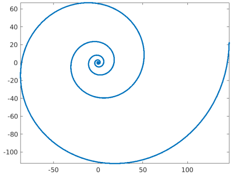
CS319: Scientific Computing (with MATLAB)

Figures and Fitting

Niall Madden

Week 5: **9am and 4pm**, 08 Feb 2023



Important: you should read:

- Chapter 5 of Learning MATLAB: https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9780898717662
- Chapter 8 of The MATLAB Guide: https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9781611974669

This week, in CS319:

Preview

At a glance, this week's class is all about plotting functions, in 1 and 2 dimensions.

But really it is about using such plots to get a better understanding of the problems we are working on.

For example, we will

- Use a plot to estimate the computational complexity of a linear solver;
- Investigate a least-squares problem

Last week we learned about working with matrices in MATLAB:

- You can define a matrix by listing its entries between square brackets. List entries by row, with a comma (or space) between columns, and a semicolon between rows.
- Use round brackets to access entries. Indexing is from 1. E.g., `A(2,4)` returns the entry in row 2, column 4 of A (assuming it exists). Possible errors:
  - `Unrecognized function or variable 'A'.`
  - `Index in position 1 exceeds array bounds. Index must not exceed 1.`
- Similarly, you can set an entry this way. E.g., `B(2,3)=4` sets the entry of B in row 2, column 2 to 4. If B does not exist, then it is created, with all entries, except `B(2,3)` set to zero.
- Vector indexing works as for vectors. E.g,
  - `A(1:2, 1:2)` refers to the 2×2 leading principle submatrix of A.
  - `A(:,3)` refers to all of column 3 of A.
- You can add, subtract, and multiply matrices using the usual arithmetic operators `+`, `-`, and `*`, respectively.
- "dot" operations are entry-wise.

- `inv(A)`

- `det(A)`

- `A'` is the transpose of A

- `eig(A)` estimate the eigenvalues and eigenvectors of A.

- `diag()` is a somewhat unusual function. Given an matrix as its argument, A, it returns the vector of diagonal entries of A.
  Given a vector as its argument, it returns a diagonal matrix with the vectors entries as its diagonal.
  Note: `B = diag(diag(A))` can be very useful. What do you think it does?

- `tril(A)`, `tril(A,k)`, `triu(A)` and `triu(A,k)`.

And there are lots of other functions that you may have met in a linear algebra module, but we wait until we need them.

2: Matrix division

For scalars (i.e, 1×1 matrices), "division" is well understood: we know what `a/b` means $\frac{a}{b} = ab^{-1}$. This is called "right division" in MATLAB.

MATLAB also has "left division": `a\b` means $\frac{b}{a} = a^{-1}b$.

The reason for this, is that, if `A` is a matrix, and `b` and `x` are vectors so that $Ax = b$, then, of course $x = A^{-1}b$.

Solving $Ax = b$

In MATLAB, if you are given a matrix A and vector b, then we usually solve $Ax = b$ with "backslash":

```
>> x = A\b
```

2: Matrix division

It is important to note that the "backslash" operator is highly optimised. And it does not compute the inverse of a matrix. More likely, it uses Gaussian elimination, or some variant (depends on the matrix).

MATLAB can invert a matrix, using the `inv(A)` function. But if you just wish to solve a linear system, backslash is is faster, and uses much less memory.

2: Matrix division

SolverTimerV01.m

```
for n=2.^(2:11)
   A = randn(n); b = ones(n,1);

   % Test matrix left divide
   mld_start = tic;
   x = A\b;
   mld_time = toc(mld_start);

   % Test inv()
   inv_start = tic;
   B = inv(A);
   x = B*b;
   inv_time = toc(mld_start);

   fprintf('n=%4d. MLD time=%6.3fs, inv time=%6.4fs (Speed up = %5.2f)\n', ...
      n, mld_time, inv_time, inv_time/mld_time);
end
```

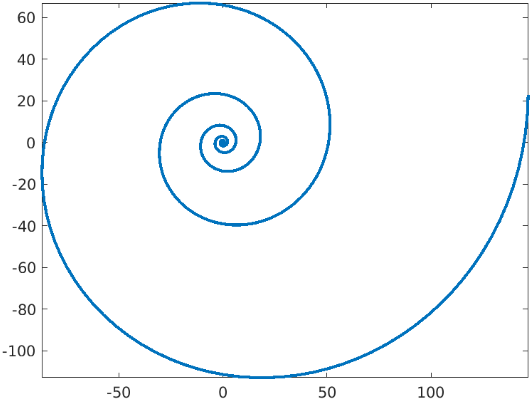
Try this. I get a speed-up of a factor of about 2.5. We'll return to this example later...

