Announcement: Final Projects

- The final Project is worth 40% of your final grade
- In the project you will write a substantial piece of code that does *one of the following:*
  - ❑ *Combines two or more of the numerical methods presented to solve a new type of problem*
  - ❑ *Explores one of the numerical methods not covered directly in the lectures or labs, or significantly extends one of the methods that we do cover*
  - ❑ *Uses the numerical methods here to solve a problem from one of your other subjects.*
- You will also give a 5 minute presentation (in Week 12) on how your code works (or is supposed to work)
- The code will be due at the end of Week 13 (i.e. at the end of Stuvac before the exam period).

Project Topics

The Project must be approved by the Subject Coordinator by the first week back after the mid-semester break

Ideas for projects:

1. Program a new numerical zero finding procedure not covered in lectures – Ridder's method, Dekker's method, Brent's method. Analyse the convergence.
2. Program one of the minimum-finding procedures not covered in Labs.
3. Implement and analyse Newton's method for complex variables
4. Implement and analyse Newton's method in 3D
5. Research and implement bracketing in 2D (and higher dimensions)
6. Program a conjugate gradient method in nD
7. Write python code to solve the Black-Scholes equation
8. Investigate and program a higher-order Newton's method
9. Implement Romberg integration
10. Program your own Simplex method. Give it an extra action that improves the search in some way.
11. Write code to solve the surface flux integrals from Vector Calculus
12. Write your own Brent's method for minimisation.
13. Write code that minimises integrals. Demonstrate this using a physical example.

Interpolation and Extrapolation

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

Van der Monde interpolation $\longleftarrow$

Lagrange Interpolation $\longleftarrow$

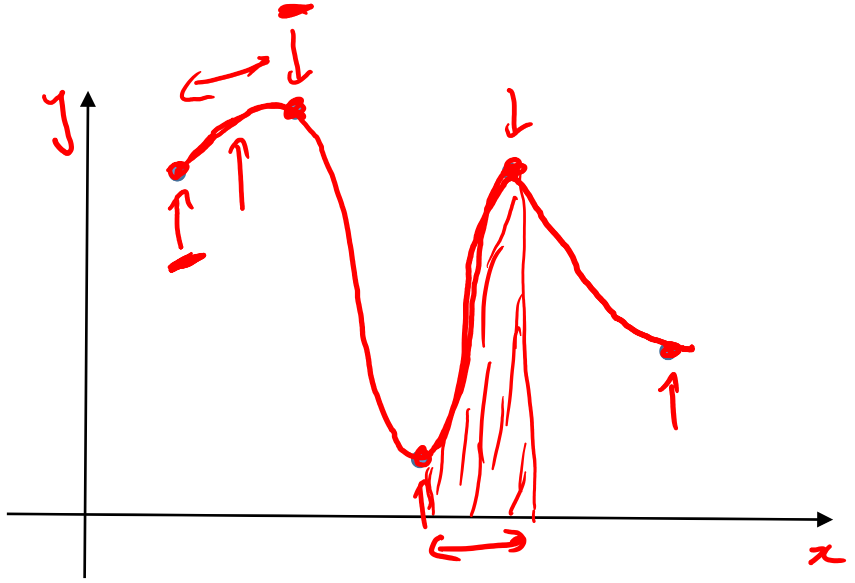
Splines $\longleftarrow$

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

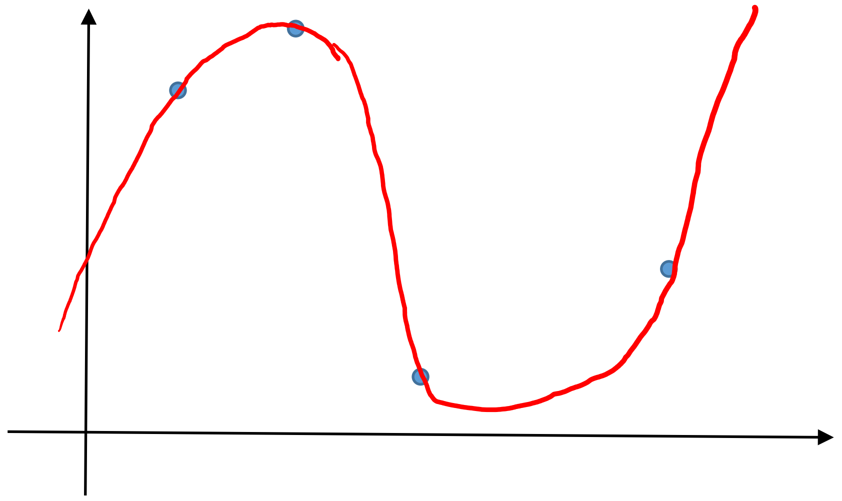
2. The data represents a curve that we would like to *integrate* or *differentiate*

