Interpolation and Extrapolation

Why this is useful, and what we want out of it

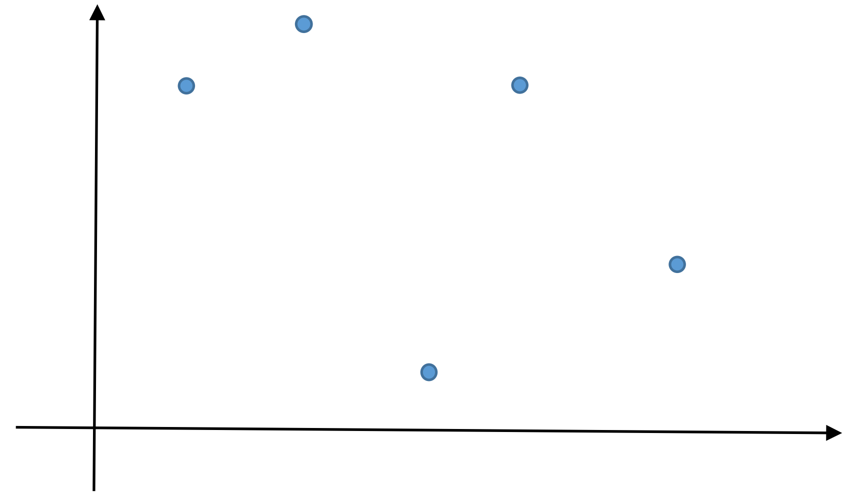
Van der Monde interpolation

Lagrange Interpolation

Splines

Often we are given a set of data, and we would like to find
an exact curve that goes through all the data points.

Why would we need this?

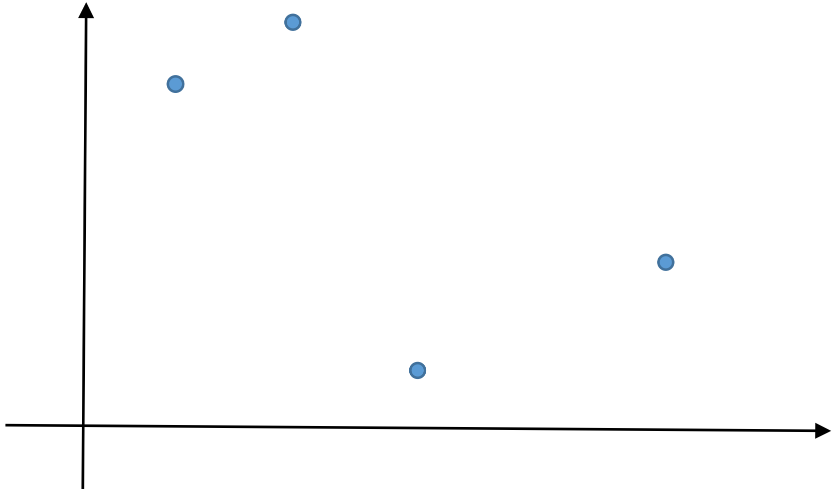


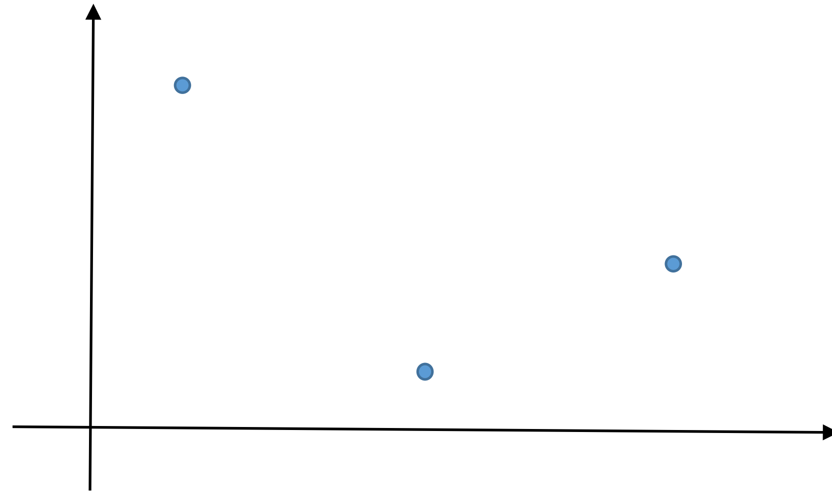
1. We often need to know, or at least guess, the value of Data at points other than where it is given

