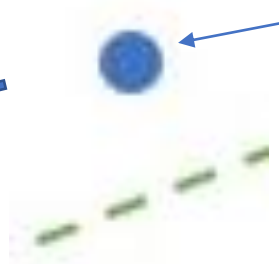
To answer this question, lets start with what we have – a collection of data points, here we have pressure (x) and voltage (y). Our dataset looks like:

$$(x_0, y_0)$$
$$(x_1, y_1)$$
$$(x_2, y_2)$$
$$(x_3, y_3)$$
$$(x_4, y_4)$$
$$\vdots$$
$$(x_N, y_N)$$

An overview of Curve Fitting



Lets consider a single point in our data set, point (x_i, y_i).



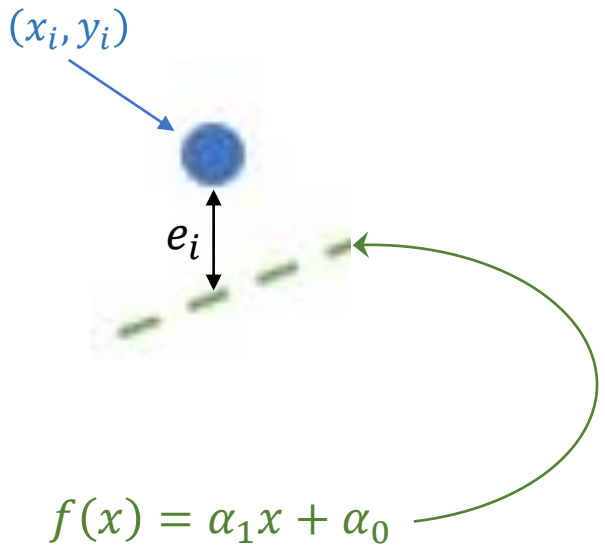
And our function that describes our data:

$$f(x) = \alpha_1 x + \alpha_0$$

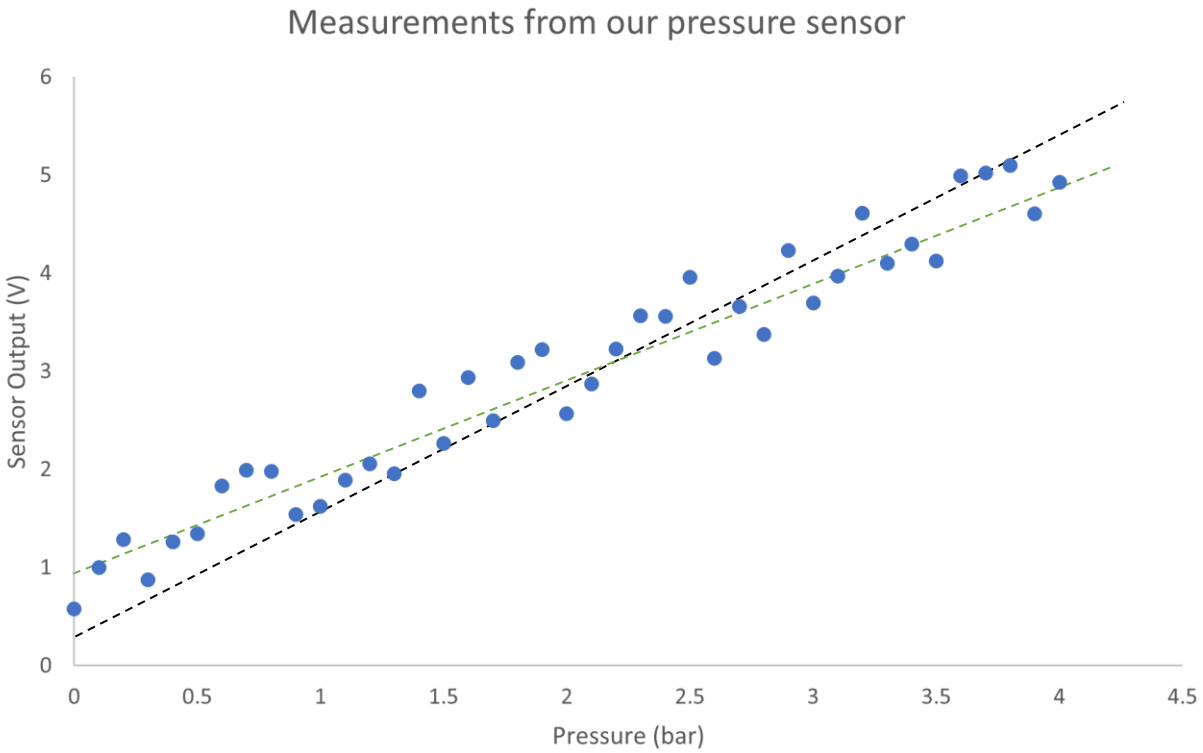
They don't match, in fact our line goes *near* our point (y_i), but has some error (e_i):

$$f(x_i) = y_i + e_i$$

An overview of Curve Fitting



An overview of Curve Fitting



So our goal is to minimise the errors (e_0 … e_N) for the whole dataset, **for all N data points** (let's call these errors E).

This gives us the ***curve of best fit***.

And we reduce our N dimensional data set to a two parameter space (α_0, α_1)

– this is dimensionality reduction.

An overview of Curve Fitting

Choice of Error Metric

We need to define a suitable error vector. We could choose to minimise the

1. Maximum error

   $$\max_i |\mathbf{e}_i|$$

   This is sometimes referred to as the **minimax** problem but is difficult to solve. Further, it places too much weight on the importance of a small amount of data that is poor (the outliers).

2. Absolute deviation error

   $$\sum_i |\mathbf{e}_i|$$

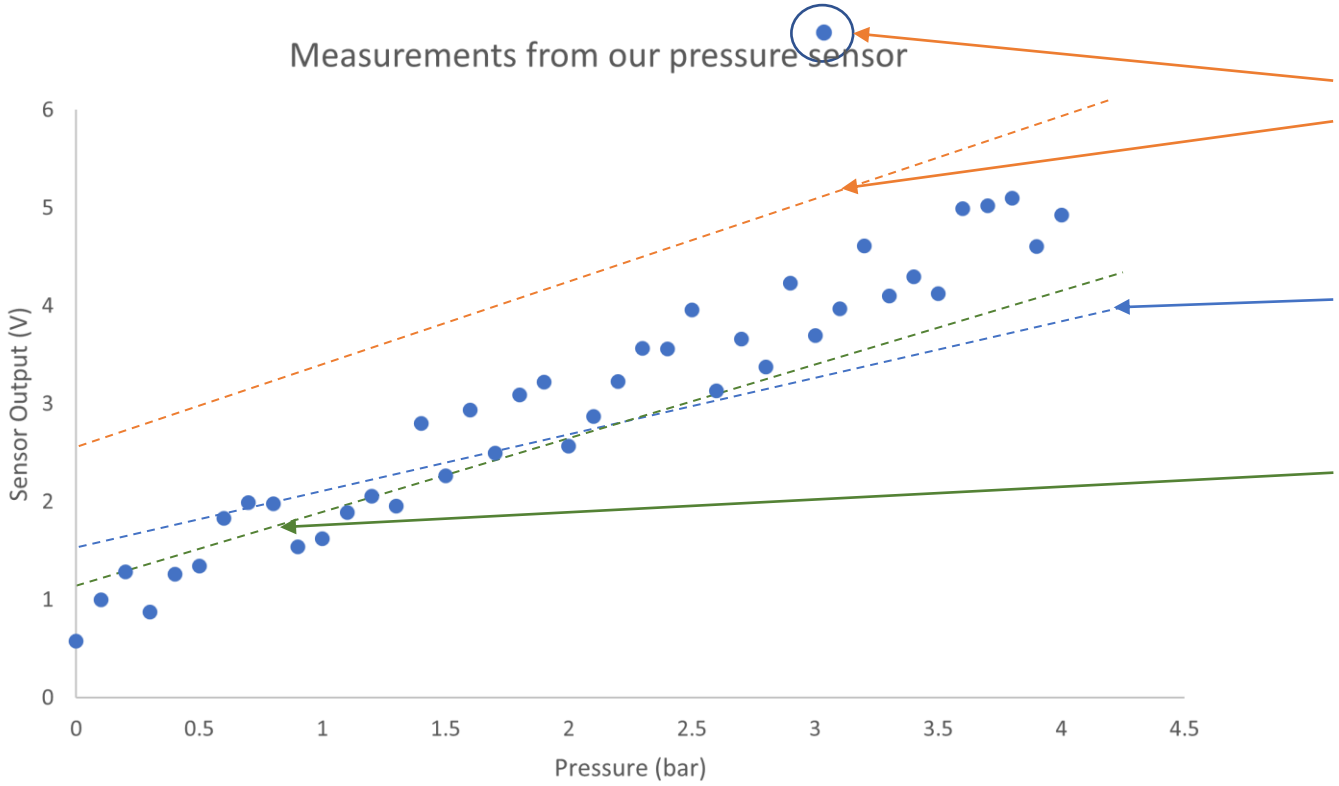
   This is again not an easy functional to minimise.This approach averages the errors but does not give much weighting to points that are considerably out of line with the rest..

3. Least squares error

   $$\sum_i (\mathbf{e}_i)^2$$

   This is the easiest error measure to use. It is a compromise of the above two; it gives some weighting to points that are out-of-line with the rest, (better than (2) but is not so biased as (1).) This is the metric used in the method described next.

An overview of Curve Fitting

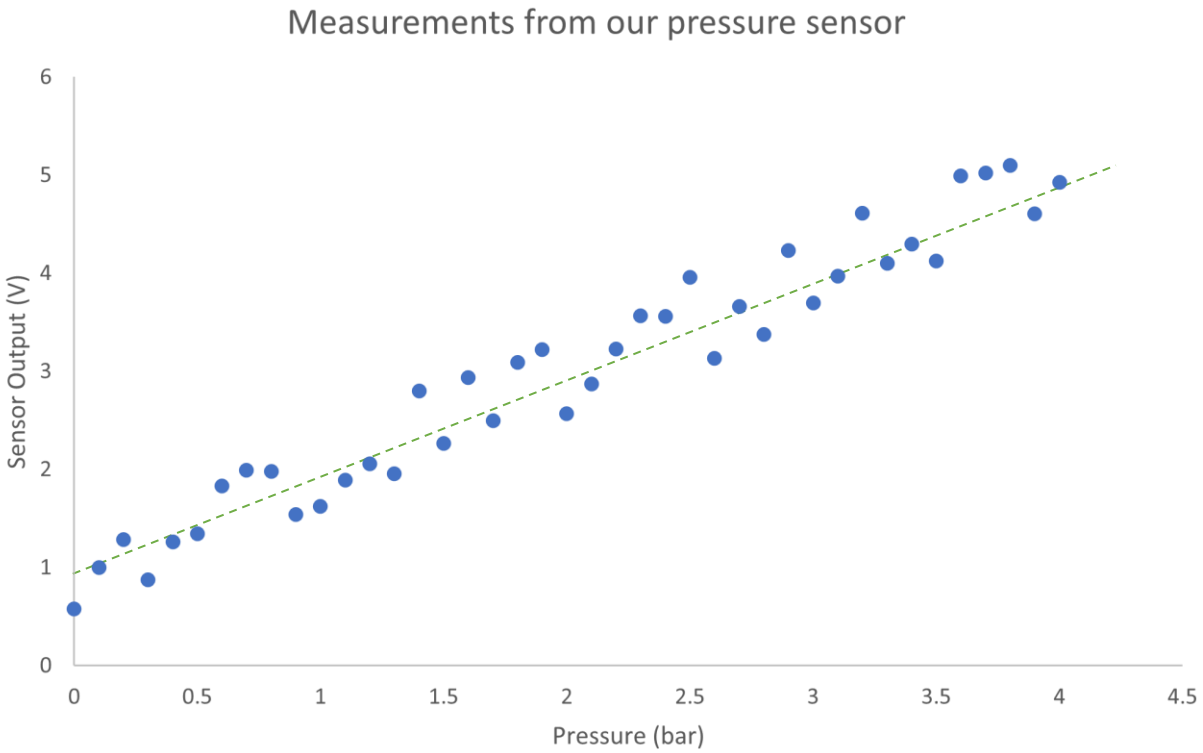


The Maximum error tries to minimise the maximum error, its very sensitive to ***outliers***

The absolute deviation error is better and is insensitive to outliers, but harder to minimise.

Finally we have the least square error. This is a reasonable compromise between the two above cases.