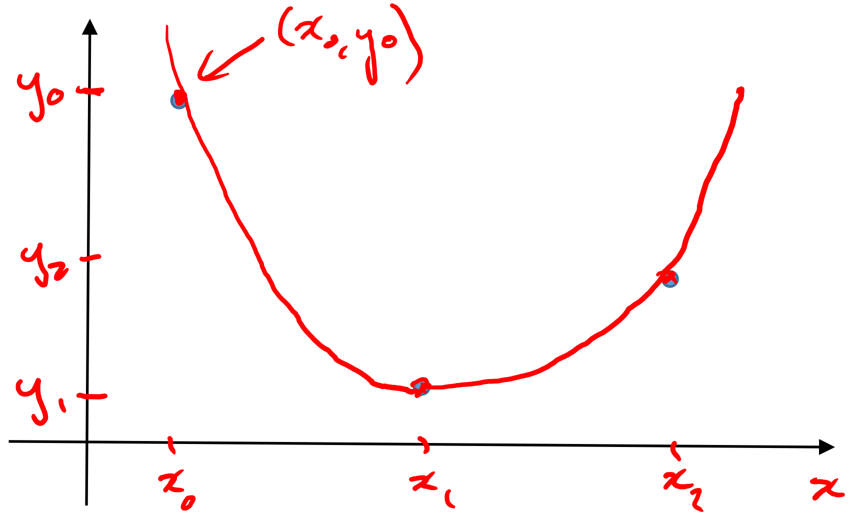


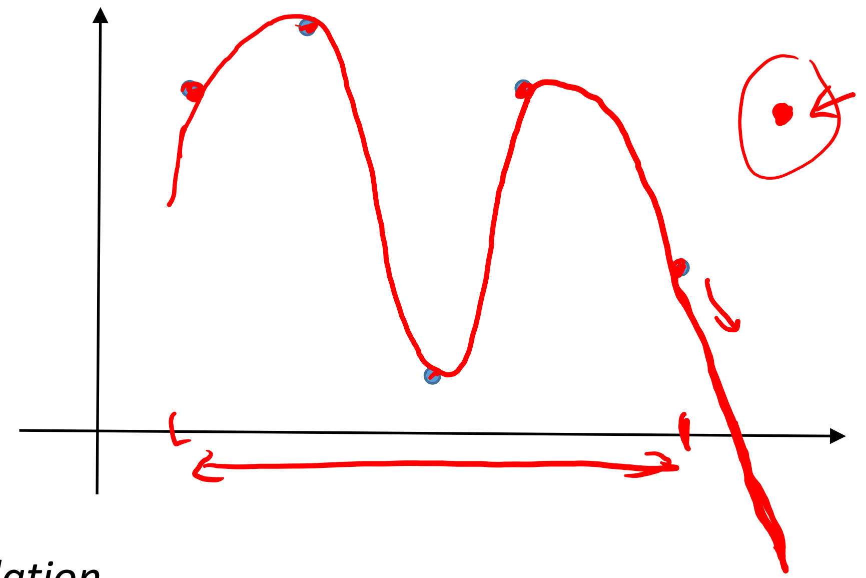
$$y(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

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.





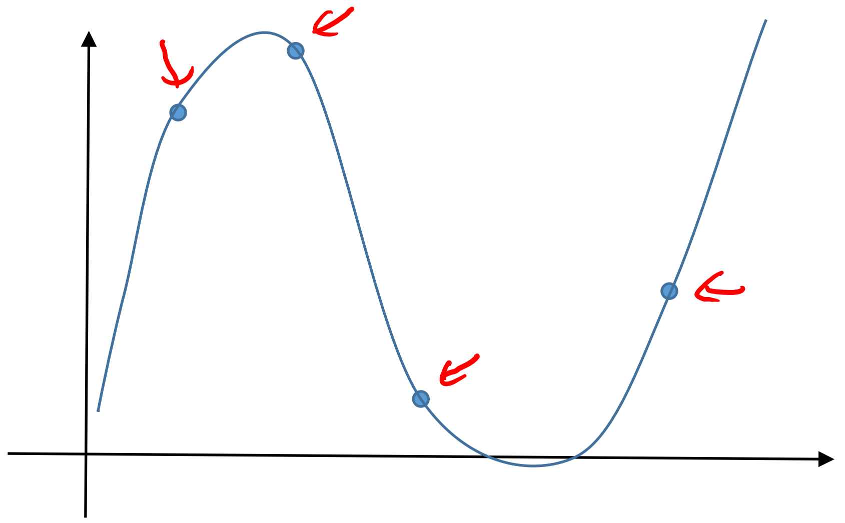


The x-points are usually called *nodes*, the y-points *values*, and the x-y pair is often called a *knot*.