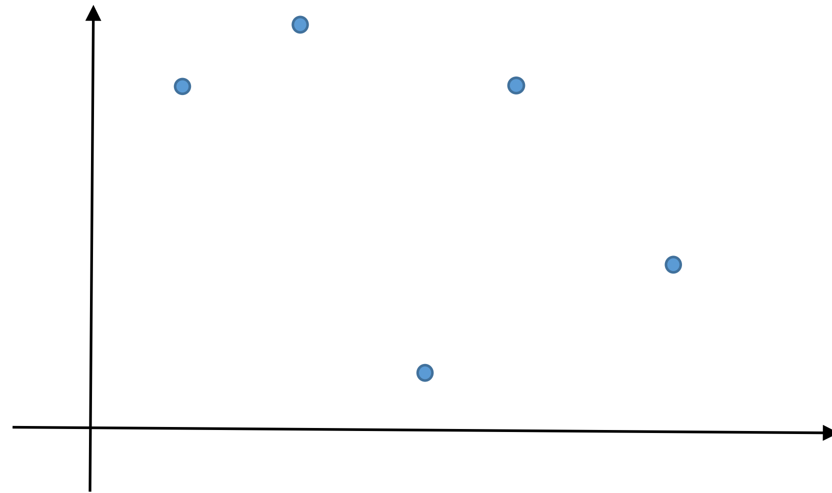
2. The data represents a curve that we would like to *integrate* or *differentiate*

The simplest type of curve that we can use for this is a *polynomial.*

Given N+1 points we can fit a unique Nth-order polynomial.





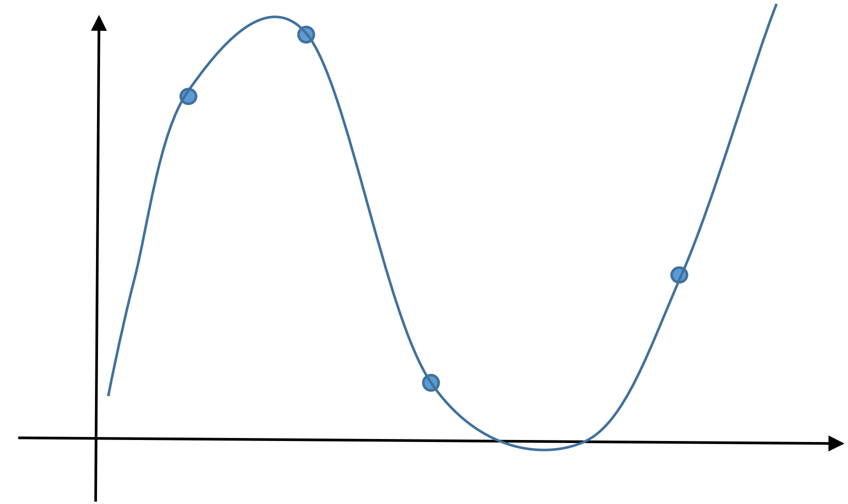


The x-points are usually called *nodes*, the y-points *values*, and the x-y pair is often called a *knot.*

If we fit *inside* the range of given data, this is known as *interpolation.*
If we then go *outside* the range of given data, this known as *extrapolation.*

Vandermonde interpolation

This is the most basic thing that we can do:
Substitute the values of the function into a polynomial and solve the resulting (big) linear system.



