Numerical Methods for Astrophysics:

INTERPOLATION

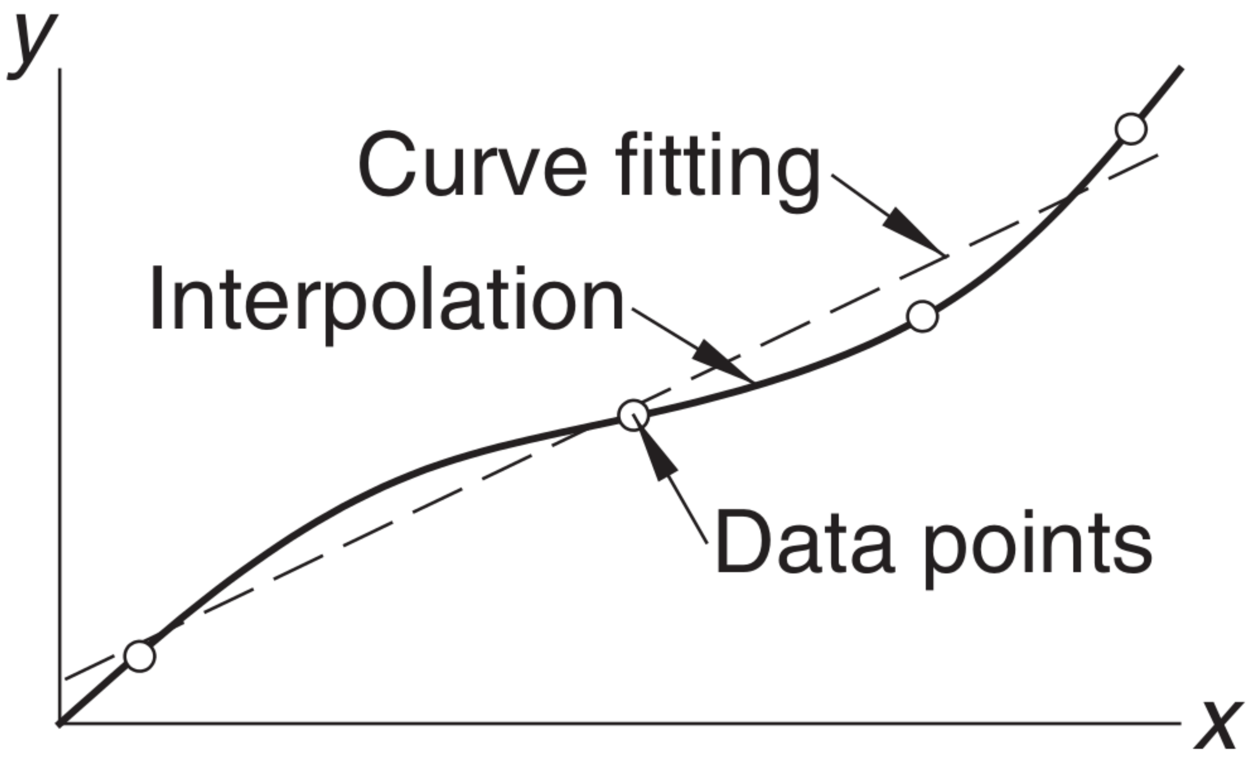
Michela Mapelli

Interpolation. Concept

Interpolation and curve fitting should not be confused.

Interpolation: curve passing through a discrete set of data points;
implicit assumption that all the data points are accurate.
Mostly done with **theoretical** sets of data points.

Curve fitting: smooth curve that approximates the data.
Thus, the curve does not necessarily hit the data points.
Mostly applied to data that contain scatter (noise), due to measurement errors.



Interpolation. Linear interpolation

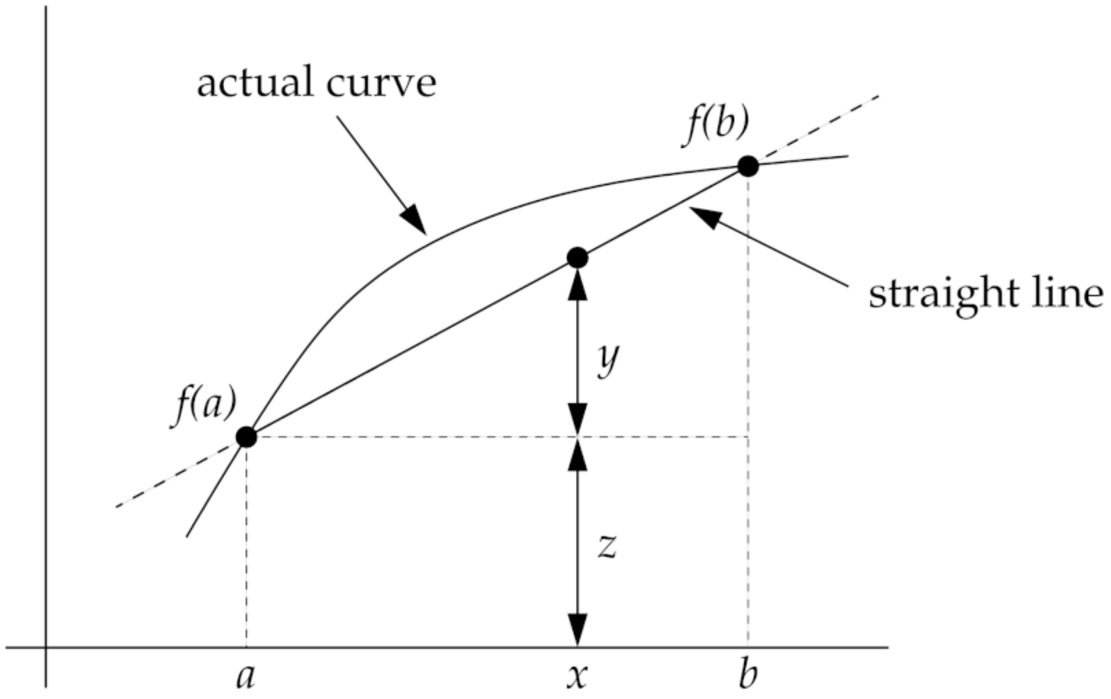
Suppose you are given the value of a function f (x) at just two points a and b and you want to know the value of the function at another point x in between.