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

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

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

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

want to find a_n

Substituting $p(x_0) = y_0$ we obtain

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

sub $p(x_1) = y_1$:

$$y_1 = a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_n x_1^n \quad (x_1, y_1)$$

n+1 equations

$$y_2 = a_0 + a_1 x_2 + a_2 x_2^2 + \cdots + a_n x_2^n$$

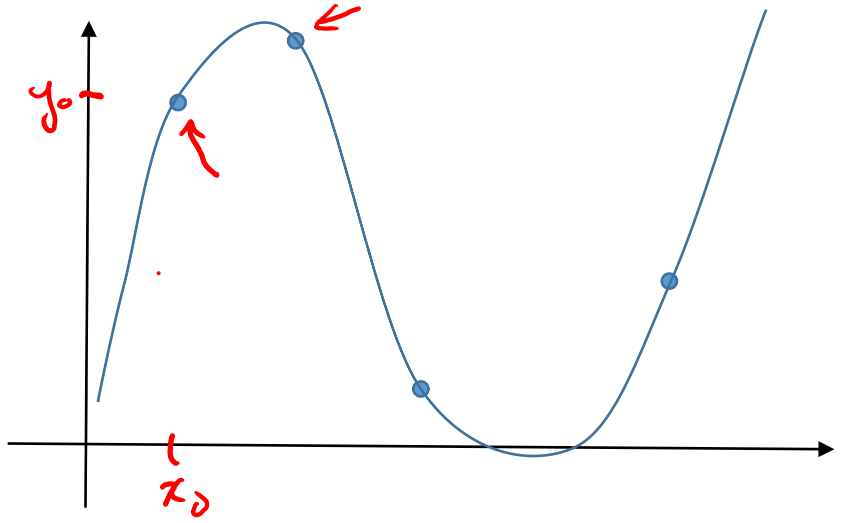
$$\vdots$$

$$y_n = a_0 + a_1 x_n + a_2 x_n^2 + \cdots + a_n x_n^n$$

or

$$\underset{\text{known}}{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}} = \overset{\text{known}}{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & & & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix}} \underset{\text{unknown}}{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}} \Rightarrow \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \\ \\ \\ \end{bmatrix}^{-1} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

inverse



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

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

Advantages

- Conceptually simple

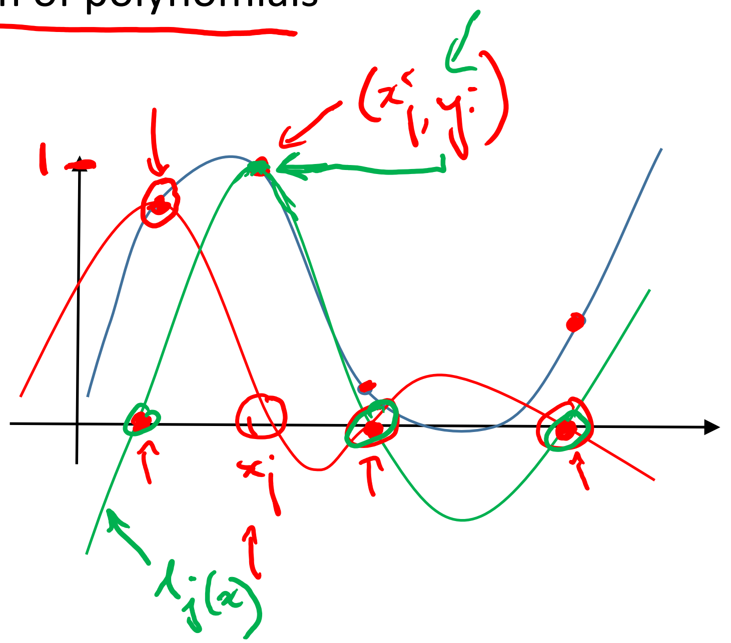
Problems:

- 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 j^{th} 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

missing

$(x - x_j)$

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

n^{th}-order polynomial

Advantages of Lagrange interpolation:

1. Quick and easy $\checkmark$

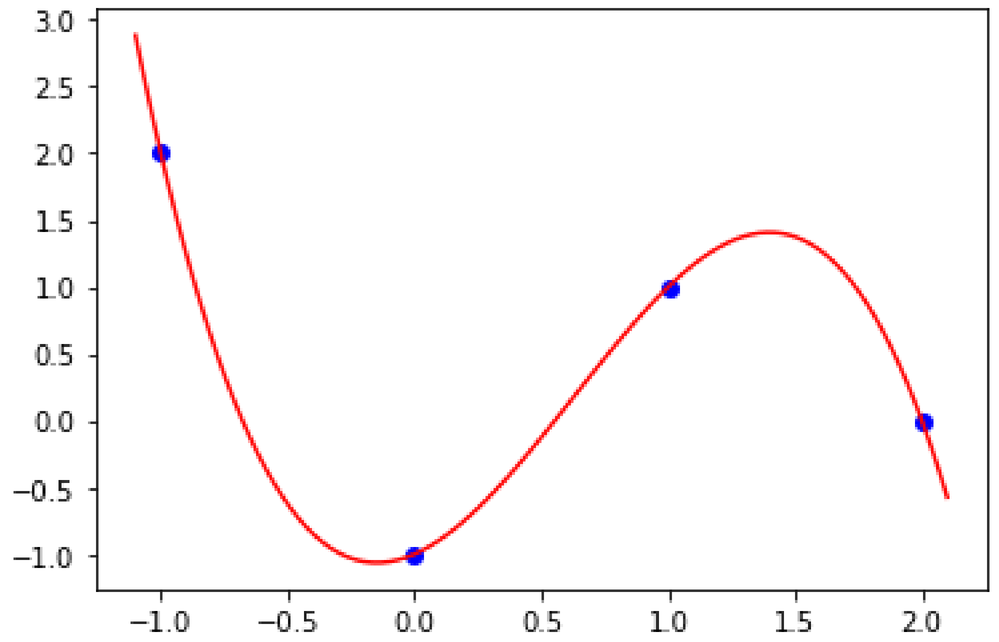
2. No matrix inversion $\underline{\text{(yay!)}}$

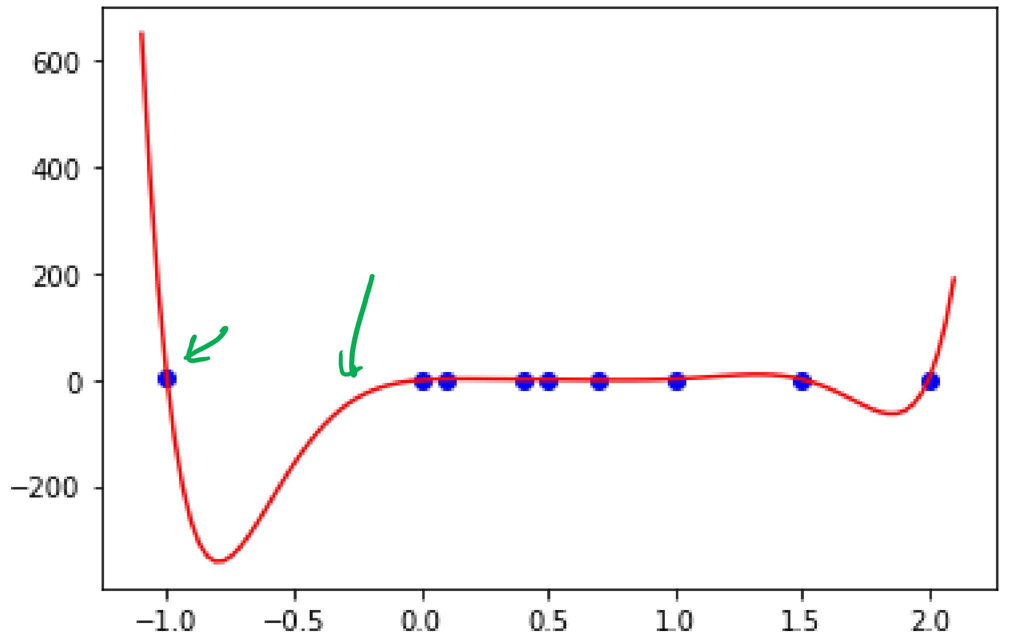
3. Everything is $\underline{\text{well-behaved}}$

Disadvantages:

<u>High order interpolation is unstable</u>

$\Rightarrow$ Low order polynomial interpolation is preferable $\checkmark$



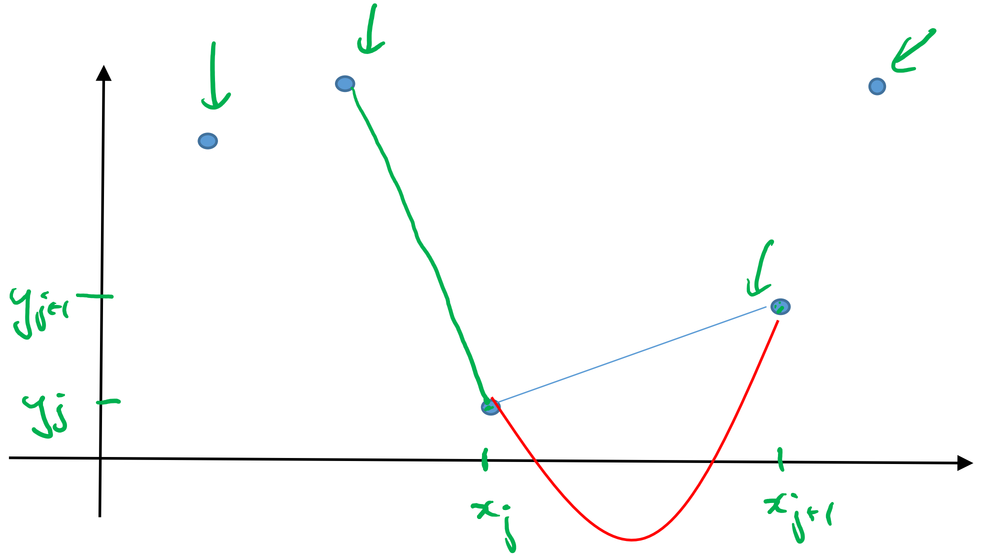


Spline interpolation

Complicated, but powerful

Start with a *linear interpolation* between two points:

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



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

$A(x)$ $B(x)$

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

The solution: add a cubic polynomial*

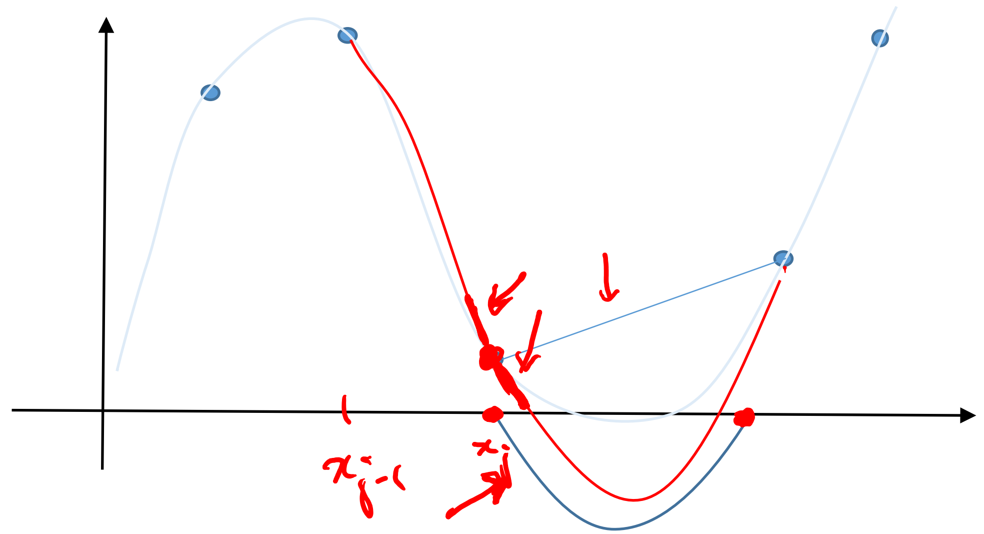
$$q_j \equiv \left.\frac{d^2y}{dx^2}\right|_{x=x_j} \equiv y_j''$$