Consider the set of n+1 knots

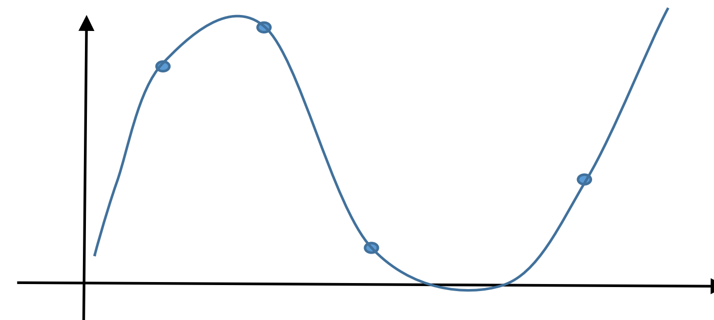
$$(x_0, y_0), \quad (x_1, y_1), \quad (x_2, y_2), \quad \ldots \quad (x_n, y_n)$$

We would like to fit an nth-order polynomial through all these points:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Substituting $p(x_0) = y_0$ we obtain

$$y_0 = a_0 + a_1 x_0 + a_2 x_0^2 + \cdots + a_n x_0^n$$



$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Vandermode interpolation:

1. Formulate the Vandermonde matrix

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & & & \ddots & \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix}$$

2. Invert to find the unknown coefficients

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & & & \ddots & \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix}^{-1} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

3. Substitute these back into the polynomial equation

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Advantages

- Conceptually simple

Problems:

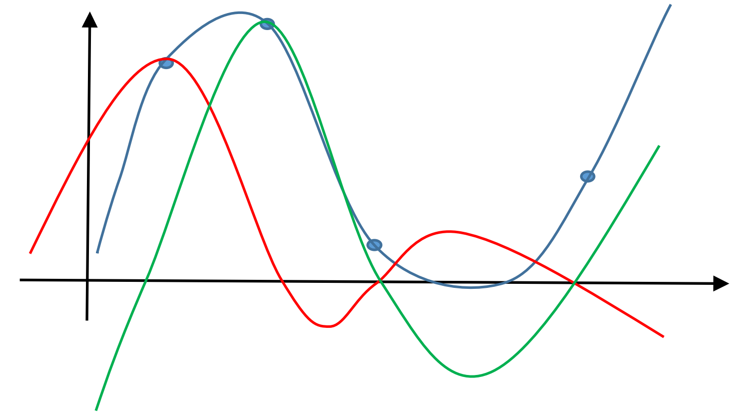
- The matrix is *dense* and is slow to compute*
- The matrix can end up being *poorly conditioned*
- There is a better, equivalent way (with more applicability)

*not really a problem nowadays

Lagrange interpolation

The main idea of Lagrange interpolation is that we expand in terms of a sum of polynomials *each of which matches the data at exactly one point*

The polynomial for the jth node must satisfy:



1. $\ell_j(x) = 1$ at $x = x_j$
2. $\ell_j(x) = 0$ at $x_i = x_j \ , i \neq j$

If we have these, then an interpolating function is

$$L(x) = \sum_{j}^{n} \ell_j(x) y_j$$

Lagrange (or more likely Euler) realised a good polynomial for this is

$$\ell_j(x) = \frac{\prod_{i \neq j}(x - x_i)}{\prod_{i \neq j}(x_j - x_i)} = \frac{(x - x_0)(x - x_1) \cdots (x - x_{j-1})\ (x - x_{j+1}) \cdots (x - x_n)}{(x_j - x_0)(x_j - x_1) \cdots (x_j - x_{j-1})\ (x_j - x_{j+1}) \cdots (x_j - x_n)}$$

Advantages of Lagrange interpolation:

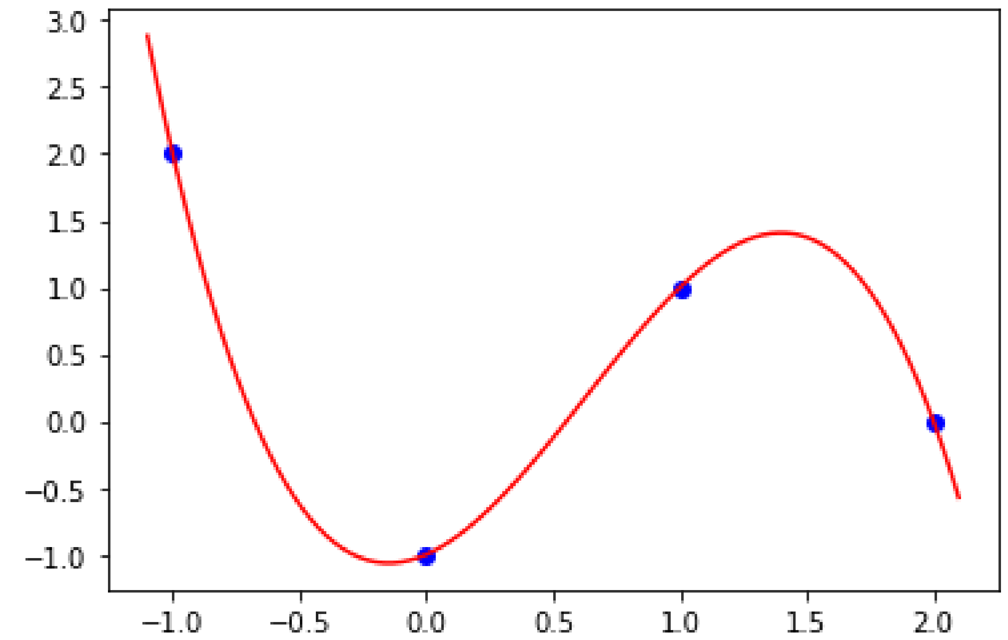
1. Quick and easy

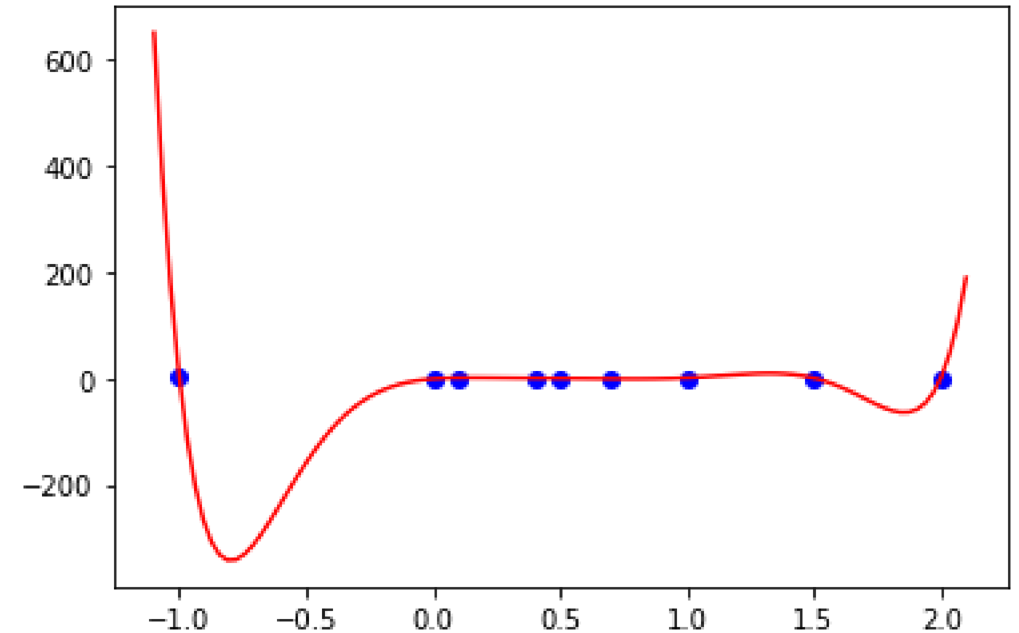
2. No matrix inversion (yay!)