An overview of Curve Fitting



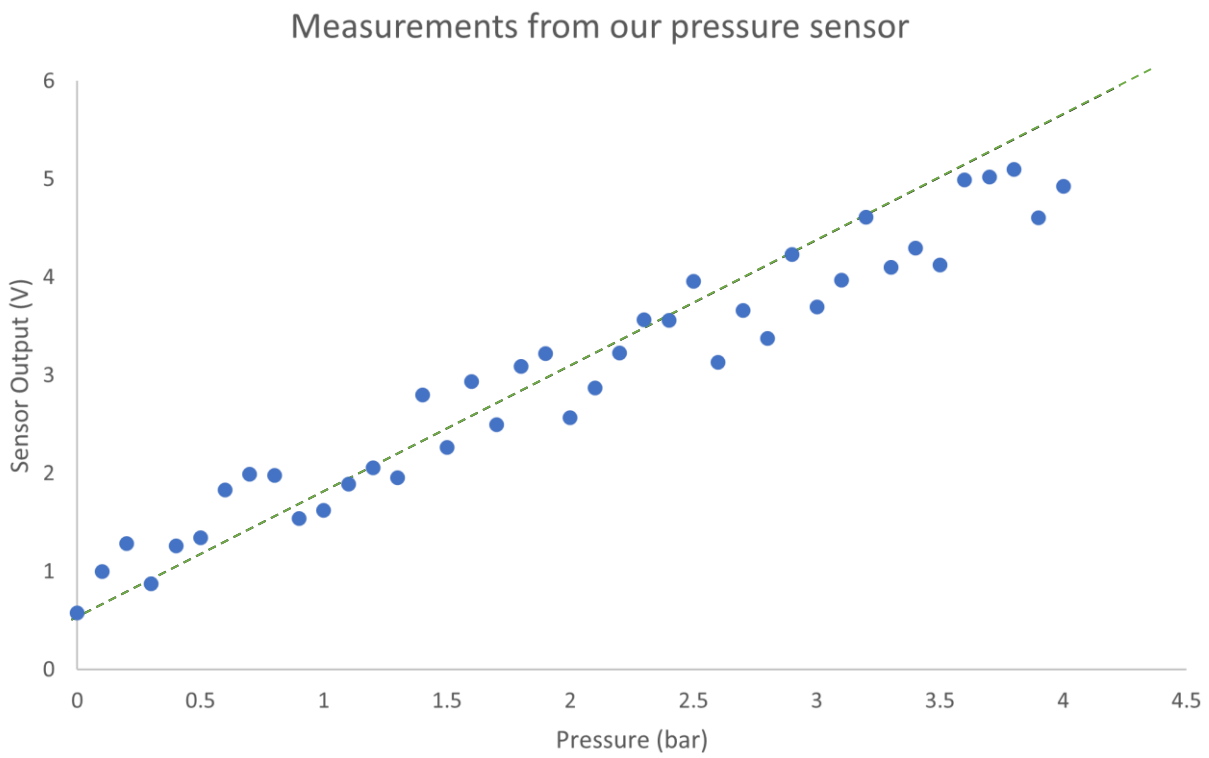
Lets concentrate on the Root-Mean-Square (RMS) error, also know as the L_2 error or the **Least squares error.**

The least squares error is the average of the individual squared errors. This is a lot like Pythagorean theorem

$$E^2 = ||e||_2 = \sqrt{\frac{1}{N}\sum_{i=0}^{N}(f(x_i) - y_i)^2}$$

$$= \sqrt{\frac{1}{N}\sum_{i=0}^{N}(e_i)^2}$$

An overview of Curve Fitting

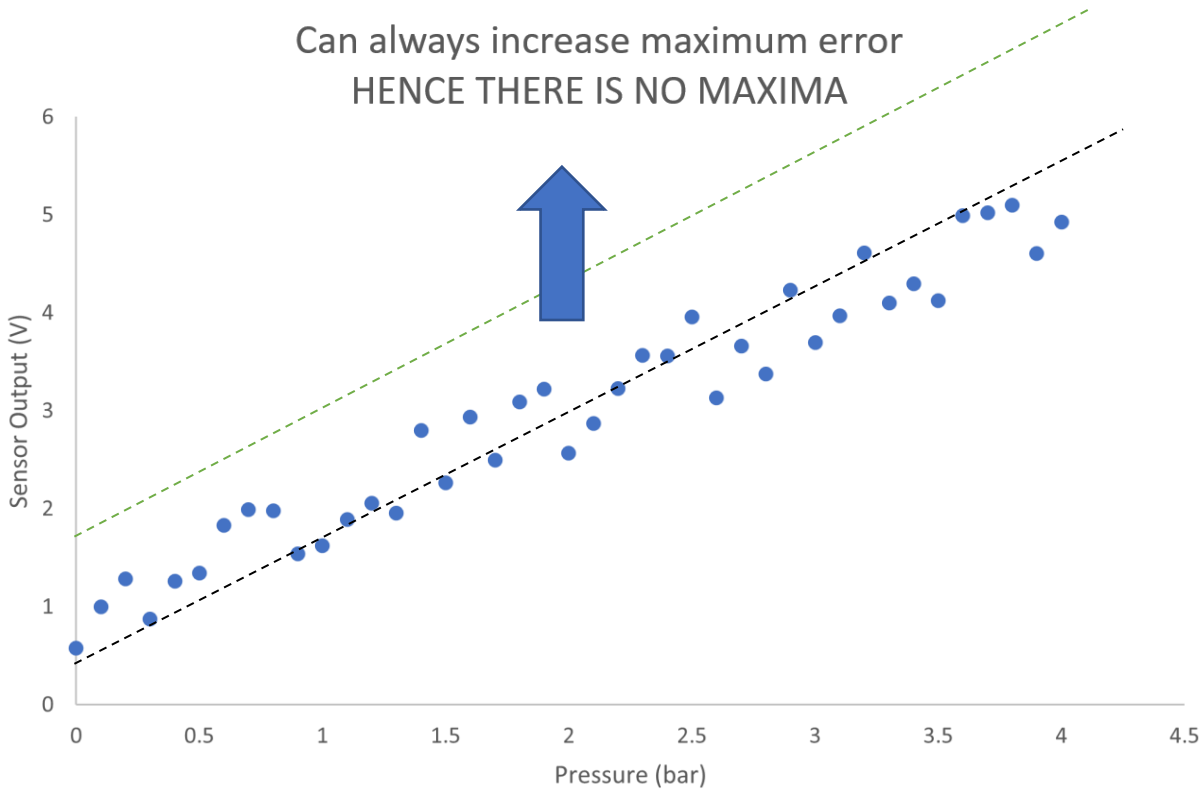


To minimise this error all we need to do is find the minimum value of the sum of squares:

$$E^2 = \frac{1}{N}\sum_{i=0}^{N}(f(x_i) - y_i)^2$$
$$= \frac{1}{N}\sum_{i=0}^{N}(\alpha_1 x_i + \alpha_0 - y_i)^2$$

To do this we turn to basic calculus. The only things we can vary in the above equation (to influence the value of ϵ) are $\boldsymbol{\alpha_1, \alpha_0}$

An overview of Curve Fitting



To minimise this error all we need to do is find the minimum value of the sum of squares:

$$E^2 = \frac{1}{N}\sum_{i=0}^{N}(f(x_i) - y_i)^2$$

$$= \frac{1}{N}\sum_{i=0}^{N}(\alpha_1 x_i + \alpha_0 - y_i)^2$$

To do this we turn to basic calculus. The only things we can vary in the above equation (to influence the value of ϵ) are $\boldsymbol{\alpha_1, \alpha_0}$

An overview of Curve Fitting

We must minimise E^2 (let's call this ϵ) with respect to $\boldsymbol{\alpha_1}, \boldsymbol{\alpha_0}$ so we need to solve:

$$\frac{\partial \epsilon}{\partial \alpha_0} = 0, \ \frac{\partial \epsilon}{\partial \alpha_1} = 0$$

Applying the chain rule…

$$\frac{\partial \epsilon}{\partial \alpha_1} = \frac{\partial \epsilon}{\partial e_i}\frac{\partial e_i}{\partial \alpha_1} = \frac{2}{N}\sum_{i=0}^{N}(\alpha_1 x_i + \alpha_0 - y_i)x_i = 0$$

$$\frac{\partial \epsilon}{\partial \alpha_0} = \frac{\partial \epsilon}{\partial e_i}\frac{\partial e_i}{\partial \alpha_0} = \frac{2}{N}\sum_{i=0}^{N}(\alpha_1 x_i + \alpha_0 - y_i) = 0$$

Rearranging gives:

$$\sum_{i=0}^{N}\left(\alpha_1 x_i^2 + \alpha_0 x_i - y_i x_i\right) = 0$$

$$\sum_{i=0}^{N}(\alpha_1 x_i + \alpha_0 - y_i) = 0$$

We have a set of simultaneous equations to solve.

Because $E^2 = \epsilon = \frac{1}{N}\sum_i (f(x_i) - y_i)$

$\equiv \frac{1}{N}\sum_i (e_i)^2$

$\therefore \frac{\partial \epsilon}{\partial e_i} = \frac{2}{N}\sum_i e_i$ & $\frac{\partial e_i}{\partial \alpha_0} = 1$ & $\frac{\partial e_i}{\partial \alpha_1} = x_i$

An overview of Curve Fitting

Writing in matrix form:

$$\begin{pmatrix} \sum_{i=0}^{N}(x_i^2) & \sum_{i=0}^{N}(x_i) \\ \sum_{i=0}^{N}(x_i) & N \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_0 \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{N}(y_i x_i) \\ \sum_{i=0}^{N}(y_i) \end{pmatrix}$$

Here we see that the solution to a **linear least squares fit** is the simple matrix inverse of a 2x2 matrix multiplied by a column vector.

Or:

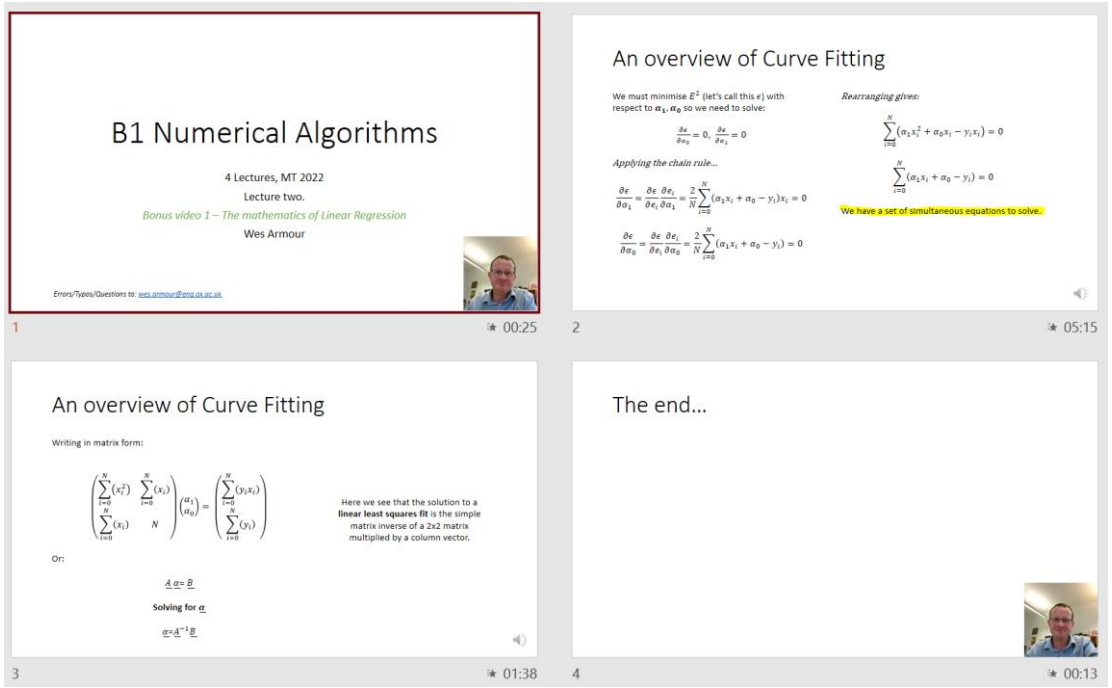
$$\underline{A}\,\underline{\alpha} = \underline{B}$$

Solving for $\underline{\alpha}$

$$\underline{\alpha} = \underline{A}^{-1}\underline{B}$$

The mathematics of Linear Regression

Panopto bonus video

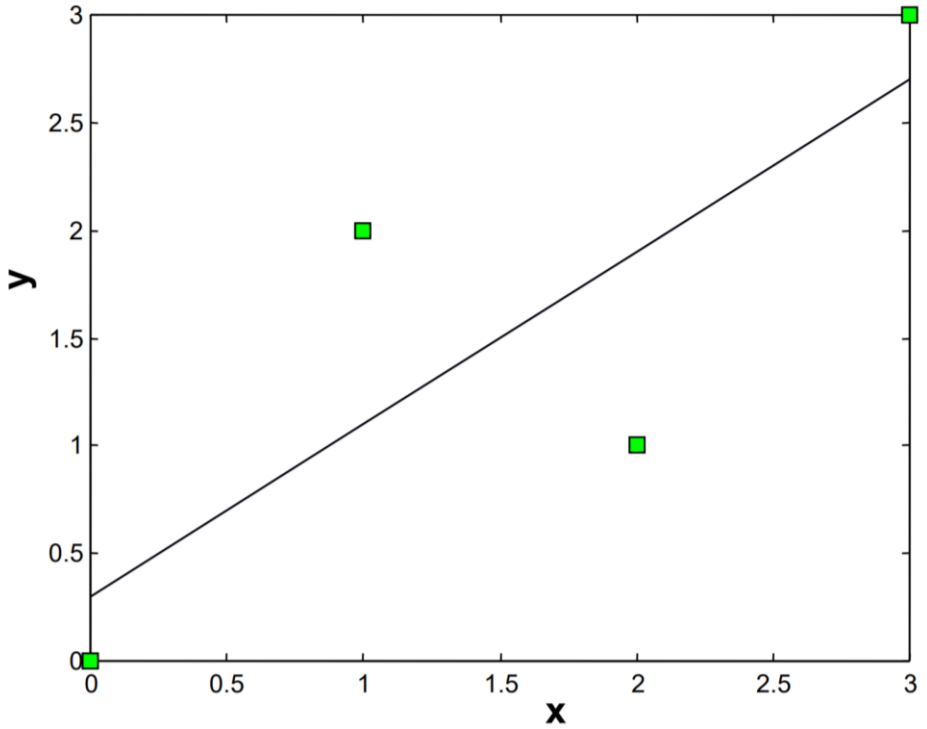


Example of linear regression

Example of linear regression ($M = 1$)

Fit a linear function to the data (0, 0), (1, 2), (2,1), (3,3) ($N = 4$).