3: 2D Graphics

MATLAB is good at visualising functions and data. We have already seen some examples, such as fplot and fsurf, last week.

fplot is quite versatile, and can plot implicit functions:

```
>> fplot(@(t)exp(-t).*sin(6*t), @(t)exp(-t).*cos(6*t))
```



However, plotting such functions is relatively rare. It is more common to plot data points. For example, we'll plot US census population data, which comes with MATLAB for illustration purposes.

The `plot(x,y)` plots the vectors x and y, which must have the same number of entries, joining the points (x_1, y_1), (x_2, y_2), $\ldots$, (x_n, y_n).

Syntax:

Other examples Try these, and see the differences:

```
load census; % loads vectors cdate and pop
plot(cdate,pop)

plot(cdate,pop,'ro') % red circles

plot(cdate,pop,'k:') % black dotted lines

plot(cdate,pop,'ms--')

plot(cdate,pop,'ms--', 'LineWidth', 3, 'MarkerSize', 10)
```

3: 2D Graphics

Some useful functions for enhancing images:

```
title('string')

legend('string')

xlabel('string')

ylabel('string')

grid on
```

Also useful:

```
figure();

figure(n);

hold on;
hold off;

subplot(r,c,n)
```

One important application of plotting is to observe the growth of functions.

Example

In Section 2 we computed the time taken to solve linear systems by two different methods. Suppose we know that these times are polynomials in n, the order of the matrix. How can we determine to degree of the polynomials?

Here is how to proceed.

1. When we say T is a polynomial in n, we mean that
$$T(n) = c_0 + c_1 n + c_2 n^2 + \cdots + c_p n^p.$$
Our goal is to find a likely value for p. (Less import, but we'll also estimate c_p).
2. Usually, n is very big. So we can approximate T as
$$T(n) \approx c_p n^p.$$
3. Taking the logarithm we get
$$\log\big(T(n)\big) = \log(c_p) + p\log(n).$$
4. So p is the slope of the line when plotting $\log(T(n))$ against $\log(n)$. This is such a common task, that there is a built-in function, `loglog()` to do this.

Once plotted, we can use trial-and-error to work out the slope. The easiest/laziest approach would be to successively include plots to cN, cN^2, cN^3, until one of them looks good. Choose c so that this line agrees with one of the data points (e.g., the last one).

SolverTimerV02.m

```
for n=2.^(4:13)
    k=k+1;
    Ns(k) = n; % Should really pre-allocate
    A = randn(n,n);
    b = ones(n,1);

    % Test matrix left divide
    mld_start = tic;
    x = A\b;x = A\b;
    mld_time(k) = toc(mld_start);

    % Test inv()
    inv_start = tic;
    B = inv(A); x = B*b;
    inv_time(k) = toc(mld_start);
end
c = mld_time(end)/Ns(end)^3;
loglog(Ns, inv_time, '--o', Ns, mld_time, '-.d', ...
    Ns, c*Ns.^3, '-k','LineWidth', 2, 'MarkerSize', 10)
```

One of the reasons that "backslash" is so much faster than `inv()` is that it first analyses the matrix to see if there are any short-cuts. Eventually it falls back on using Gaussian Elimination (or equivalent).

To demonstrate that, and to use some functions we mentioned earlier, let's apply the same approach to solving a linear system where the matrix is triangular.

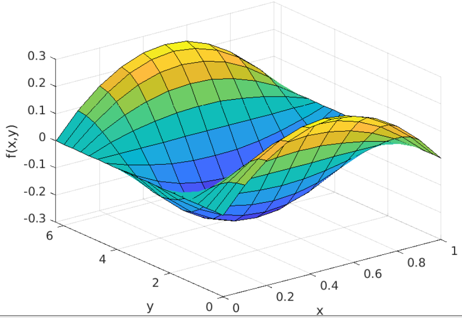
(See notes from class for explanation and final code).

4: 3D Graphics

Plotting surfaces is slightly more complicated than 2D plots. Often, one needs to create matrices, X and Y, Z where $Z(i,j) = f(X(i,j), Y(i,J))$.