3. Everything is well-behaved

Disadvantages:

High order interpolation is unstable

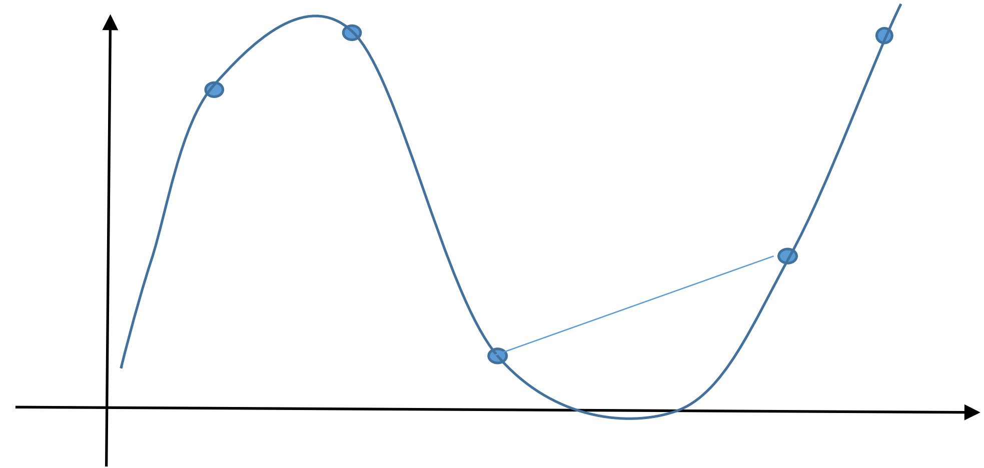




Low order polynomial interpolation is preferable

Spline interpolation

Complicated, but powerful



Start with a *linear interpolation* between two points:

$$y = A(x)y_j + B(x)y_{j+1}$$

How can we fix this to make the first derivative *smooth,*
and *at the same time* make the second derivative *continuous*?

$$A \equiv \frac{x_{j+1} - x}{x_{j+1} - x_j} \qquad B \equiv 1 - A = \frac{x - x_j}{x_{j+1} - x_j}$$

The solution: add a cubic polynomial*

$$y = A(x)y_j + B(x)y_{j+1} + C(x)q_j + D(x)q_{j+1}$$

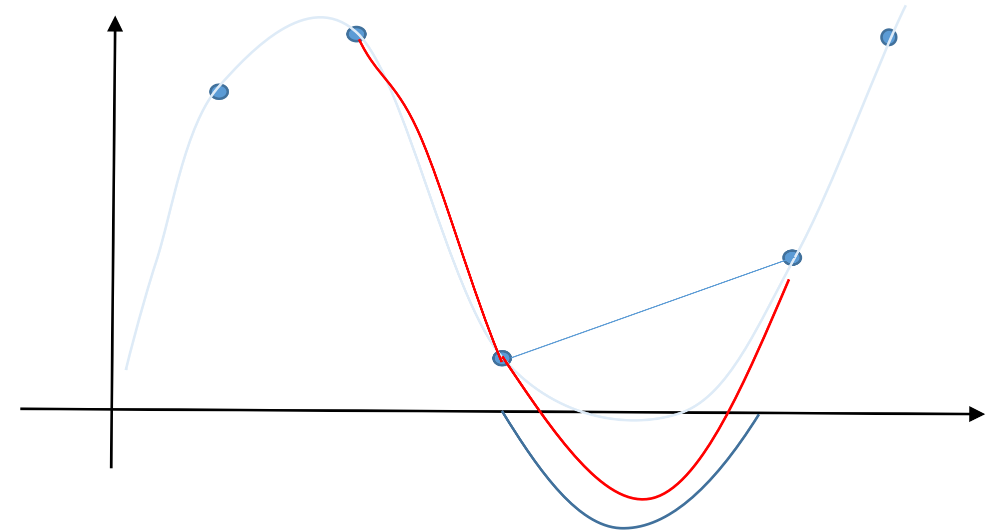
*other types of splines are possible, but cubic splines are the most common

We focus on a single segment.