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

*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

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

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

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

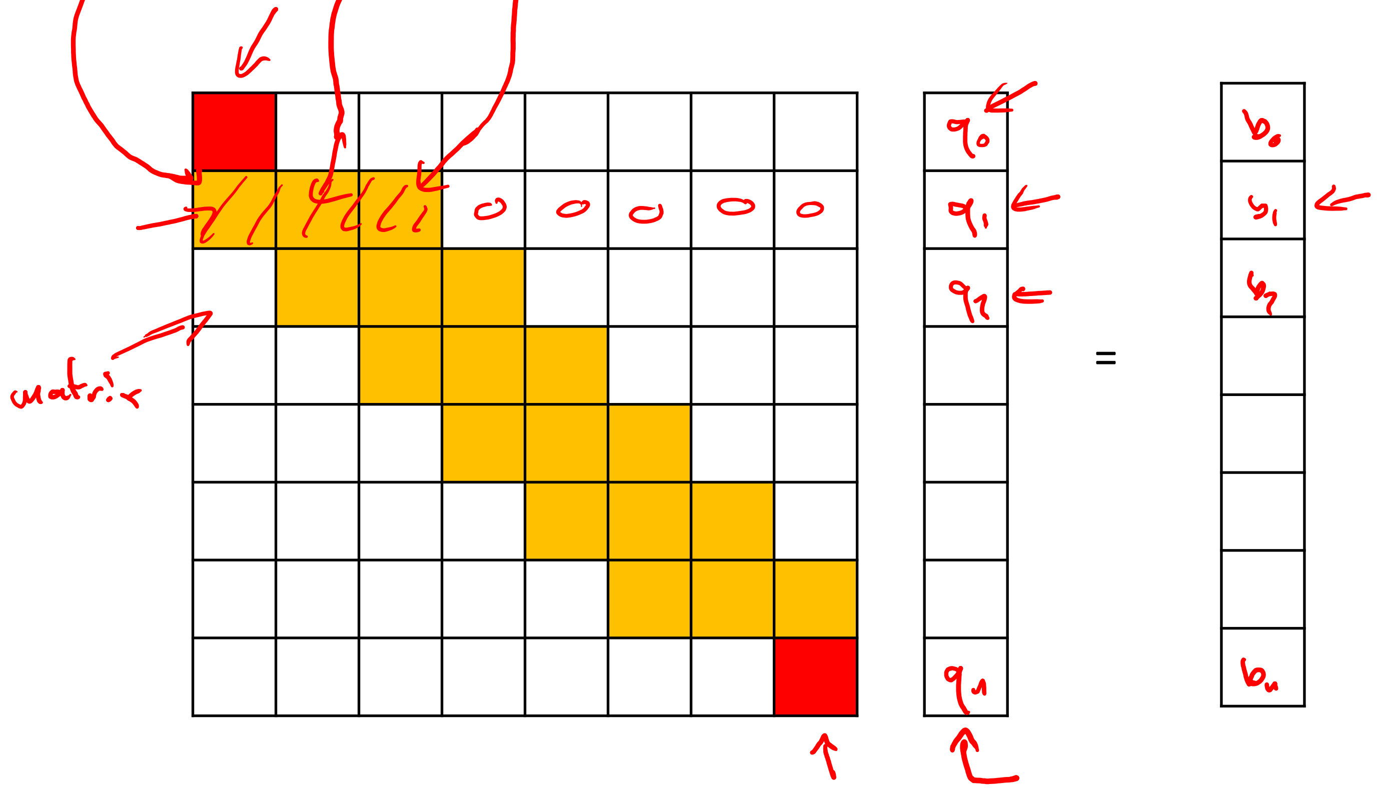
And this implies the relation

$$\frac{x_j - x_{j-1}}{6}\overset{q_{j-1}}{y_{j-1}''} + \frac{x_{j+1} - x_{j-1}}{3}\overset{q_j}{y_j''} + \frac{x_{j+1} - x_j}{6}\overset{q_{j+1}}{y_{j+1}''} = \underbrace{\frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}}}_{\text{known!}}$$

Horribly complicated!