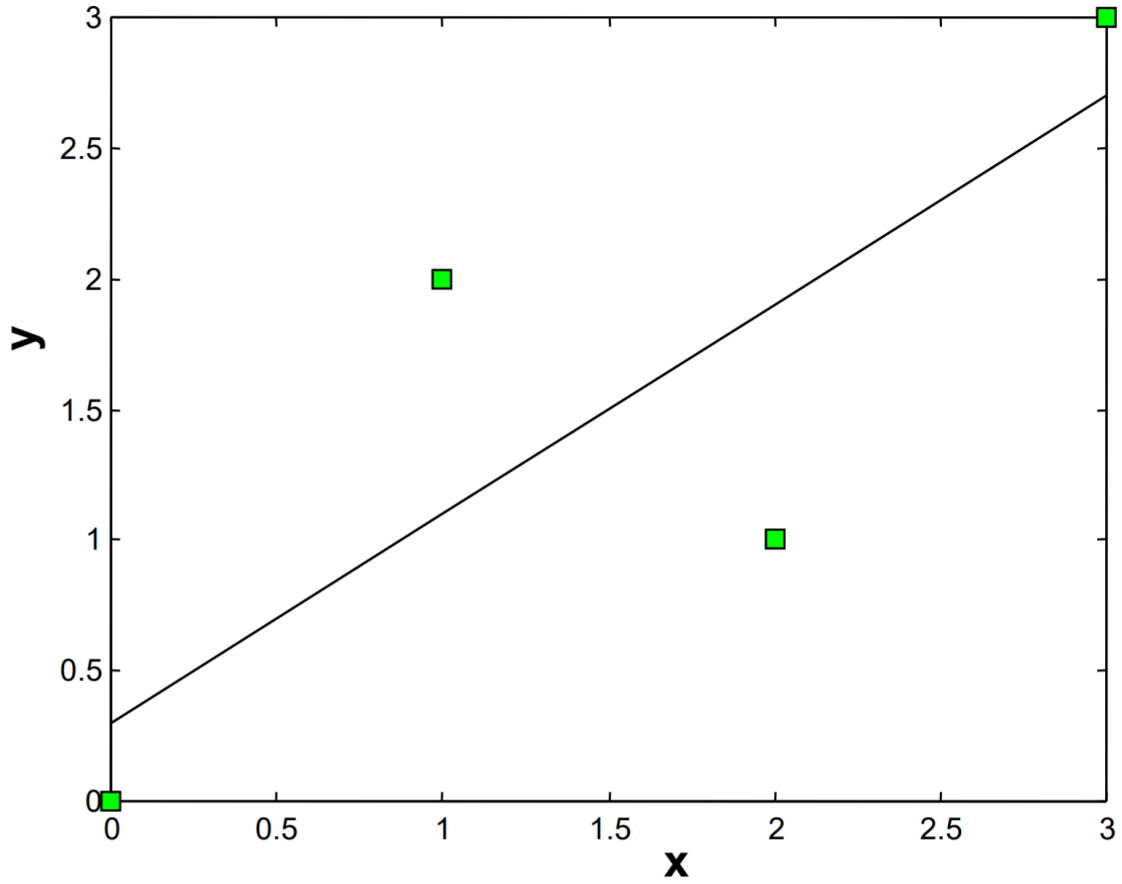
```
close all; clear all; clc;

data = [ 0,0;1,2;2,1;3,3]

sum_x_squared = sum(data(:,1).*data(:,1))

sum_x = sum(data(:,1))

two_by_two = [sum_x_squared, sum_x; sum_x, 4]

sum_xy = sum(data(:,1).*data(:,2))

sum_y = sum(data(:,2))

xy_vector = [sum_xy, sum_y]

xy_vector / two_by_two
```



The fitted line is then $y = 0.3 + 0.8x$

Example of linear regression

The fitted line looks reasonable – or does it?
In practice, check with cross-validation data



Assumptions when fitting data

A further warning…

In the above line fitting, The unstated assumption is that:

x - values are known exactly,

y - values are in error.

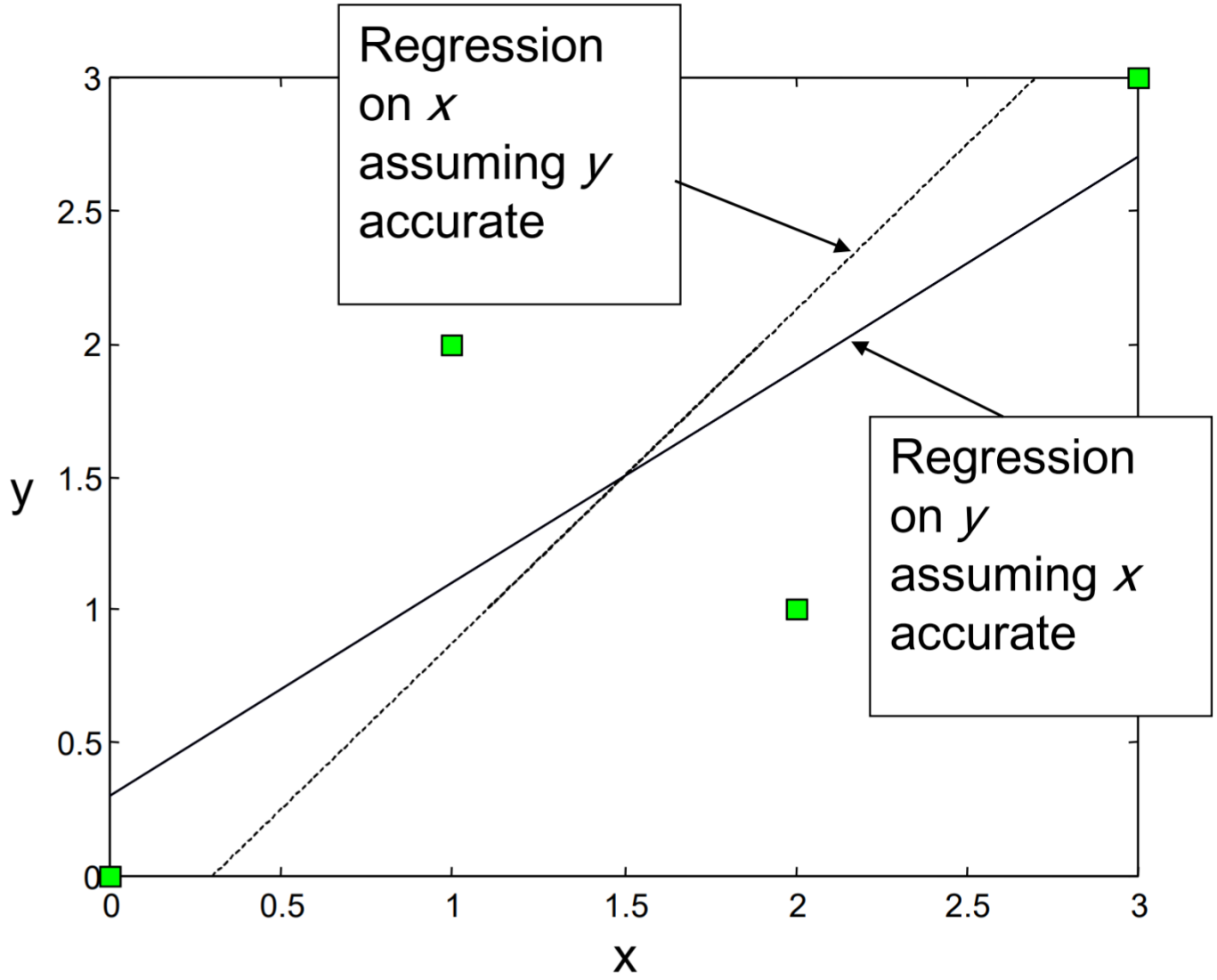
If we take the other assumption,

y - values are known exactly,

x - values are in error,

we get a different fit…

Error assumptions



Machine Learning and Linear Regression

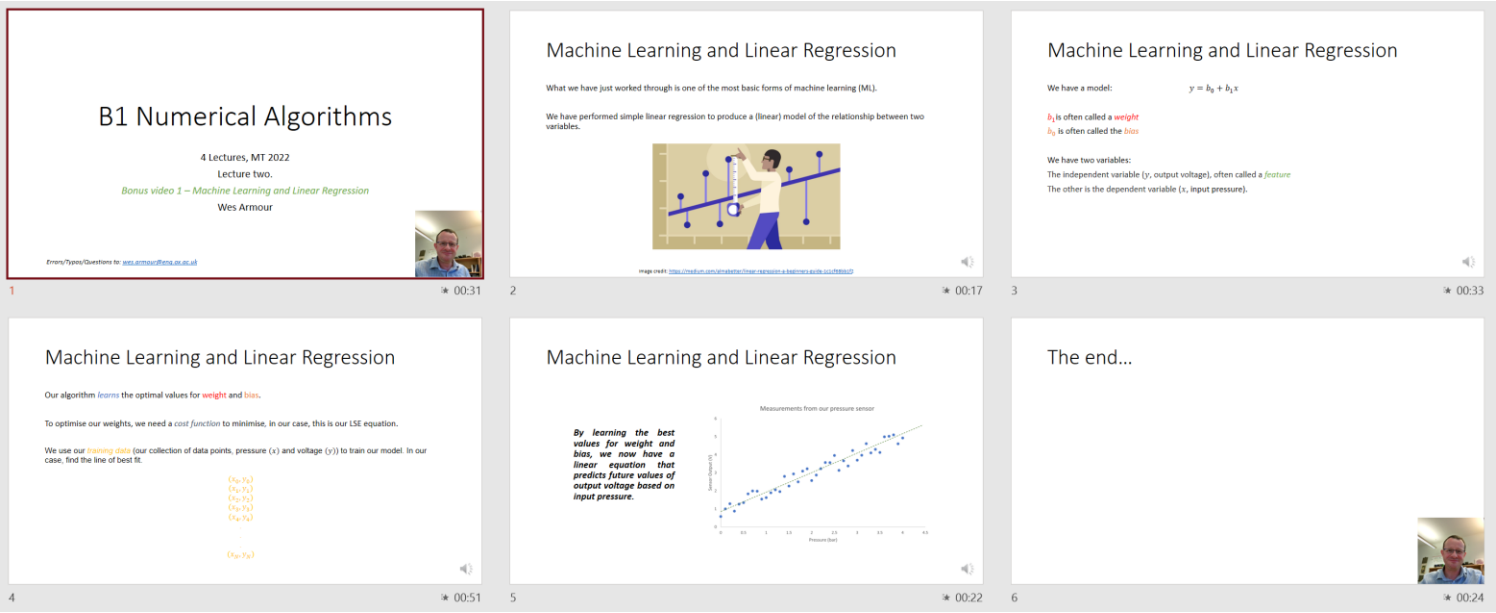
What we have just worked through is one of the most basic forms of machine learning (ML).

We have performed simple linear regression on our training data (x_i, y_i) to produce a (linear) model of the relationship between two variables.

Image credit: https://medium.com/almabetter/linear-regression-a-beginners-guide-1c1cf68bb1f2

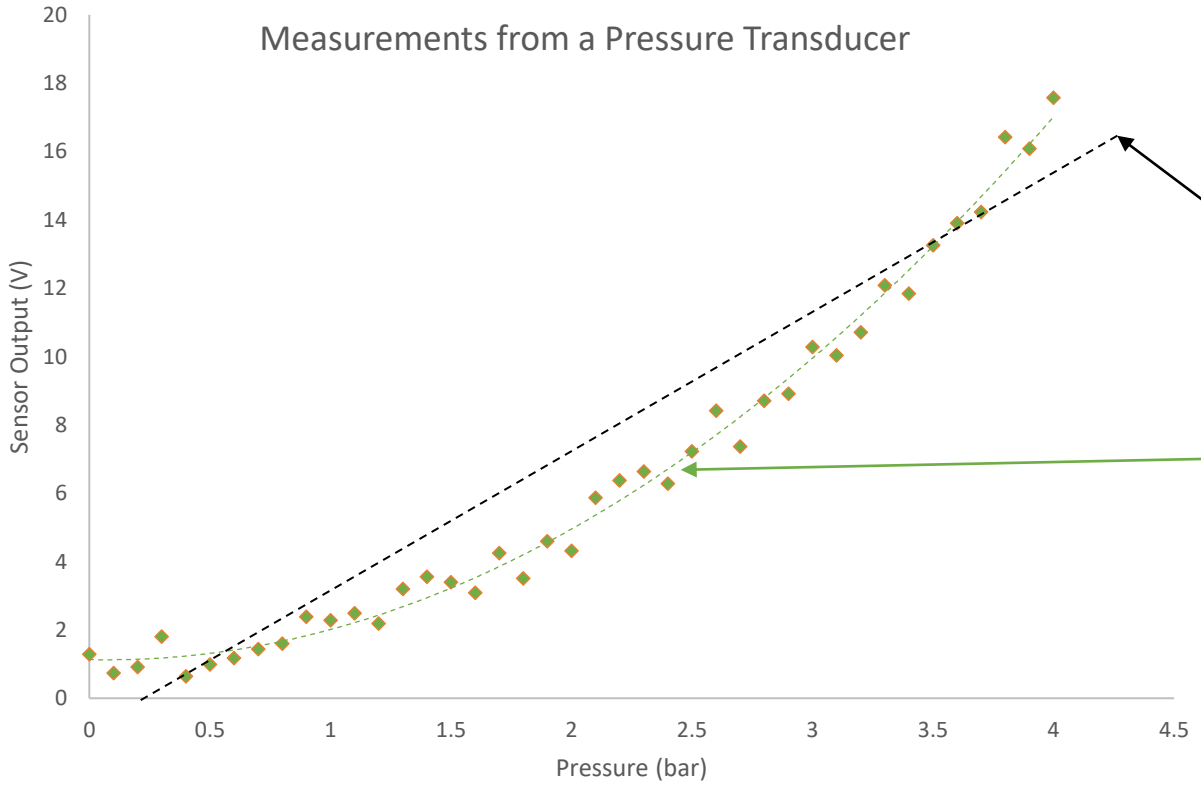
Machine Learning and Linear Regression

Panopto bonus video



Part B – Polynomial regression and overfitting

Polynomial Regression



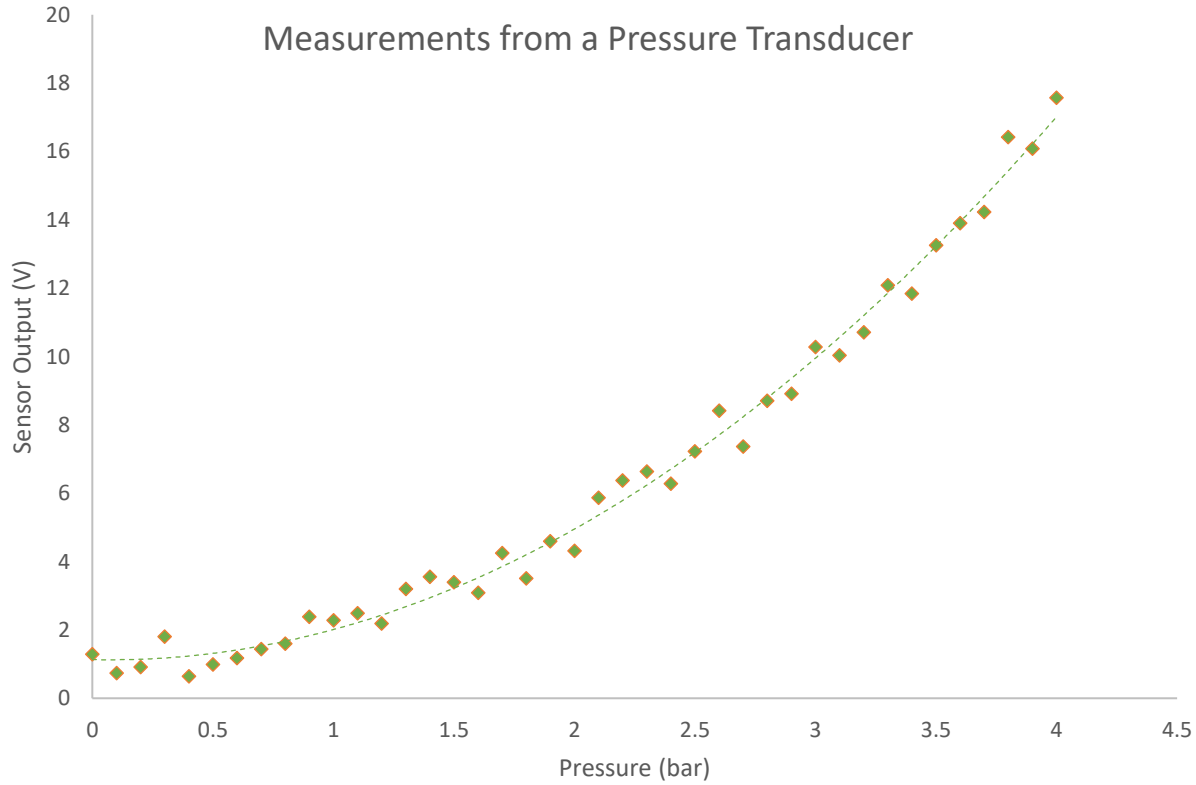
Consider the output from a ***non-linear*** pressure transducer (left).