$$y = A(x)y_j + B(x)y_{j+1} + C(x)q_j + D(x)q_{j+1}$$

We specify that the 2nd derivative of the interpolating polynomial must vary *linearly* over the segment, as well as having zero *values at the end-points.* This implies

$$C \equiv \frac{1}{6}(A^3 - A)(x_{j+1} - x_j)^2 \qquad D \equiv \frac{1}{6}(B^3 - B)(x_{j+1} - x_j)^2$$



We also want the first derivative in one segment to be equal to the first derivative in the next. The expression for the 1st derivative is

$$\frac{dy}{dx} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{3A^2 - 1}{6}(x_{j+1} - x_j)y_j'' + \frac{3B^2 - 1}{6}(x_{j+1} - x_j)y_{j+1}''$$

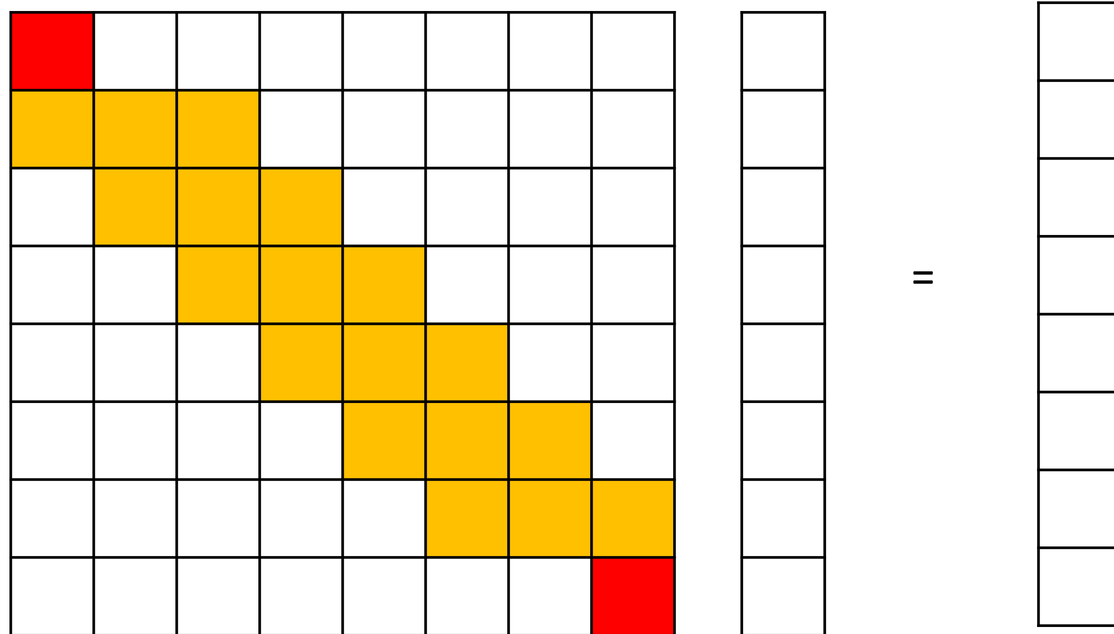
And this implies the relation

$$\frac{x_j - x_{j-1}}{6}y_{j-1}'' + \frac{x_{j+1} - x_{j-1}}{3}y_j'' + \frac{x_{j+1} - x_j}{6}y_{j+1}'' = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}}$$

Horribly complicated!

$$\frac{x_j - x_{j-1}}{6} y''_{j-1} + \frac{x_{j+1} - x_{j-1}}{3} y''_j + \frac{x_{j+1} - x_j}{6} y''_{j+1} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}}$$

These are a set of equations, one for each interval except for the end-points,
Which we have to solve for the unknown q_j coefficients



$$A \equiv \frac{x_{j+1} - x}{x_{j+1} - x_j} \qquad B \equiv 1 - A = \frac{x - x_j}{x_{j+1} - x_j}$$