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

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



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

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

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

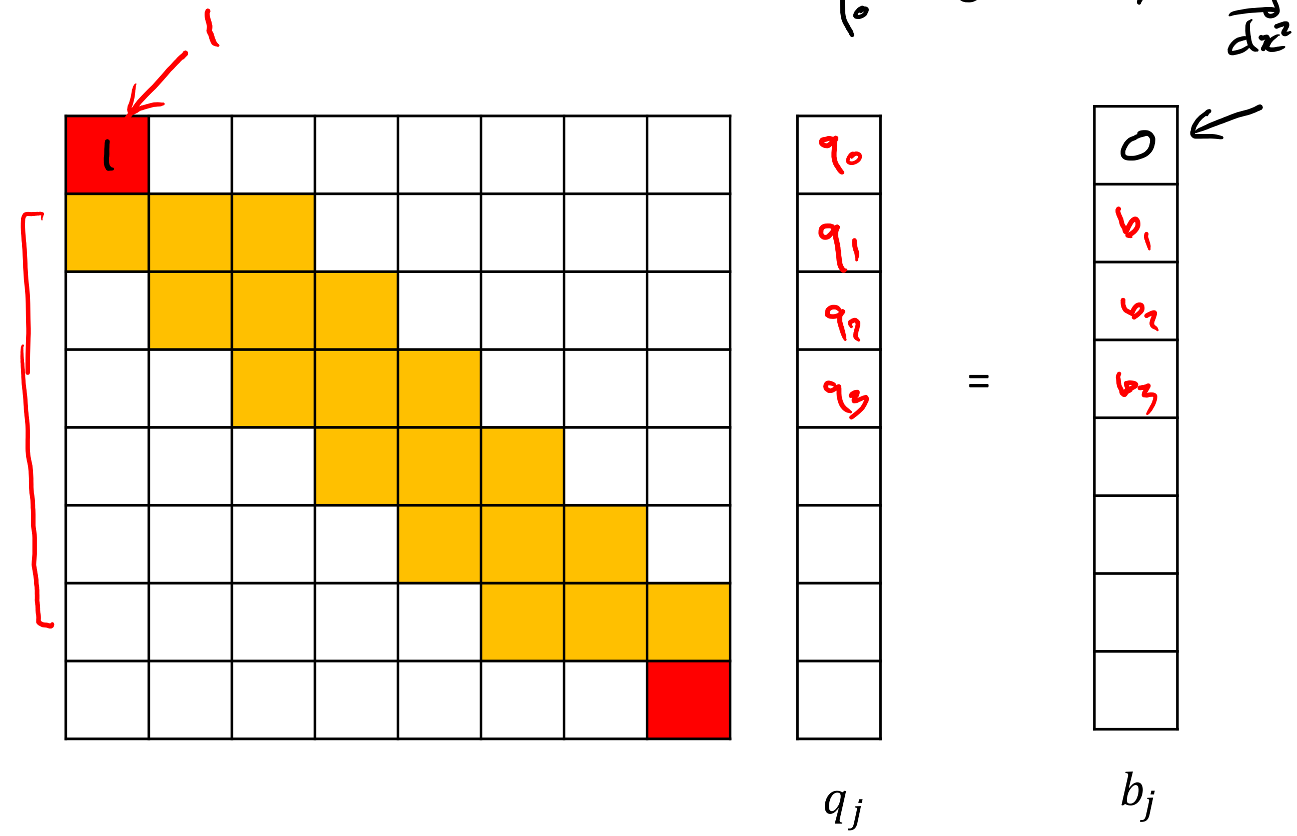
Once these coefficients are known, we substitute them back into

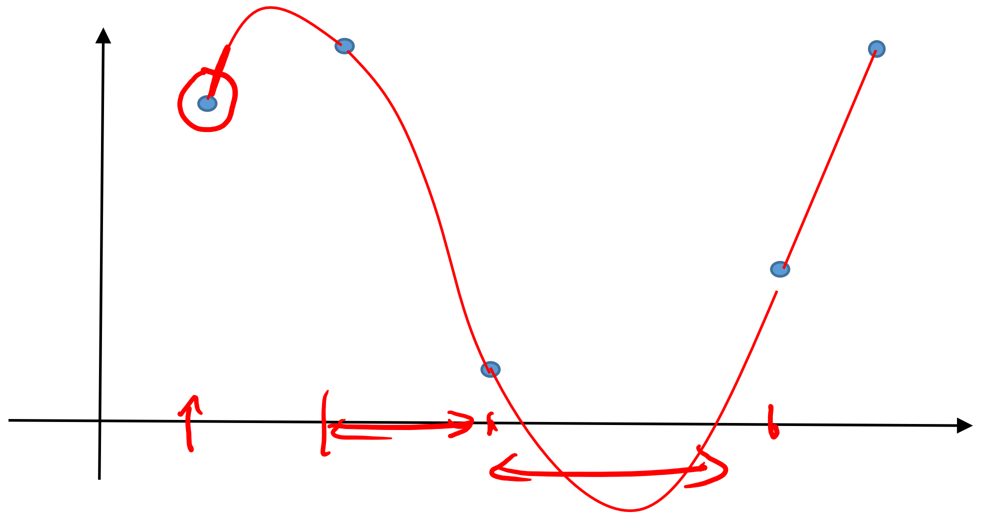
$$y = A(x)y_j + B(x)y_{j+1} + C(x)q_j + D(x)q_{j+1}$$

spline interpolating function

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.