A straight line fit will be a poor descriptor of our data and will have lots of error associated with it.

It's easy to generalise our previous example to a quadratic fit.

Lets see how…

Polynomial Regression



To fit the quadratic we now have:

$$\epsilon = \frac{1}{N}\sum_{i=0}^{N}(f(x_i) - y_i)^2$$

$$\epsilon = \frac{1}{N}\sum_{i=0}^{N}(\alpha_2 x_i^2 + \alpha_1 x_i + \alpha_0 - y_i)^2$$

Again the only things we can vary in the above equation (to influence the value of ϵ) are $\boldsymbol{\alpha_2, \alpha_1, \alpha_0}$

Polynomial Regression

We must minimise ϵ with respect to $\boldsymbol{\alpha_2, \alpha_1, \alpha_0}$ so we need to solve:

$$\frac{\partial \epsilon}{\partial \alpha_0} = 0, \ \frac{\partial \epsilon}{\partial \alpha_1} = 0, \frac{\partial \epsilon}{\partial \alpha_2} = 0$$

Applying the chain rule…

$$\frac{\partial \epsilon}{\partial \alpha_2} = \sum_{i=0}^{N} 2\left(\alpha_2 x_i^2 + \alpha_1 x_i + \alpha_0 - y_i\right)x_i^2 = 0$$

$$\frac{\partial \epsilon}{\partial \alpha_1} = \sum_{i=0}^{N} 2\left(\alpha_2 x_i^2 + \alpha_1 x_i + \alpha_0 - y_i\right)x_i = 0$$

$$\frac{\partial \epsilon}{\partial \alpha_0} = \sum_{i=0}^{N} 2\left(\alpha_2 x_i^2 + \alpha_1 x_i + \alpha_0 - y_i\right) = 0$$

Rearranging gives:

$$\sum_{i=0}^{N}\left(\alpha_2 x_i^4 + \alpha_1 x_i^3 + \alpha_0 x_i^2 - y_i x_i^2\right) = 0$$

$$\sum_{i=0}^{N}\left(\alpha_2 x_i^3 + \alpha_1 x_i^2 + \alpha_0 x_i - y_i x_i\right) = 0$$

$$\sum_{i=0}^{N}\left(\alpha_2 x_i^2 + \alpha_1 x_i + \alpha_0 - y_i\right) = 0$$

Hence we now have a set of 3 simultaneous equations and 3 unknowns $(\alpha_2, \alpha_1, \alpha_0)$…

Polynomial Regression

Writing in matrix form:

$$\begin{pmatrix} \sum_{i=0}^{N}(x_i^4) & \sum_{i=0}^{N}(x_i^3) & \sum_{i=0}^{N}(x_i^2) \\ \sum_{i=0}^{N}(x_i^3) & \sum_{i=0}^{N}(x_i^2) & \sum_{i=0}^{N}(x_i) \\ \sum_{i=0}^{N}(x_i^2) & \sum_{i=0}^{N}(x_i) & N \end{pmatrix} \begin{pmatrix} \alpha_2 \\ \alpha_1 \\ \alpha_0 \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{N}(y_i x_i^2) \\ \sum_{i=0}^{N}(y_i x_i) \\ \sum_{i=0}^{N}(y_i) \end{pmatrix}$$

Again we solve the matrix equation:

$$\underline{A}\,\underline{\alpha} = \underline{B}$$

Solving for $\underline{\alpha}$

$$\underline{\alpha} = \underline{A}^{-1}\underline{B}$$

These are known as the ***normal equations*** for the function fitting problem

General Polynomial Regression

Note the change from linear to quadratic…

$$\begin{pmatrix} \sum_{i=0}^{N}(x_i^2) & \sum_{i=0}^{N}(x_i) \\ \sum_{i=0}^{N}(x_i) & N \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_0 \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{N}(y_i x_i) \\ \sum_{i=0}^{N}(y_i) \end{pmatrix}$$

$$\begin{pmatrix} \sum_{i=0}^{N}(x_i^4) & \sum_{i=0}^{N}(x_i^3) & \sum_{i=0}^{N}(x_i^2) \\ \sum_{i=0}^{N}(x_i^3) & \sum_{i=0}^{N}(x_i^2) & \sum_{i=0}^{N}(x_i) \\ \sum_{i=0}^{N}(x_i^2) & \sum_{i=0}^{N}(x_i) & N \end{pmatrix} \begin{pmatrix} \alpha_2 \\ \alpha_1 \\ \alpha_0 \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{N}(y_i x_i^2) \\ \sum_{i=0}^{N}(y_i x_i) \\ \sum_{i=0}^{N}(y_i) \end{pmatrix}$$

Generalise using the ***Vandermonde*** matrix…

$$\underline{V} = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^M \\ 1 & x_2 & x_2^2 & \dots & x_2^M \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_N & x_N^2 & \dots & x_N^M \end{bmatrix}$$

Expressing in matrix-vector form:

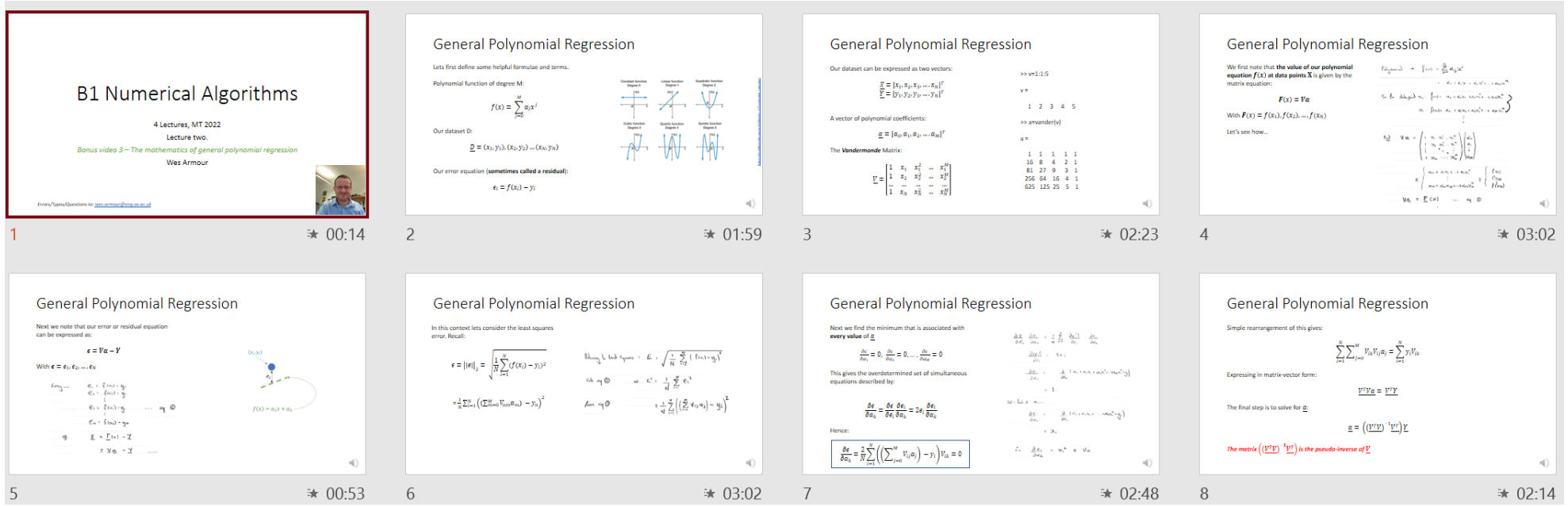
$$\underline{V^T V \alpha} = \underline{V^T Y}$$

The final step is to solve for $\underline{\alpha}$:

$$\underline{\alpha} = \left(\left(\underline{V^T V} \right)^{-1} \underline{V^T} \right) \underline{Y}$$

The mathematics of General Polynomial Regression

Panopto bonus video



1 ★ 00:14 2 ★ 01:59 3 ★ 02:23 4 ★ 03:02

5 ★ 00:53 6 ★ 03:02 7 ★ 02:48 8 ★ 02:14

General Polynomial Regression.

Now lets briefly cover two different cases. These are Interpolation and Approximate curve fitting.

Interpolation $M + 1 \geq N$

The polynomial is of higher order than the number of data points. The interpolated polynomial goes through all the data points. All the errors can be reduced to zero. (See Lecture Notes for the case M+1 = N)
But leads to "polynomial wiggle" at the endpoints! – splines fix this.

Approximate curve fitting $M + 1 < N$

The polynomial is of lower order than the number of data points. The interpolated polynomial does not go through all the data points. Generally the errors cannot be reduced to zero (except in the unlikely case of the data points lying on a polynomial!).

We consider only the curve approximating case here

An example of a polynomial fit

Let us try a real polynomial fit. First let us construct some data, with a little noise added:

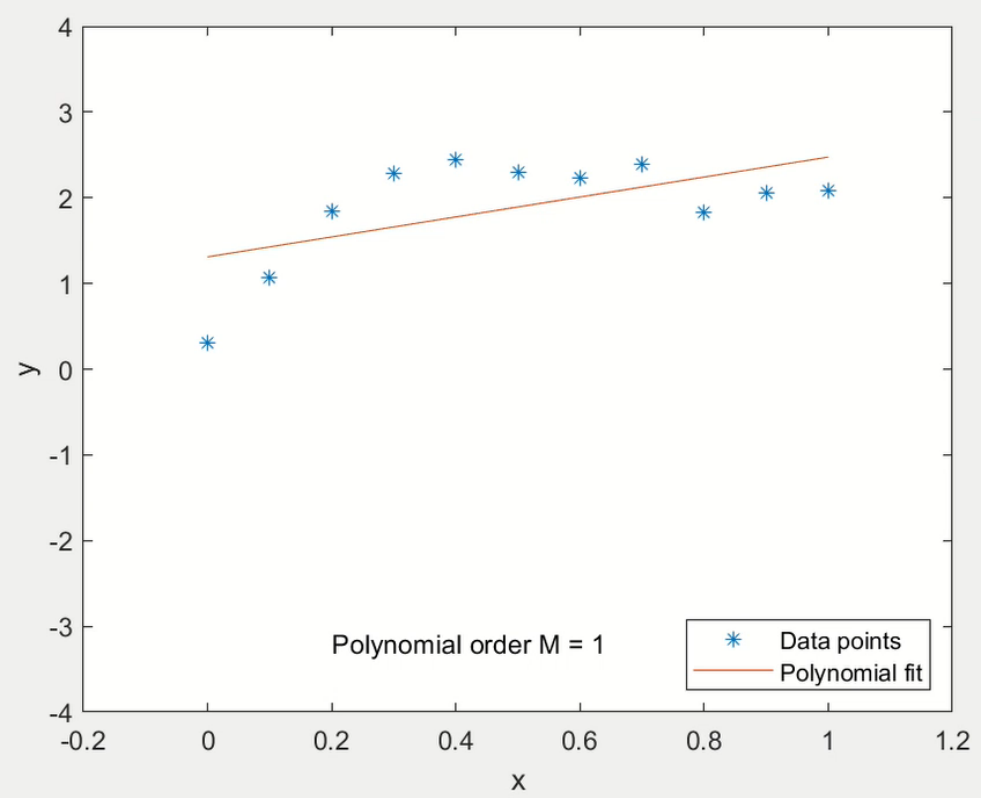
```
x=[0:0.1:1]’;
y=3*x + sin(1.5*pi*x) + 0.5*rand(length(x),1);
plot(x,y,'*’)
```

Now write a function to carry out a polynomial fit:

Fitting code

```
function f1=regress1(x,y,M,x1)
%   fits and plots Mth order polynomial to (x,y)
%   Plots fitted data at points of x1

V = zeros(length(x),M+1);    %construct Vandermonde matrix
for i = 0:M
   V(:,i+1) = x.^i;
end

VI = inv(V'*V)*V';  %Pseudo inverse matrix

alpha = VI*y;    %Compute polynomial coefficients

V1 = zeros(length(x1),M+1); %construct Vandermonde matrix for x1
for i = 0:M
   V1(:,i+1) = x1.^i;
end

f1 = V1*alpha;  %evaluate polynomial at each x1

%plot(x,y,'*',x1,f1);    %plot data and fitted polynomial
```

Results



Running this on our data, and plotting against the x values x1 = [0:0.01:1]; gives a fit that gets better as M increases.

Once M reaches 3, 4, 5 we get a reasonable fit to our data (notice the polynomial doesn't change very much).