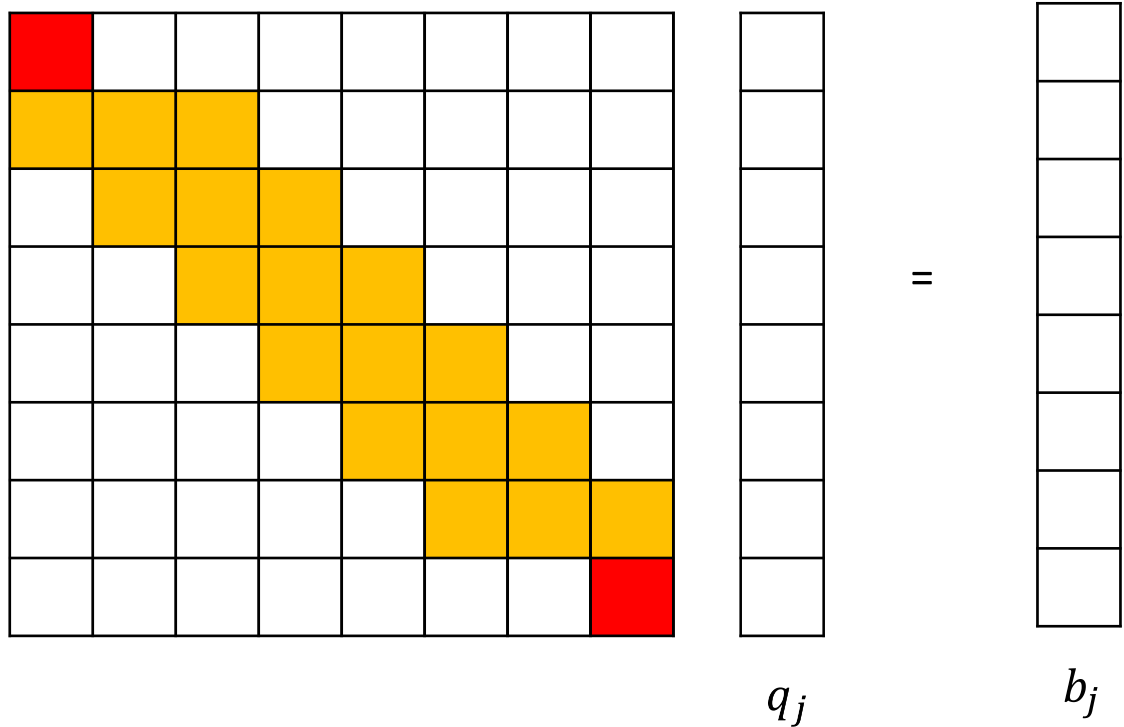
$$C \equiv \frac{1}{6}(A^3 - A)(x_{j+1} - x_j)^2$$

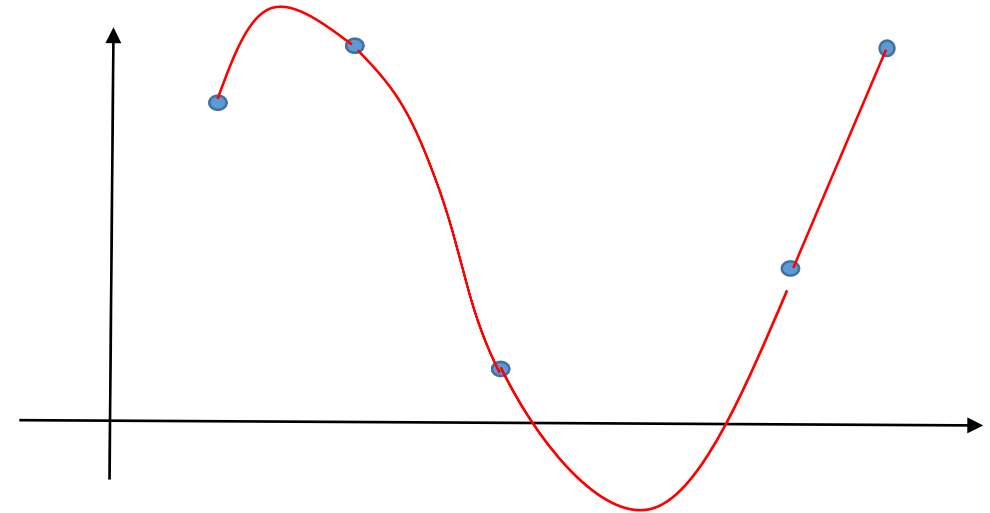
$$D \equiv \frac{1}{6}(B^3 - B)(x_{j+1} - x_j)^2$$