Plot3D_demo.m

```
f = @(x,y)x.*(1-x).*cos(y);
x = linspace(0,1,11);
y = linspace(0,2*pi,21);
[X,Y]=meshgrid(x,y); % semi-colon is important here!
surf(X,Y,f(X,Y))
```



To explain this code...

Plot3D_demo.m

```
f = @(x,y)x.*(1-x).*cos(y);
x = linspace(0,1,11);
y = linspace(0,2*pi,21);
[X,Y]=meshgrid(x,y); % semi-colon is important here!
surf(X,Y,f(X,Y))
```

Some variation is possible. For example...

- Try using `grid()` instead of `surf()`.
- Once plotted, you can change the shading using one of
  `shading flat`   `shading faceted`   `shading interp`
- Very useful: use `colormap` to select the colours used. There are lots of options but the most useful are `hot`, `cool`, `gray`, `autumn`, `winter`, `summer` and `spring`.

One of the key concepts of Scientific Computing is approximation of "complicated" functions by "simpler" ones.

The simplest functions are usually **polynomials:**

- Constants (degree zero)
- Linear (degree one), e.g.,
- Quadratic, etc, e.g.,

If we know a function, and its derivatives, we can approximate it with a polynomial as follows.

Taylor Polynomials

Given a function, f, its **Taylor Polynomial, p_n, of degree n at $x = a$** has the property that

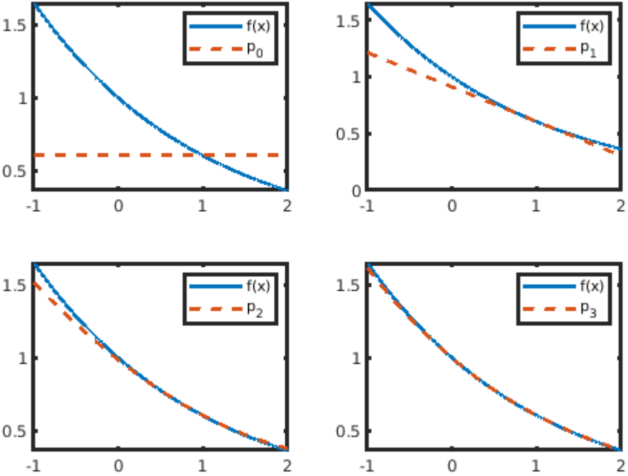
$$p_n(a) = f(a),\ p_n'(a) = f'(a),\ p_n''(a) = f''(a), \ldots, p_n^{(n)}(a) = f^{(n)}(a),$$

where $f^{(k)}$ denotes $\frac{d^k}{dx^k} f$. Its formula is

$$p_n(x) := f(a) + f'(a)(x-a) + \frac{1}{2} f''(a)(x-a)^2 + \frac{1}{3!} f'''(a)(x-a)^3 + \cdots + \frac{1}{n!} f^{(n)}(a)(x-a)^n.$$

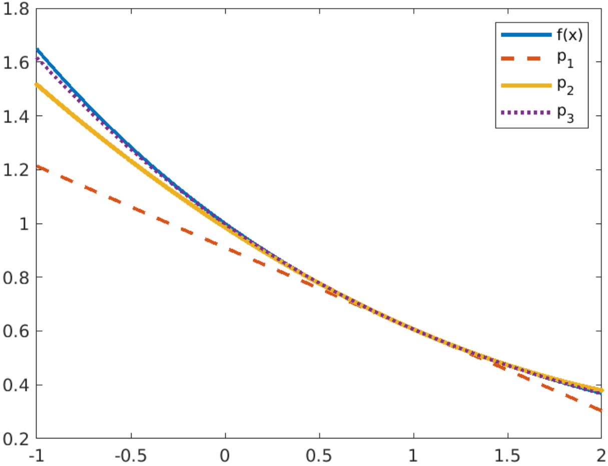
TaylorPoly.m

```
f = @(x)exp(-x/2);
df = @(x)-(1/2)*exp(-x/2);
d2f = @(x)(1/4)*exp(-x/2);
d3f = @(x)-(1/8)*exp(-x/2);

a = 1.0;
p0 = @(x)(f(a) + x*0);
p1 = @(x)(p0(x) + (x-a).*df(a));
p2 = @(x)(p1(x) + (x-a).^2.*d2f(a)/2);
p3 = @(x)(p2(x) + (x-a).^3.*d3f(a)/6);

X = linspace(-1,2,1001); % Lots of points for plotting;

figure(1); subplot(2,2,1);
plot(X, f(X), X, p0(X), '--'); legend('f(x)', 'p_0')

subplot(2,2,2);
plot(X, f(X), X, p1(X), '--'); legend('f(x)', 'p_1');

subplot(2,2,3);
plot(X, f(X), X, p2(X), '--'); legend('f(x)', 'p_2')
```