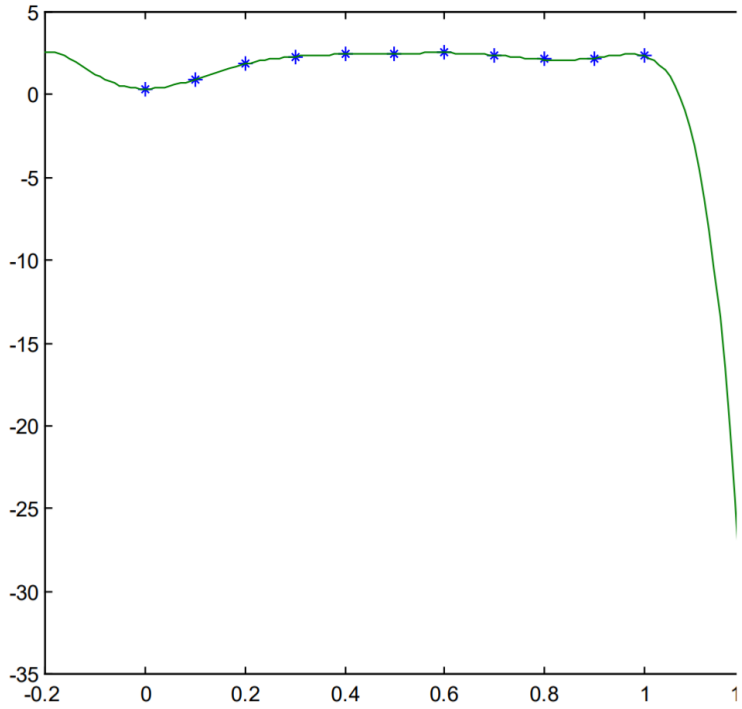
Once M reaches 8, 9, 10 the polynomial travels through every data point – we are fitting noise. The polynomial starts to look rather ***"ripply"*** at the ends.

With M greater than 10, we no longer have enough data to constrain the fit. The polynomial **no longer fits!**

Overfitting

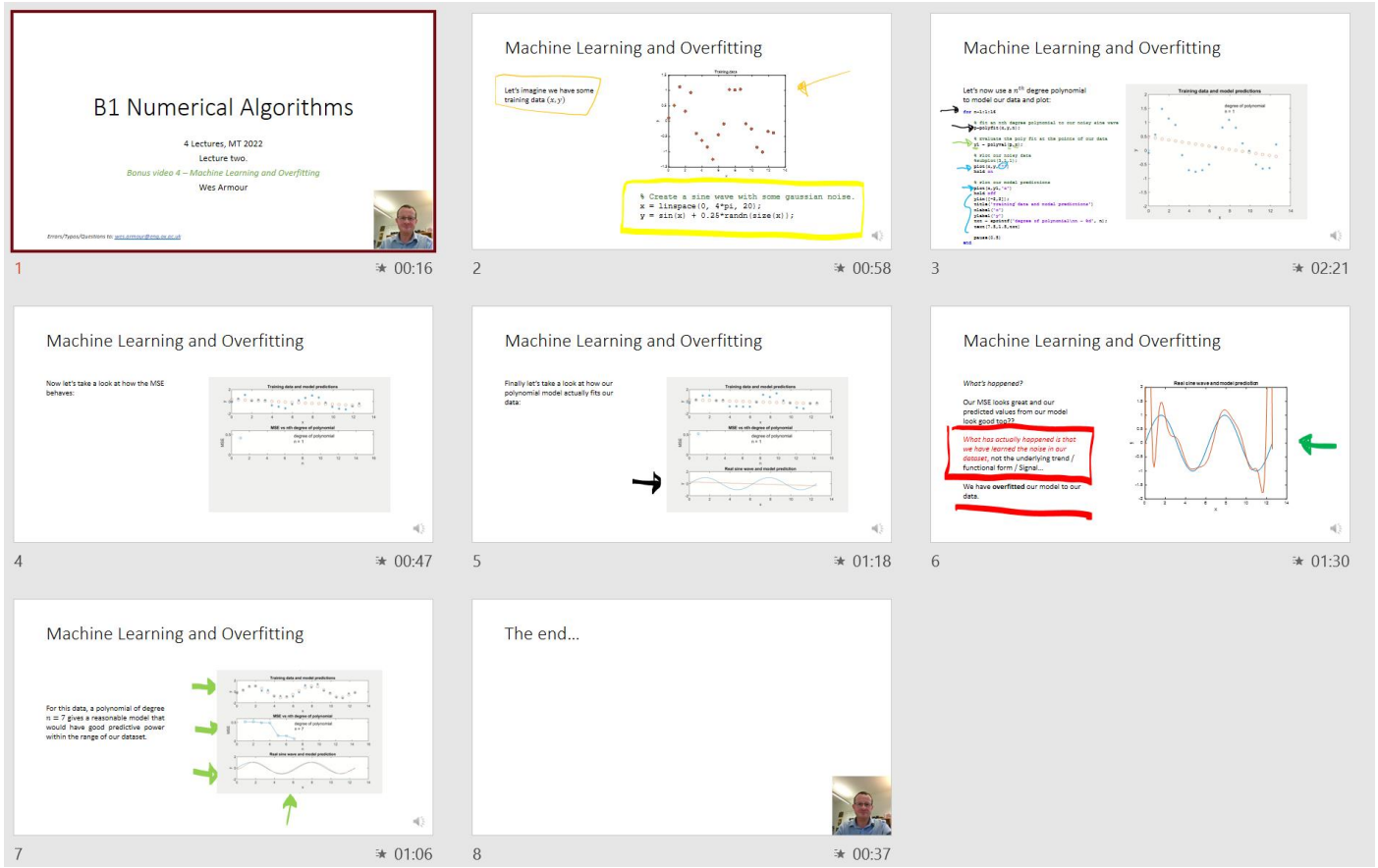
Life gets even worse when we try and use the polynomials to **extrapolate beyond the limits of the original data**, with the 8th order polynomial (See right).

In practice, high order polynomial fits tend to diverge wildly outside the source data, and do not give a good approximation to the original data.



Machine Learning and Overfitting

Panopto bonus video

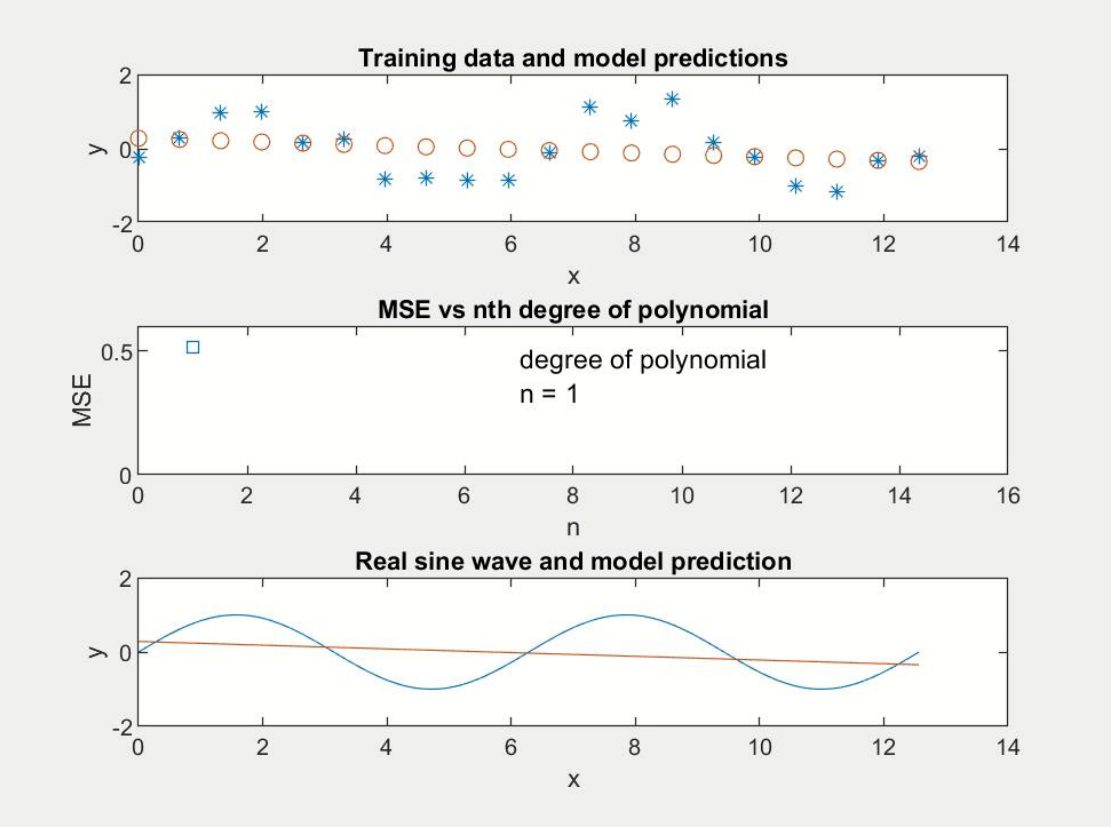


Machine Learning and Overfitting

This example fits a polynomial of degree M to a noisy sine wave.

Look at the Mean Squared Error as the polynomial degree changes.

Finally let's take a look at how our polynomial model actually fits our data:



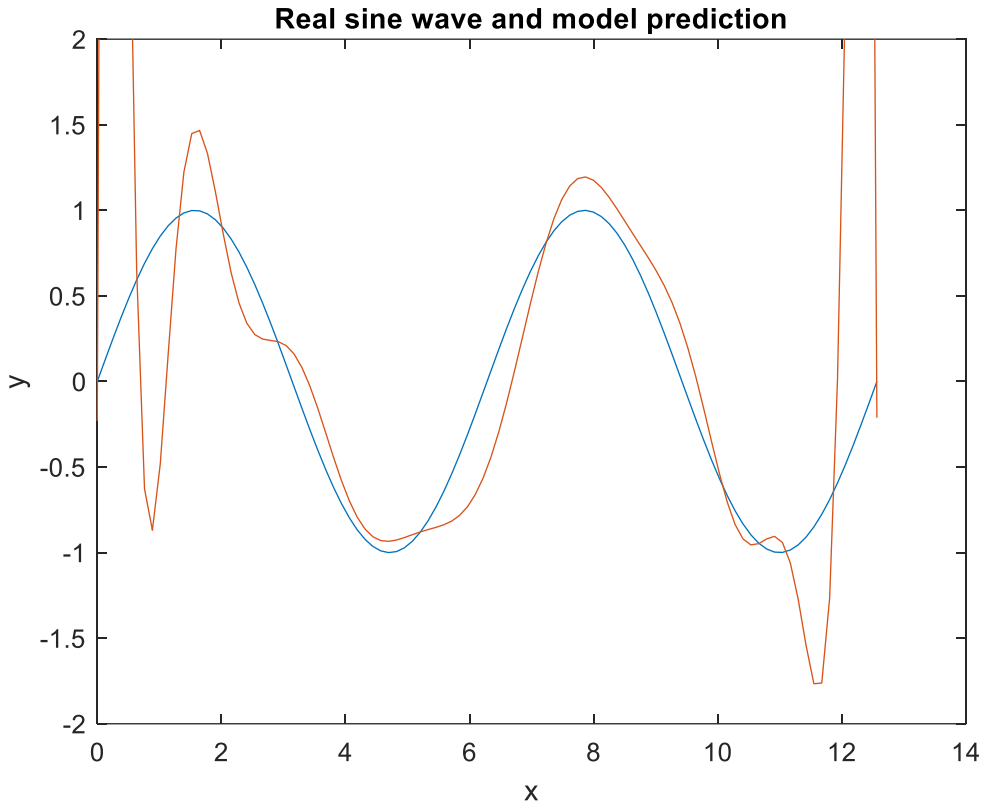
Machine Learning and Overfitting

What's happened?

Our MSE looks great and our predicted values from our model look good too??

What has actually happened is that we have learned the noise in our dataset, not the underlying trend / functional form / Signal…

We have **overfitted** our model to our data.



Validating our model

How do we know when we have fitted our model (our polynomial) correctly?
Or put another way, how do we evaluate our model's skill at prediction?

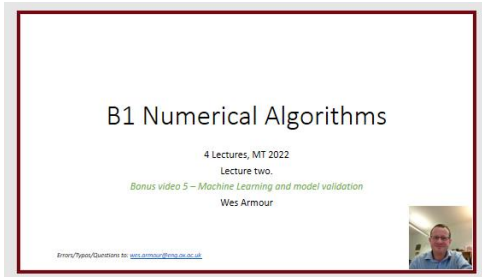
We use validation data!

Here you can see how I've held back data from our model training (yellow squares), and used this to test the predictive skill of our model (MSE, third plot, stars). We can see when $n \sim 7$ our model has the lowest MSE for the training data and the validation data that it hasn't seen before.

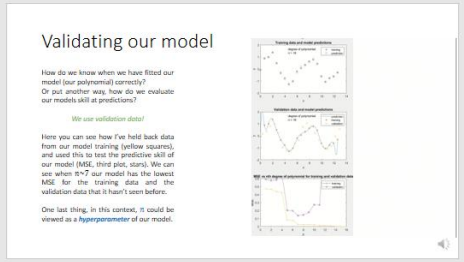
One last thing, in this context, n could be viewed as a ***hyperparameter*** of our model.

Machine Learning and Model Validation

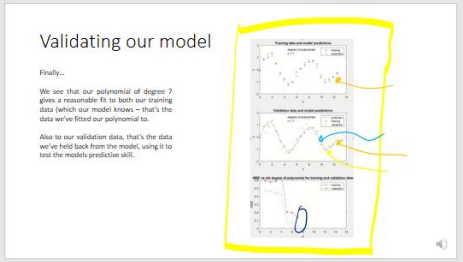
Panopto bonus video



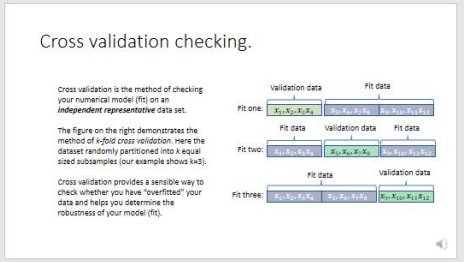
1 ★ 00:12



2 ★ 03:02



3 ★ 01:27



4 ★ 01:35

Recommendations for Polynomial fitting:

1. Use the lowest order polynomial that gives a reasonable approximation to the data. Do not ***overfit*** the data, or the fitted polynomial will follow every bit of the noise on the data.

2. Realise that the fit may be bad near the ends of the data.

3. Do not use the polynomial approximation outside the range of the input data. i.e. **Do not extrapolate the data!**

Far more information is given in the lecture notes covering divided differences and the properties of polynomial fits to data.

MATLAB routines for polynomial fits.

MATLAB has good routines:

polyfit and polyval to fit and evaluate polynomial approximations.

But note that MATLAB assumes that the vector alpha stores the coefficients of the polynomials in the opposite order, i.e. coefficient of x^N first.

You may need to use flipud or fliplr to get them in the desired order.

Part C – The condition number

Condition Number

The condition number essentially quantifies

How large the change in output is for a small change in input.

$$\kappa(A) = \begin{cases} \left||A|\right|\left||A^{-1}|\right| & Non\ Singular \\ \infty & Singular \end{cases}$$

Here $\left||A|\right|$ is the matrix norm.