Once these coefficients are known, we substitute them back into

$$y = A(x) y_j + B(x) y_{j+1} + C(x) q_j + D(x) q_{j+1}$$

Note that we only have n-2 equations for n unknowns. To supply the missing information, we have to either specify the value of the *first* or *second derivative* on the edges.



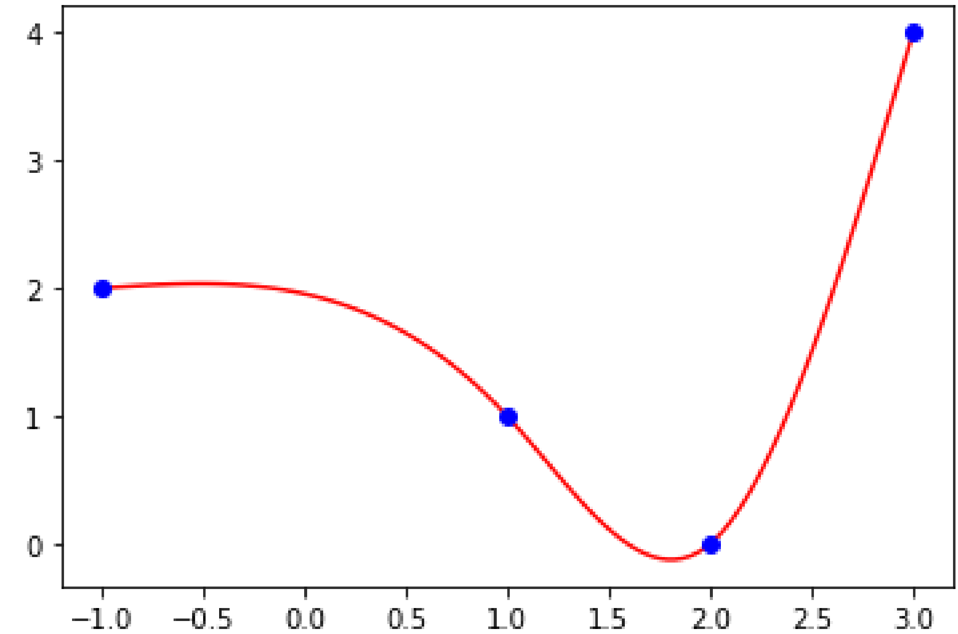


Specifying that the 2nd derivative is zero on the edge leads to so-called *natural* boundary conditions.

Advantages of cubic splines:

- The matrix system is *tridiagonal* and so is easy to solve

- You only have to solve the system *once* - when this is done, you can use the coefficients q_j to generate any point on the spline.



Available python module:

```
 7  from scipy.interpolate import CubicSpline
 8  import numpy as np
 9  import matplotlib.pyplot as plt
10
11  x = [-1, 1, 2,3]
12  y = [2, 1,0,4]
13
14  f = CubicSpline(x, y, bc_type='natural')
15  x_new = np.linspace(-1, 3, 100)
16  y_new = f(x_new)
17
18  plt.plot(x_new, y_new, 'r')
19  plt.plot(x, y, 'bo')
20
21  plt.show()
```

Other important interpolation schemes:

- Rational function interpolation – especially *Pade approximation*

  Interpolate using a fraction of polynomials, and so can deal with singularities.

- Neville's algorithm

  A way of computing the Lagrange interpolation very fast and in an organised way

- Interpolation in multiple dimensions

  Each technique has its N-dimensional analogue. Lagrange interpolation in 3D is particularly important for the Finite Element method.

- A fistful of splines

  Linear splines, quadratic splines, Hermite splines, B-splines, P-splines, non-uniform rational B-splines (NURBS) etc etc