f(x)
p_0
1.5
1
0.5
-1
0
1
2
f(x)
p_1
1.5
1
0.5
0
-1
0
1
2
f(x)
p_2
1.5
1
0.5
-1
0
1
2
f(x)
p_3
1.5
1
0.5
-1
0
1
2

```
plot(X, f(X), X, p1(X), '--', X, p2(X), '.-', X, ...
  p3(X), ':', 'LineWidth', 2);
legend('f(x)', 'p_1', 'p_2', 'p_3')
```



There are many other ways of approximating a function (or data set) using polynomials, usually at multiple points.

This is called **interpolation** which comes in two versions:

Data interpolation: given a set of $n + 1$ points in 2D, find a polynomial of degree n that goes through them.

Function interpolation: given function find a polynomial that agrees with it at n points.

Questions: Why do this?

- To evaluate the polynomial at multiple other points;
- Estimate derivatives;
- Estimate integral;
- ...

Question: How can we do this? **Answer:** `polyfit()`

In the following examples, we will construct the polynomial interpolant using the function `polyfit()`.

Syntax: `p = polyfit(x, y, n);`
where `x` and `y` are vectors of the same length, and n is a integer.

If `n = length(x)-1`, then it should interpolate the data. For smaller n, a least-squares fit is used (more of that later).

`p` is a vector with $n+1$ entries, where $p(i)$ is the coefficient of x^{n+1-i} in the polynomial. That is, it represents the polynomial

$$p(1)x^n + p(2)x^{n-1} + \cdots + p(n)x + p(n+1).$$

```
>> x=1:3
x =
     1     2     3
>> y = 1-3*x-2*x.^2
y =
     6    17    34
>> p = polyfit(x,y,2)
p =
   -2.0000   -3.0000    1.0000
```

Polynomial Data Interpolation

Given the $n+1$ points (x_0, y_0), (x_1, y_1), ..., (x_n, y_n), we want to find the polynomial p_n such that

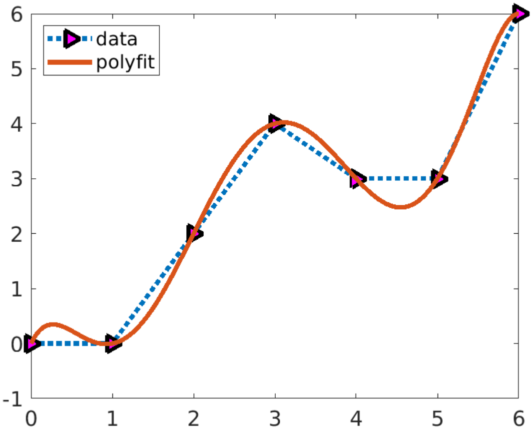
$$p_n(x_0) = y_0, p_n(x_1) = y_1, \ \ldots, \ p_n(x_n) = y_n.$$

PolyDataInterp.m

```
x = 0:6;
y = [0, 0, 2, 4, 3, 3, 6];
p = polyfit(x, y, length(x)-1);

X = linspace(x(1), x(end), 1001); % Points for plotting;
Y = polyval(p, X);
plot(x, y, ':>', X, Y, '-', ...
    'LineWidth',3, 'MarkerSize', 10,...
    'MarkerFaceColor', 'magenta', 'MarkerEdgeColor', 'k');
legend('data', 'polyfit', 'location', 'northwest')
set(gca, 'FontSize', 14)
```

PolyDataInterp.m

```
x = 0:6;
y = [0, 0, 2, 4, 3, 3, 6];
p = polyfit(x, y, length(x)-1);
```



Polynomial Function Interpolation

In practice, this works very similarly to data interpolation: given a set of points x_0, x_1, ..., x_n, and a function f, we compute the interpolant to $(x_0, f(x_0))$, $(x_1, f(x_0))$, ..., $(x_n, f(x_n))$.