Various forms of norm exist and are all equally useful.

https://academic.oup.com/qjmam/article/1/1/287/1883483 (Turing 1948)

Ill-conditioning

Since the polynomial coefficients came from the solution of the simultaneous linear equation set

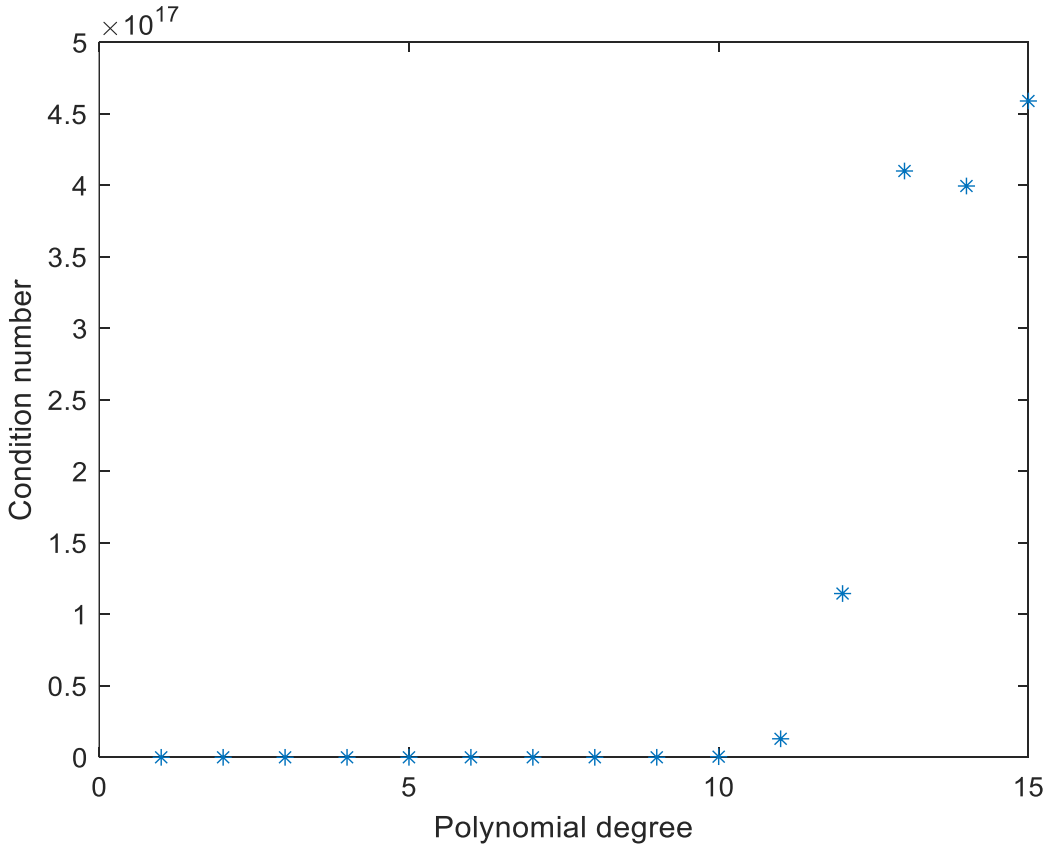
$$V^T V\, \alpha = V^T Y,$$

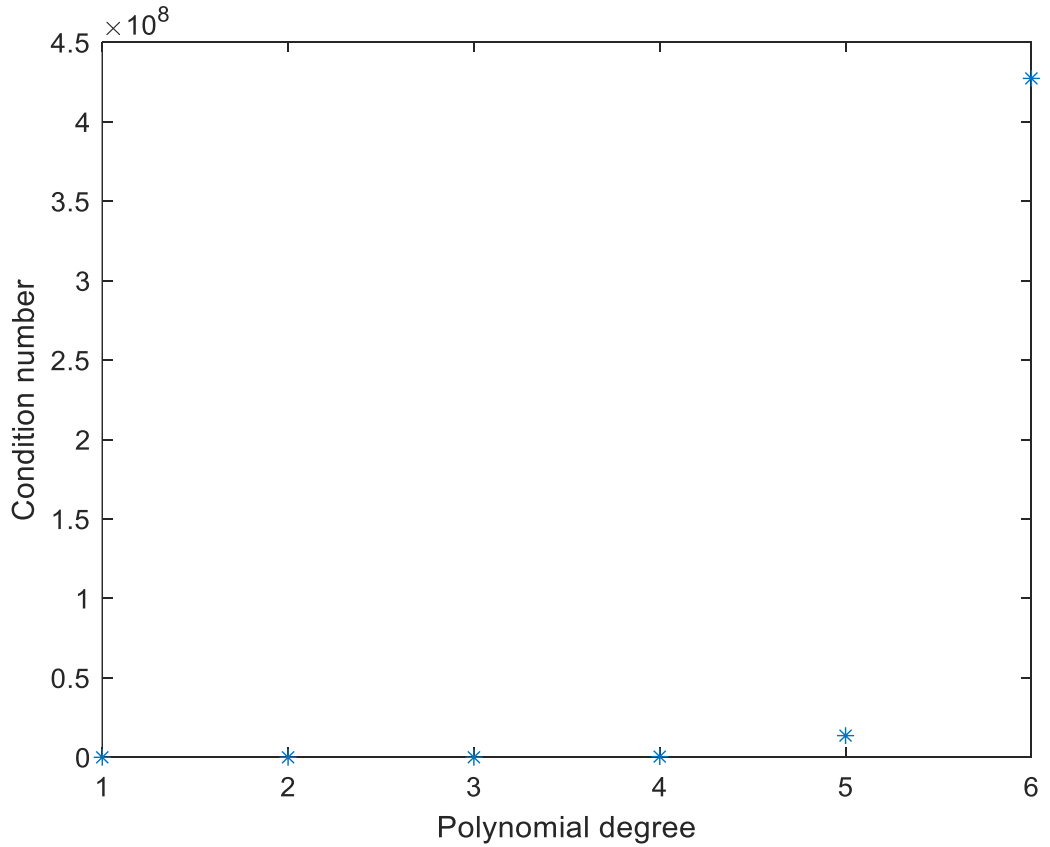
the Ill-conditioning can be examined by evaluating the *condition number* $K(V^T V)$ using the MATLAB command cond(V'*V).

For our example, this gives condition numbers which grow wildly!:

m = 1 condition number = 16.1631
m = 2 condition number = 379.754
m = 3 condition number = 10119.7
m = 4 condition number = 298450
m = 5 condition number = 9.84817e+006
m = 6 condition number = 3.73265e+008
m = 7 condition number = 1.6942e+010
m = 8 condition number = 9.84243e+011
m = 9 condition number = 8.25222e+013
m = 10 condition number = 1.37408e+016

Ill-conditioning





Part D – Bringing things together…

Numerical differentiation in the presence of noise and gradient descent.

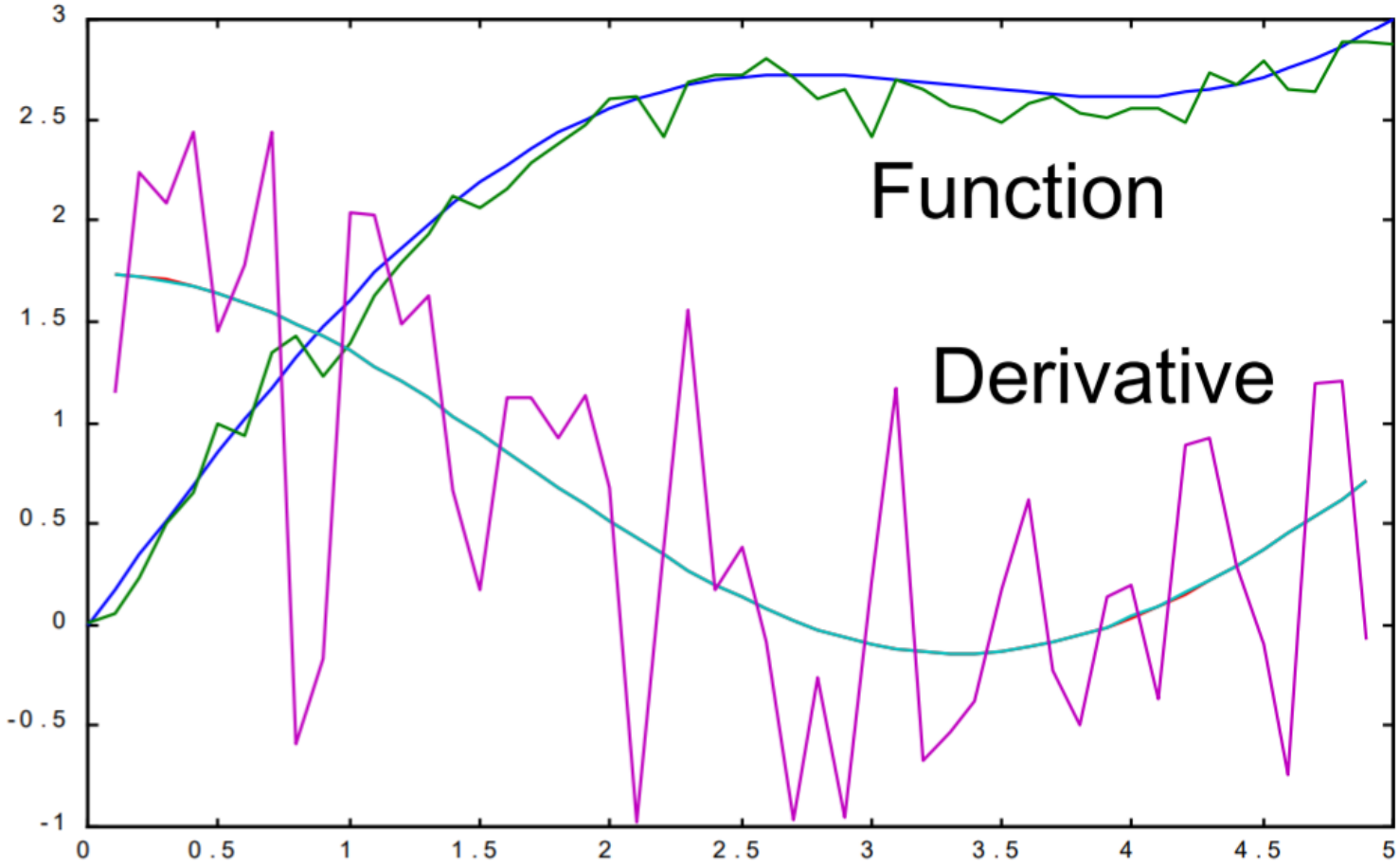
Differentiation and noise

The numerical differentiation process amplifies any noise in your data. Consider using the central difference formula with h = 0.1 to find the derivative of $\sin(x)$ with little added noise, using a MATLAB m-file:

```
%   diff1.m
%plots the differential coefficient of noisy data.

h=0.1;   %set h
x = [0:h:5]';  %data range.
n=length(x);
x2 = x(2:n-1);     %trim x

fn = 0.8*x + sin(0.3*pi*x);      %function to differentiate.

dfn = 0.8 + 0.3*pi*cos(0.3*pi*x);  %exact differential
dfn = dfn(2:n-1);%trim to n-2 points

fn2 = fn + 0.1*(randn(n,1)-0.5);   %add a little noise

df1=zeros((n-2),1);   %initialise non-noisy diff.
df2=zeros((n-2),1);   %initialise noisy diff.

for i=2:n-1
  df1(i-1)=(fn(i+1)-fn(i-1))/(2*h);  %non-noisy differential
  df2(i-1)=(fn2(i+1)-fn2(i-1))/(2*h);     %noisy differential
end

plot(x,fn,x,fn2,x2,dfn,x2,df1,x2,df2)   %plot functions and derivatives
```

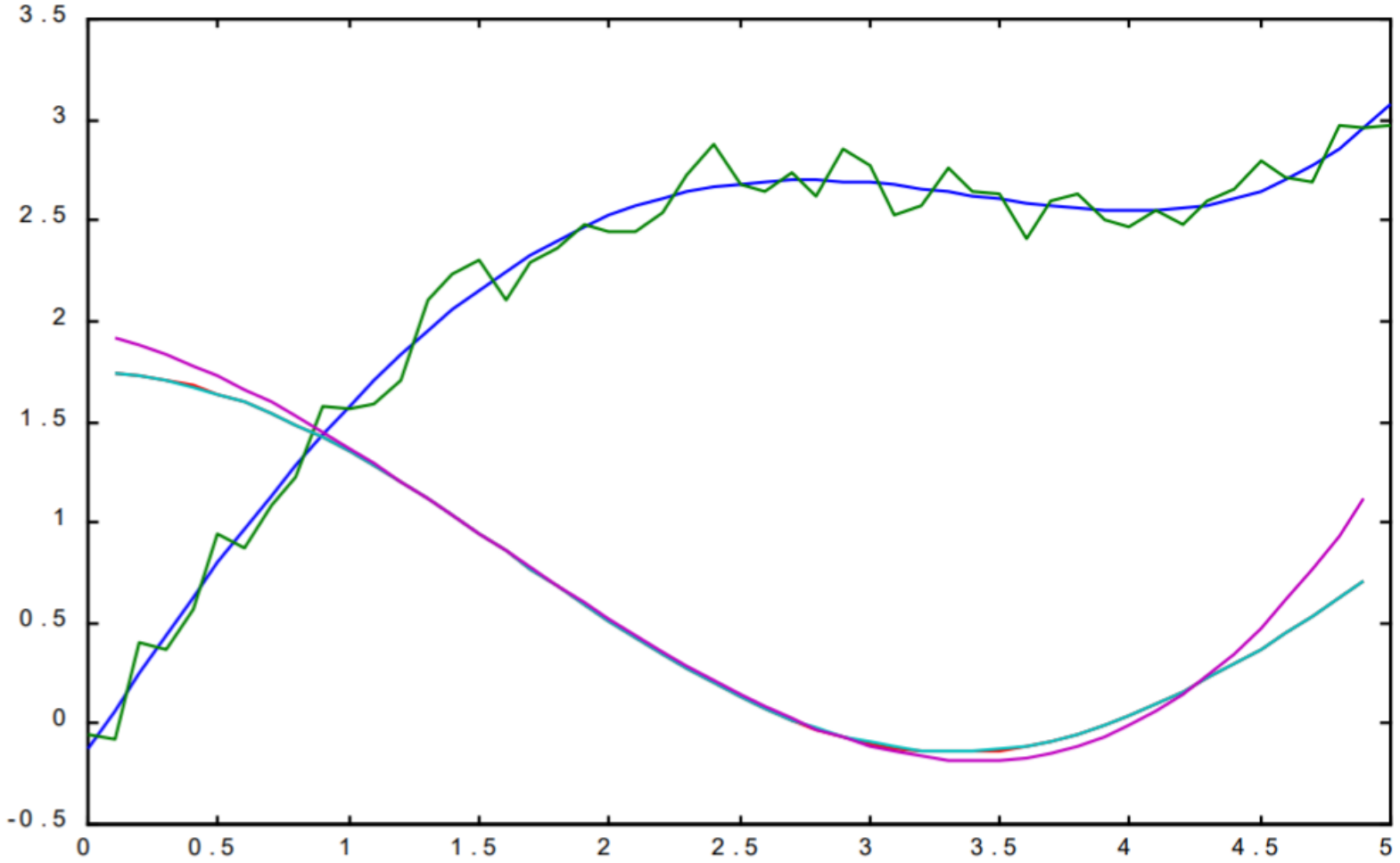
Matlab note: for loops are not efficient and are better avoided

Differentiation and noise



Even though the noise on the function is relatively small, the noise in the derivative is horrendous.