$$q_0 = 0 \Rightarrow \frac{d^2y}{dx^2}\Big|_{x=x_0} = 0 .$$



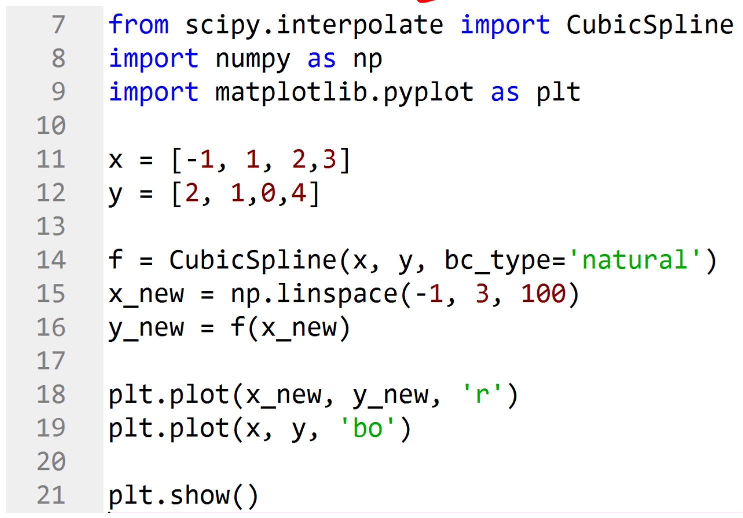


Specifying that the 2nd derivative is zero on the edge leads to so-called *natural* boundary conditions.

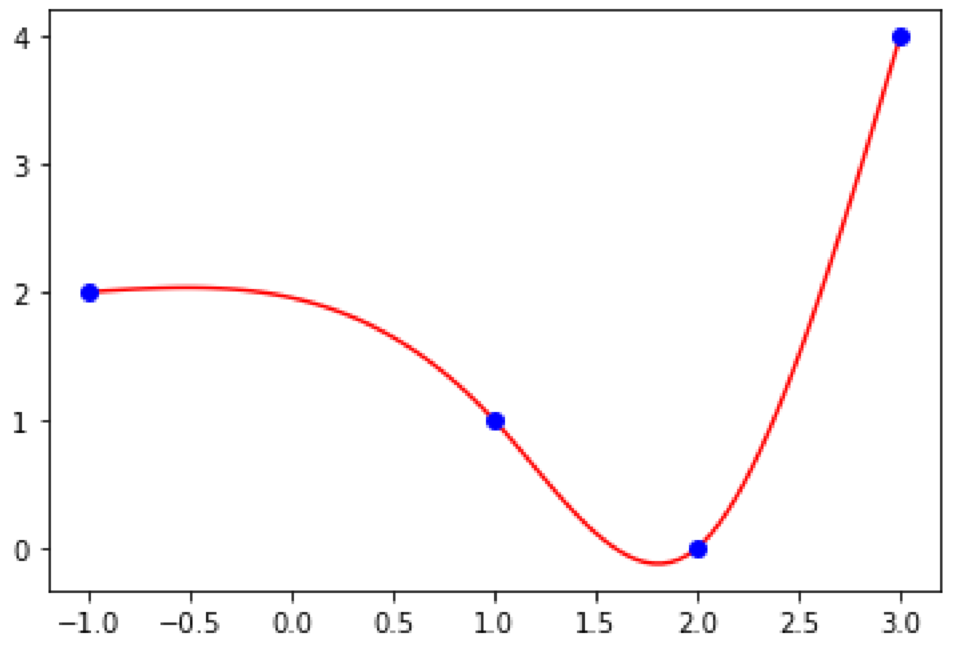
Advantages of cubic splines:

- 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.

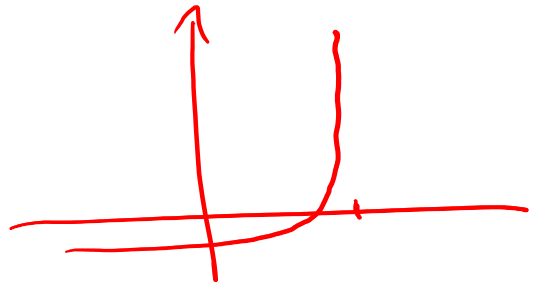
Available python module:



```python
from scipy.interpolate import CubicSpline
import numpy as np
import matplotlib.pyplot as plt

x = [-1, 1, 2,3]
y = [2, 1,0,4]

f = CubicSpline(x, y, bc_type='natural')
x_new = np.linspace(-1, 3, 100)
y_new = f(x_new)

plt.plot(x_new, y_new, 'r')
plt.plot(x, y, 'bo')

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