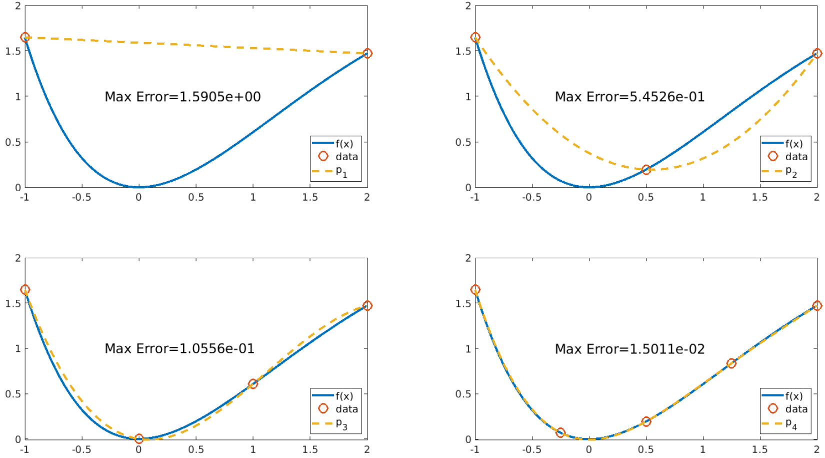
The main differences are

- We are free to choose any points in the function's domain;
- We can vary the number of points easily;
- We can estimate errors: $\|f(x) - p_n(x)\|$.

In this example, we'll construct the polynomial interpolant to $f(x) = x^2 e^{-x/2}$, in the interval $[-1, 2]$.

PolyFunctionInterpolation.m

```
f = @(x)(x.^2).*exp(-x/2);
for n=1:4
   xp = linspace(-1,2,n+1);
   p = polyfit(xp, f(xp), n);

   X = linspace(-1,2, 1001); % Points for plotting;
   Y = polyval(p, X);
   Error = norm(f(X)-Y,'inf');
   fprintf('n=%2d, Error=%10.5e\n', n, Error)
   subplot(2,2,n);
   plot(X, f(X), xp, f(xp), 'o', X, Y, '--', ...
      'LineWidth',3, 'MarkerSize', 12);
   leg_str = sprintf('p_{%d}', n);
   legend('f(x)', 'data', leg_str, 'FontSize', 14, ...
      'location', 'southeast')
   Error_str = sprintf('Max Error=%5.4e', Error);
   text(-0.3,1, Error_str, 'FontSize', 18)
   set(gca, 'FontSize', 14)
end
```


Max Error=1.5905e+00
Max Error=5.4526e-01
Max Error=1.0556e-01
Max Error=1.5011e-02
f(x)
data
p_1
p_2
p_3
p_4

In all the previous examples,

- We fitted polynomials of degree n to $n+1$ points, and
- There was no noise in the data.

In many applications, we wish to use low-order polynomials,

- because there are more stable;
- the underlying data is noisy.

Applications:

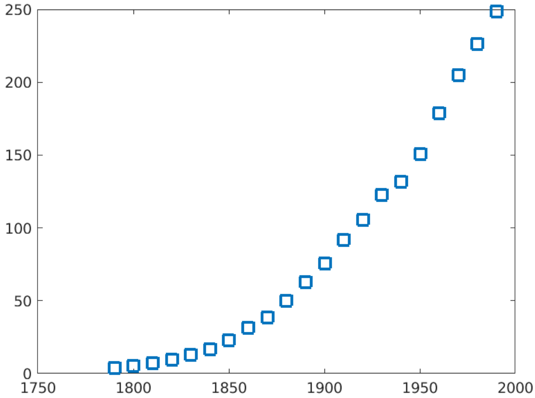
1. Estimating values at non-observed points;
2. Estimating growth rates.

Example: US Census Data

1. Construct and plot linear, quadratic and cubic fits to the US census data in `census.mat`.
2. For each, estimate the "root mean squared error" ("l^2", or "Euclidian"norm to its friends.
3. For each, what is the estimated rate of growth in 1990?
4. For each, what is the estimated population in 1985 (not a census year)?
5. For each, what is the estimated population in 2000, 2010, and 2020, and how accurate is that?