Differentiation and noise

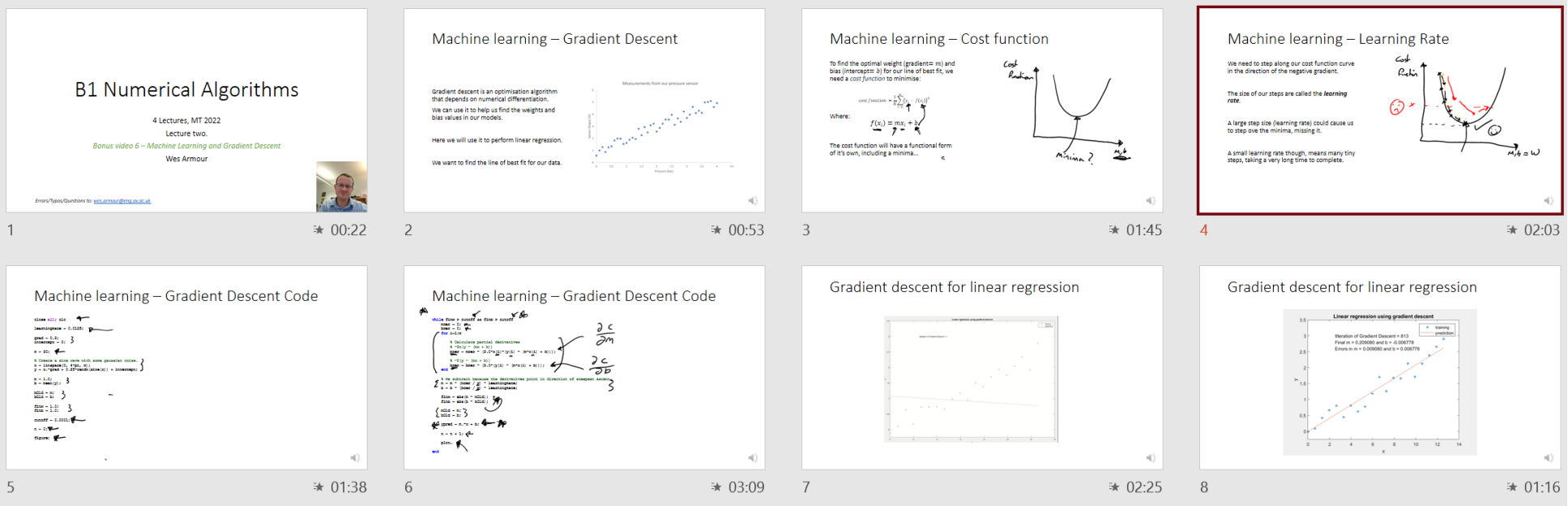


One solution to this problem is to fit a polynomial to the original function and then differentiate that.

Fitting a 4th order polynomial to the noisy data gets a smooth differential coefficient (even if it is off at the ends).

Machine Learning – Gradient Descent

Panopto bonus video



Conclusions – what have we learnt?

- How to perform linear regression.
- General polynomial regression and curve fitting.
- Condition number and how it tells us something about the stability of a fit.
- How noise impacts numerical differentiation and how to beat it.

Further reading - Approximation of functions.

We now turn to a second important topic in approximation theory. This relates to deriving a simple polynomial function f(x) to fit to an original function g(x), that can be used to approximate the original function. This is particularly useful when the original function is difficult to differentiate or integrate.

To introduce this topic we need to give some definitions

Approximation of functions.

Functional Norms

We need a norm:

the error in the fit, $e(x) = f(x) - g(x)$ can be quantified as an L_2 norm:

$$\|e\| = \left(\int_0^1 (f(x) - g(x))^2 \, dx \right)^{\frac{1}{2}}$$

In fitting polynomials to functions, minimising this norm gives a *least squares fit*.

[Note that we have integrated over the interval from 0 to 1. this is OK, since any interval can be scaled in to 0 - 1].

Approximation of functions.

Inner Product

We need an equivalent to the vector scalar product $\mathbf{x}^T\mathbf{y}$ (or **x.y**) for continuous functions:

One possible *Inner Product* is

$$\langle f, g \rangle = \int_0^1 f(x) g(x) \mathrm{d}x \, .$$

Approximation of functions.

Formula for Polynomial coefficients

It is simple to see that the L_2 norm can be expressed in terms of inner product:

$$\|e\|^2 = \langle e, e \rangle$$

To find the least squares fit for a polynomial

$$f(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \cdots + \alpha_M x^M = \sum_{m=0}^{M} \alpha_m x^m$$

we need to solve the equations $\dfrac{\partial \|e\|^2}{\partial \alpha_m} = 0, \quad m = 0, \cdots, M$

for the polynomial coefficients $\boldsymbol{\alpha} = [\alpha_0, \alpha_1, \alpha_2, \cdots \alpha_M]^{\mathrm{T}}$.

Approximation of functions.

By differentiating by parts,

$$\frac{\partial \|e\|^2}{\partial \alpha_m} = \frac{\partial \langle e, e \rangle}{\partial \alpha_m} = 2\left\langle e, \frac{\partial e}{\partial \alpha_m} \right\rangle .$$

Now, $$\frac{\partial e}{\partial \alpha_m} = \frac{\partial (f - g)}{\partial \alpha_m} = \frac{\partial f}{\partial \alpha_m} = x^m \ .$$

So the equations become

$$\left\langle e, x^m \right\rangle = 0 \ \text{ or } \ \left\langle f, x^m \right\rangle = \left\langle g, x^m \right\rangle \ \text{ for each } \ m = 0, 1, \ldots, M.$$

Expressing the polynomial f in terms of the coefficients α_m, we get, in matrix-vector form,

$$\mathbf{A\alpha = b} \ \text{ where } \ A_{m,n} = \left\langle x^m, x^n \right\rangle \ \text{ and } \ b_m = \left\langle g, x^m \right\rangle .$$

These are the *Normal Equations* for the function fitting problem.

Approximation of functions.

$$A_{m,n} = \left\langle x^m, x^n \right\rangle = \int_0^1 x^{m+n} \mathbf{d}x = \frac{1}{m+n+1}$$

is a special matrix called the Hilbert Matrix.

(Note: This matrix element is expressed in powers of x whereas in the next slide we express the matrix using indexes into a Matlab array where the first row is row 1 NOT row 0 (those of you who also know 'C' will recognise the confusion generated by using both 1 and 0 offsets into arrays!))

Approximation of functions.

(see lecture 2 notes)

The Hilbert Matrix

When fitting polynomials to data in a later lecture, we will come across the Hilbert Matrix having elements (when written using Matlab 1-offsets) $H_{ij} = \frac{1}{i+j-1}$.

e.g. $\mathbf{H}_4 = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{bmatrix}$.

Approximation of functions.

```
>> for n=1:10
        fprintf('Condition number for Hilbert matrix of order %d = %f \n', n,  cond(hilb(n)));
   end
```

```
Condition number for Hilbert matrix of order 1 = 1.000
Condition number for Hilbert matrix of order 2 = 19.281
Condition number for Hilbert matrix of order 3 = 524.056
Condition number for Hilbert matrix of order 4 = 15513.738
Condition number for Hilbert matrix of order 5 = 476607.250
Condition number for Hilbert matrix of order 6 = 14951058.641
Condition number for Hilbert matrix of order 7 = 475367356.296
Condition number for Hilbert matrix of order 8 = 15257575550.015
Condition number for Hilbert matrix of order 9 = 493154109752.877
Condition number for Hilbert matrix of order 10 = 16024922771324.436
```

Hilbert Matrices rapidly become ill-conditioned
In the particular application you will consider this causes problems when fitting high order polynomials to data

Approximation of functions.

$$A_{m,n} = \left\langle x^m, x^n \right\rangle = \int_0^1 x^{m+n} \mathbf{d}x = \frac{1}{m+n+1}$$

This is the ill-conditioned *Hilbert Matrix*

So these equations are ill-conditioned for *M* large.

The cure is to use an *Orthonormal basis*. Rather than the simple polynomial basis we have used above.

i.e. to approximate *g*(*x*) by a sum of *orthonormal polynomials*

such as *Legendre or Chebychev polynomials*.

We discuss this further after an example of using the method……

Approximation of functions.

Example

Find the least squares approximating polynomial of degree one for the function $f(x) = e^x$ over the interval [0,2].

The approximating polynomial is $p(x) = a_1 x + a_0$.

The normal equations in this case are

$$a_0 \int_0^2 1dx + a_1 \int_0^2 xdx = \int_0^2 e^x dx$$

$$a_0 \int_0^2 xdx + a_1 \int_0^2 x^2 dx = \int_0^2 xe^x dx$$

Integrating out

$$2a_0 + 2a_1 = e^2 - 1$$

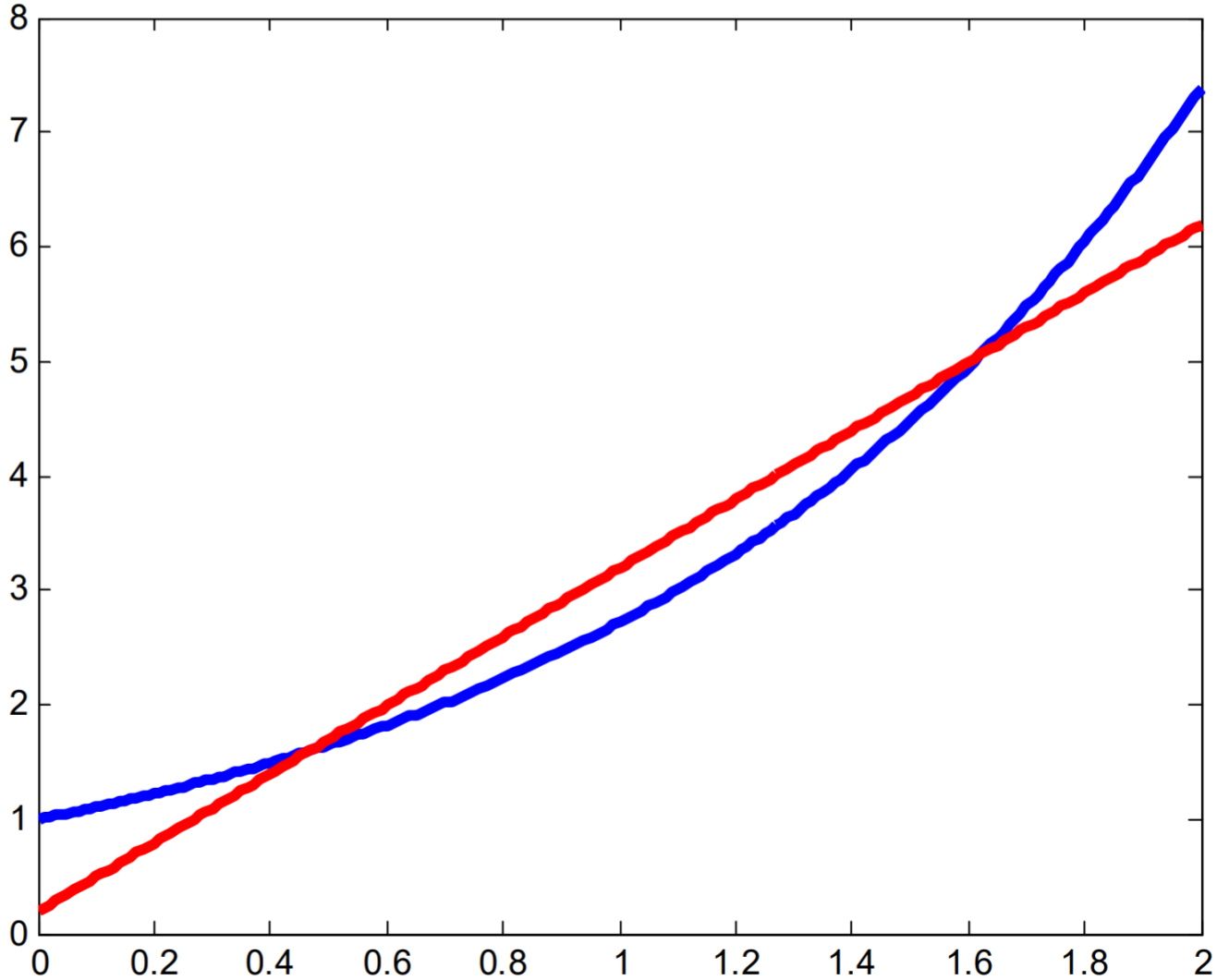
$$2a_0 + \frac{8}{3}a_1 = 1 + e^2$$

Subtracting these two equations (2)-(1) gives $a_1 = 3.0.$ Hence from equation (1)

$a_0 = 0.5(e^2 - 1 - 6) = 0.1945$

Hence the solution is $p(x) = 3.0x + 0.1945$

Approximation of functions.



Approximation of functions.

Chebyshev Polynomials

Chebyshev polynomials are an orthonormal basis and very useful for functional approximation.