8: Least Squares

```
load census;
plot(cdate, pop, 's', 'MarkerSize', 10, 'LineWidth', 2);
```



8: Least Squares

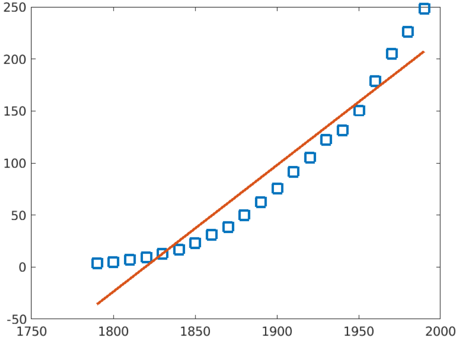
We can construct the linear fit as follows:

USCensusLeastSquares.m

```
load census;
p1 = polyfit(cdate, pop, 1); %% NOTE: degree=1
t = linspace(1790, 1990,1001);
plot(cdate, pop, 's', t, polyval(p1,t),...
   'LineWidth', 3, 'MarkerSize',10);
Diff1 = norm(pop - polyval(p1,cdate));
dp1 = polyder(p1);
fprintf('p=1, Error=%5.2f, Growth (1990)=%5.2f\n', ...
   Diff1, polyval(dp1, 1990))

Extrap = polyval(p1, 2000:10:2020);
Actual = [281.4, 308.7, 331.5];

fprintf('2000: pop estimate (actual) %.1f (%.1f)\n', ...
   Extrap(1), Actual(1));
fprintf('2010: pop estimate (actual) %.1f (%.1f)\n', ...
   Extrap(2), Actual(2));
fprintf('2020: pop estimate (actual) %.1f (%.1f)\n', ...
   Extrap(3), Actual(3));
```

Output:

```
p=1, Error=98.78, Growth Estimate (1990)= 1.22
2000: population estimate (actual) 219.5 (281.4)
2010: population estimate (actual) 231.6 (308.7)
2020: population estimate (actual) 243.8 (331.5)
```

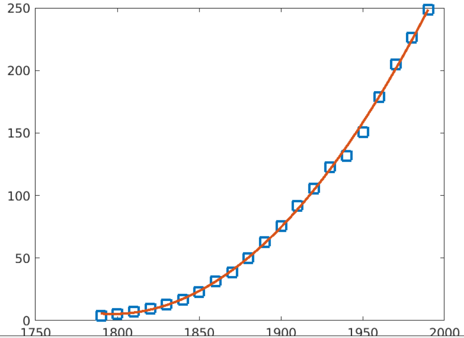


8: Least Squares

You can change the polyfit line to get a higher-order estimate.

The result for a quadratic would be: Output:

```
p=2, Error=12.61, Growth Estimate (1990)= 2.52
2000: population estimate (actual) 274.6 (281.4)
2010: population estimate (actual) 301.8 (308.7)
2020: population estimate (actual) 330.3 (331.5)
```



8: Least Squares

Experiment with higher-order interpolation. Convince yourself that the quadratic fit is most appropriate.

Suppose we want to find the polynomial $p_2 = a_2x^2 + a_1x + a_0$, that fits some data. If it is to fit the point (x_i, y_i), that means

$$a_2x_i^2 + a_1x_i + a_0 = y_i.$$

With 3 unknowns we need 3 equations. So if we have three points is there is an exact solution. The equations would be

$$\begin{aligned} a_2x_1^2 + a_1x_1 + a_0 &= y_1 \\ a_2x_2^2 + a_1x_2 + a_0 &= y_2 \\ a_2x_2^2 + a_1x_3 + a_0 &= y_3. \end{aligned}$$

We could write this as a matrix-vector equation: $Xa = y$, i.e.,

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix} \begin{bmatrix} a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}$$

But in these problems we have many more equations than unknowns:

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \\ \vdots & \vdots & \vdots \\ x_n^2 & x_n & 1 \end{bmatrix} \begin{bmatrix} a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

Now there is no (exact) solution to this problem: there is no vector a for which $Xa = y$.

But we can find a solution that is "better" than the rest. For that we need some way to understand "norms". (See next slide).

[Stuff about norms, hand-written in class].

Recall again the problem is

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

where $n > 3$. There is no solution to $Xa = y$. That is, $\|Xa - y\| > 0$ for all a.

The “least squares” solution is the one for which, the residual, $Xa - y$, is as small as possible.

In a linear algebra course, we would prove such a vector exists, and explain how to find it. But here we will just compute it:

```
X = [x.^0, x, x.^2];
a = X\y
```

Try this for the US Census data.

You should take `x=cdate`, `y=pop`. Compare the resulting a with the value of p computed by `polyfit()`.