They are defined by $T_n(x) = \cos\left(n \cos^{-1} x\right)$

Hence $T_0(x) = \cos\left(0 \cos^{-1} x\right) = 1$,

$T_1(x) = \cos\left(1.\cos^{-1} x\right) = x$ and

$T_2(x) = \cos\left(2\cos^{-1} x\right) = \cos 2\theta$

$= 2\cos^2\theta - 1 = 2x^2 - 1$

A useful recurrence relationship:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \quad .$$

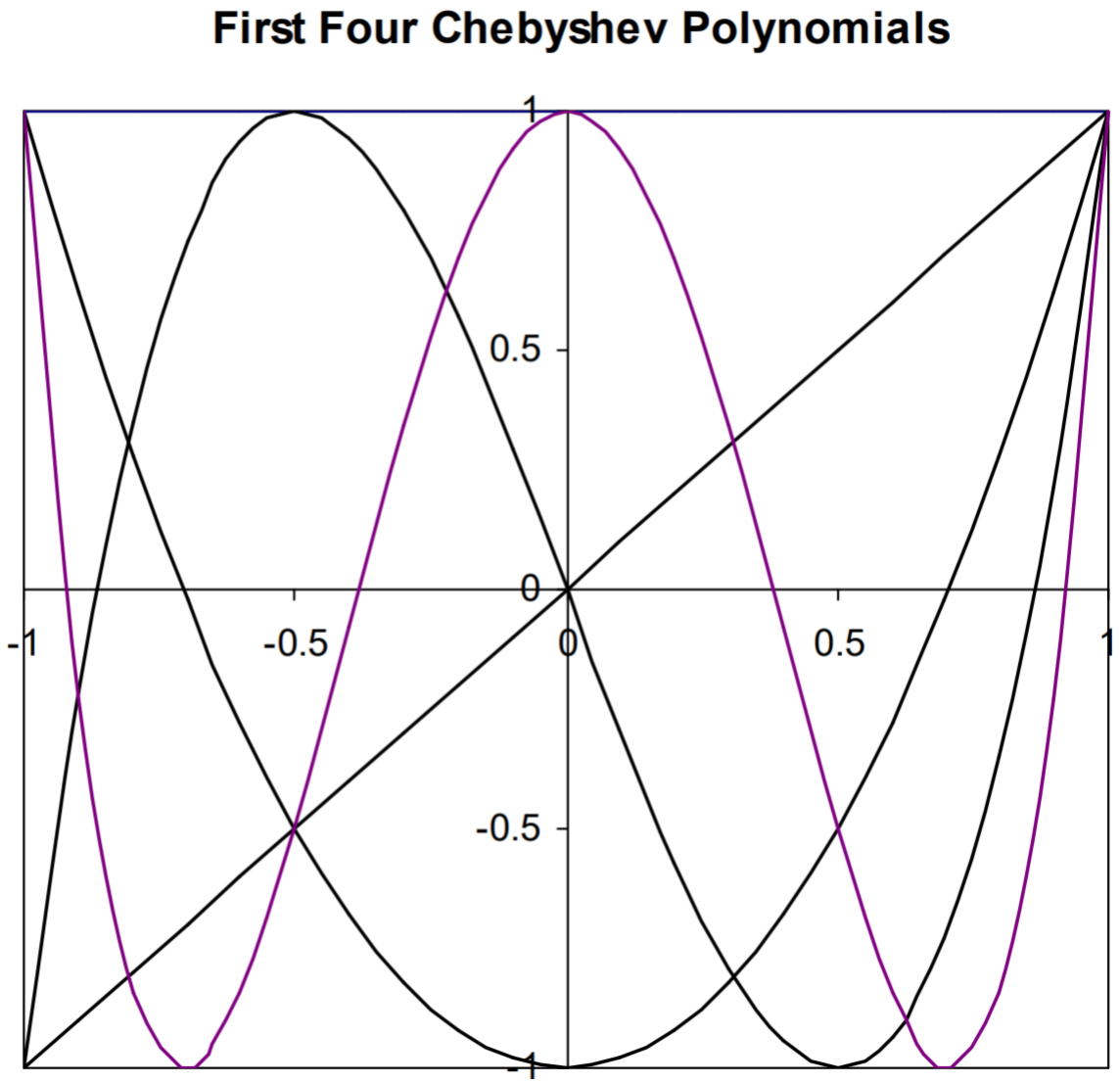
Use it to prove that $T_3(x) = 4x^3 - 3x$

$T_4(x) = 8x^4 - 8x^2 + 1$

Matlab:

```
>> syms x
chebyshevT([0, 1, 2, 3, 4], x)

ans =

[ 1, x, 2*x^2 - 1, 4*x^3 - 3*x,
8*x^4 - 8*x^2 + 1]
```

Approximation of functions.



Approximation of functions.

Chebyshev polynomials are *orthonormal* in the range $-1 \le x \le 1$

if we include a *weighting factor* $w(x) = \dfrac{1}{\sqrt{1-x^2}}$.

Then, you can easily show that the weighted inner product becomes

$$\left\langle T_m(x), T_n(x) \right\rangle = \int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} T_m(x) T_n(x) \mathrm{d}x$$

$$= \frac{\pi}{2} = \left\| T_n \right\|^2 \text{ for } m = n \neq 0$$

$$= \pi = \left\| T_0 \right\|^2 \text{ for } m = n = 0$$

$$= 0 \text{ for } m \neq n$$

Approximation of functions.

Then, if we approximate our function $g(x)$ by a *Chebyshev series*

$$f(x) = \alpha_0 T_0(x) + \alpha_1 T_1(x) + \alpha_2 T_2(x) + \cdots + \alpha_M T_M(x) = \sum_{m=0}^{M} \alpha_m T_m(x),$$

the inner product formula $\alpha_n \|T_n\|^2 = \int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} g(x) T_n(x) dx$ can be used to get the coefficients α.

It can be shown that the Chebyshev series thus obtained is the best polynomial approximation, in the sense of minimising $|f(x) - g(x)|_{\max}$ over $-1 \le x \le 1$.

This can be a useful property in some fitting problems where you can not tolerate a large deviation (eg in surface rendering/NC machining). See (Burden and Faires Ch8) for further discussions.

Approximation of functions.

Example of Chebyshev series

Find the second order Chebyshev approximation for the

semicircle $g(x)=\sqrt{1-x^2}$.

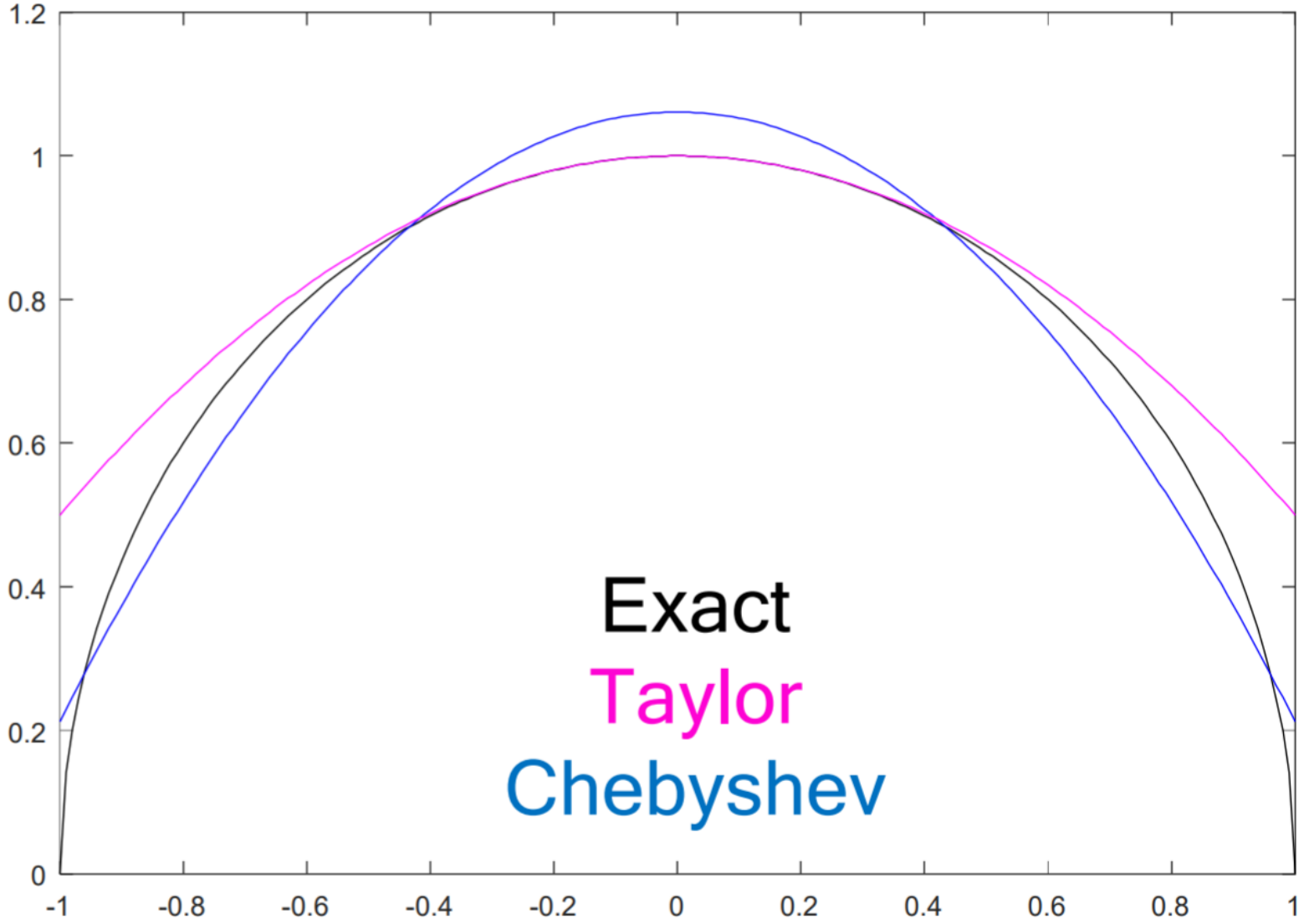
Solving $$\alpha_n \|T_n\|^2 = \int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} \sqrt{1-x^2}\, T_n(x)\mathrm{d}x = \int_{-1}^{1} T_n(x)\mathrm{d}x$$

gives the series $$f(x) = \frac{2}{\pi} - \frac{4}{3\pi}\left(2x^2 - 1\right)$$

A better approximation than

truncated Taylor series $y = 1 - \frac{x^2}{2}$.

Approximation of functions.



Approximation of functions.

Another useful orthogonal Basis: Legendre Polynomials

Legendre polynomials are orthogonal over [-1,1] and have a weighting function $w(x) = 1$.

The first four polynomials are

$$p_0(x) = 1, \quad p_1(x) = x, \quad p_2(x) = x^2 - \frac{1}{3}, \quad p_3(x) = x^3 - \frac{3}{5}x$$

You apply them in the same way as described for Chebyshev polynomials above.

Approximation of functions.

Other Points

1. As with data fitting, in function approximation the advantages of using algebraic polynomials are that you can take derivatives and integrals of them easily and also readily estimate arbitrary values.

2. The main disadvantage is that they can oscillate. This problem can be overcome by using **rational functions** that take on the form $r(x) = \frac{p(x)}{q(x)}$ where $p(x), q(x),$ are polynomials. These give better (in the sense of less oscillatory) fits than polynomials for the same amount of computation (see (Burden & Faires p507)).These are not considered on this course however.

Approximation of functions.

Other Points (2)

1. Another topic not in the syllabus, but we must mention are **cubic splines**. (See Kreyszig 7th Ed. pp 949-).

The name *spline* derives from an old draftsman's device: a flexible strip of wood which can be bent over pins placed at the knots to follow a smooth curve.

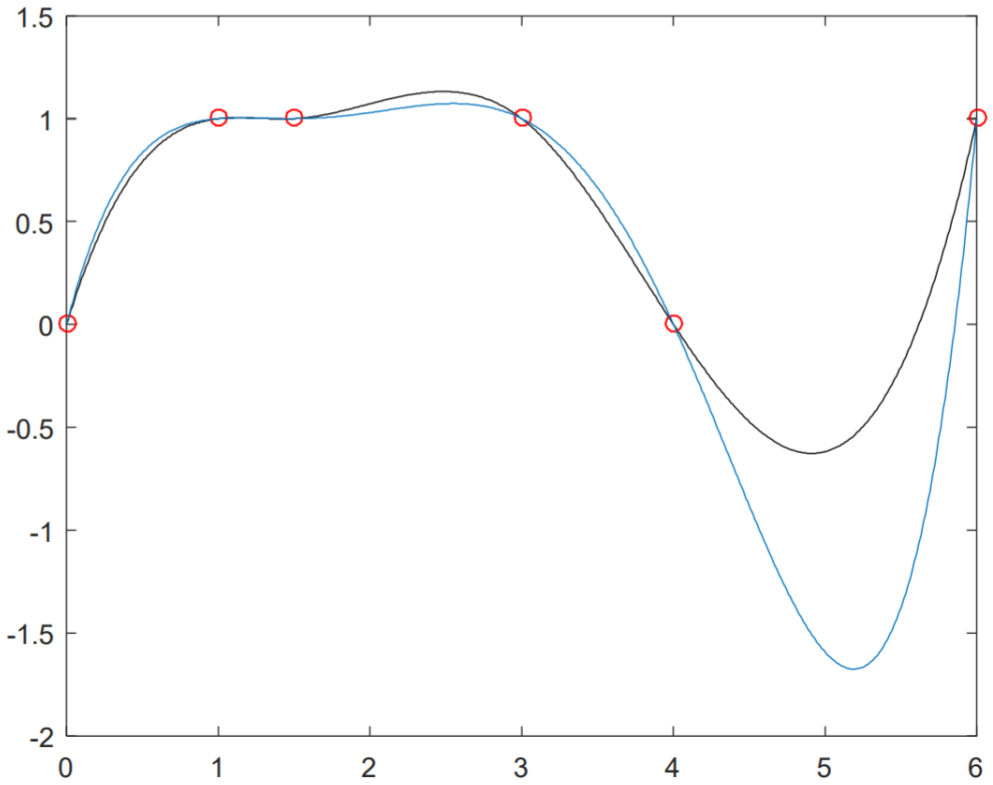
Cubic-spline based methods involve *local interpolation* between points (known as *knots*) in the *x-y* plane, to give a smooth curve between data points. Normally *cubic* polynomials are fitted between the knots, with continuous first and second derivatives across the knots. At the free ends, the second derivatives are often put to zero.

They are less susceptible to the oscillations than polynomial interpolation as disturbances do not propagate far.

Approximation of functions.

Other Points

Here is an example, comparing the MATLAB function spline (full curve) through 6 points, compared with a 5th order polynomial:



Spline

Polynomial