The simplest way to proceed (and often the BEST WAY to proceed) is to assume our function follows a **straight line** from f (a) to f (b).

Slope: $m = \dfrac{f(b) - f(a)}{b - a}$

Hence:

$$f(x) \simeq y + z = \frac{f(b) - f(a)}{b - a}(x - a) + f(a)$$
$$= \frac{(b - x)\, f(a) + (x - a)\, f(b)}{b - a}$$



In most astrophysical cases, we do not have a functional form but two arrays of points

$$\begin{matrix} x_0 & x_1 & x_2 & \cdots & x_n \\ y_0 & y_1 & y_2 & \cdots & y_n \end{matrix}$$

The procedure of linear interpolation gives its best if we perform one linear interpolation every pair of data:

$$\boxed{y(x) = \frac{(x_{i+1} - x)\, y_i + (x - x_i)\, y_{i+1}}{x_{i+1} - x_i}}$$

Interpolation. Linear interpolation

Suppose you are given the value of a function f (x) at just two points a and b and you want to know the value of the function at another point x in between.

The simplest way to proceed (and often the BEST WAY to proceed) is to assume our function follows a **straight line** from f (a) to f (b).

Slope: $m = \frac{f(b) - f(a)}{b - a}$

Hence:

$$f(x) \simeq y + z = \frac{f(b) - f(a)}{b - a}(x - a) + f(a)$$
$$= \frac{(b - x)\, f(a) + (x - a)\, f(b)}{b - a}$$

$$\boxed{y(x) = \frac{(x_{i+1} - x)\, y_i + (x - x_i)\, y_{i+1}}{x_{i+1} - x_i}}$$

OFTEN THE BEST WAY TO INTERPOLATE and to EXTRAPOLATE A GRID OF DATA

In most astrophysical cases, we do not have a functional form but two arrays of points

$$\begin{array}{ccccc} x_0 & x_1 & x_2 & \dots & x_n \\ y_0 & y_1 & y_2 & \dots & y_n \end{array}$$

The procedure of linear interpolation gives its best if we perform one linear interpolation every pair of data:

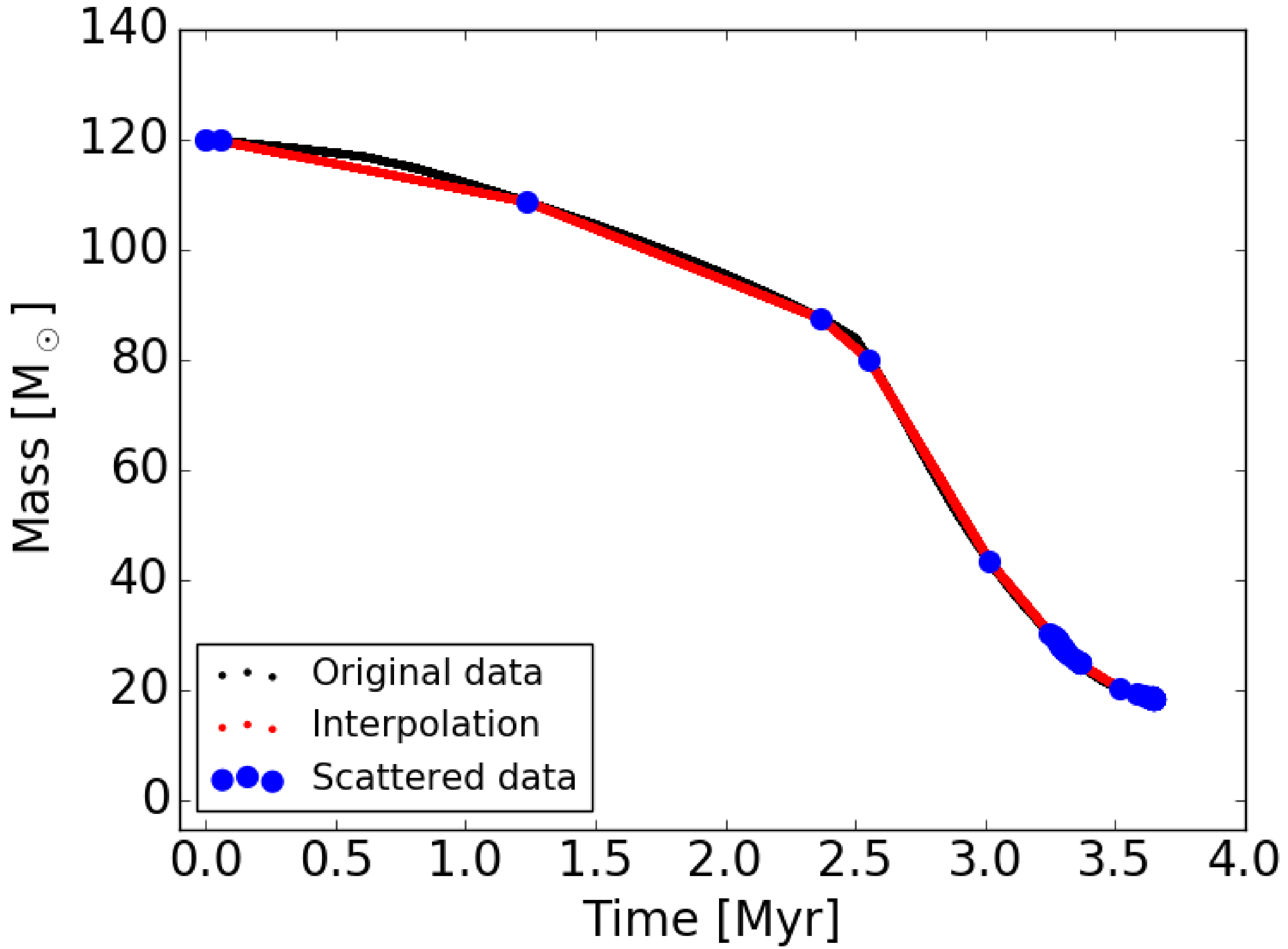
$$\boxed{y(x) = \frac{(x_{i+1} - x)\, y_i + (x - x_i)\, y_{i+1}}{x_{i+1} - x_i}}$$

Interpolation. Linear interpolation, exercise

EXERCISE:

The file 120a300.fis contains the evolution of a 120 $M_\odot$ star at solar metallicity integrated with the stellar evolution code FRANEC [Limongi and Chieffi, 2018]. Columns 0 and 7 are the evolutionary time (in years) and the mass (in $M_\odot$). The file evol_120msun_scattered.dat shows a subsample of data points with respect to file 120a300.fis, with a much coarser time grid. Columns 0 and 1 are the evolutionary time (in Myr) and the mass (in $M_\odot$). In both files the comments are preceded by #.
Using the linear interpolation, interpolate the values of the mass from file evol_120msun_scattered.dat in all the times of 120a300.fis. Then compare your interpolation with the values in the original file 120a300.fis. The plot should look like Figure 55.

Interpolation. Linear interpolation, exercise



Interpolation. Polynomial interpolation, Lagrange method

It is always possible to construct an unique polynomial of degree n that passes through n + 1 data points.

There are different formulas to build that polynomial.

LAGRANGE's formula: $P_n(x) = \sum_{i=0}^{n} y_i \, l_i(x)$

where the subscript n demotes the degree of the polynomial and

$$l_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}, \qquad i = 0, 1, ..., n$$

are the cardinal functions.

For n = 1 $\quad P_1(x) = y_0 \, l_0(x) + y_1 \, l_1(x)$ $\quad$ with

$$l_0(x) = \frac{x - x_1}{x_0 - x_1}$$

$$l_1(x) = \frac{x - x_0}{x_1 - x_0}$$

it's the straight line

For n = 2 $\quad P_2(x) = y_0 \, l_0(x) + y_1 \, l_1(x) + y_2 \, l_2(x)$ with

$$l_0(x) = \frac{x - x_1}{x_0 - x_1} \frac{x - x_2}{x_0 - x_2}$$

$$l_1(x) = \frac{x - x_0}{x_1 - x_0} \frac{x - x_2}{x_1 - x_2}$$

$$l_2(x) = \frac{x - x_0}{x_2 - x_0} \frac{x - x_1}{x_2 - x_1}$$

it's a parabolic curve

This method is conceptually simple
but inefficient to implement numerically.

Interpolation. Polynomial interpolation, Newton's method

Newton's method: polynomial interpolation method, similar to Lagrange's method but much better to implement numerically.

The interpolating polynomials are written as

$$P_k(x) = a_{n-k} + (x - x_{n-k})\, P_{k-1}(x)$$

where n is the interpolation order and k = 0, 1, 2, ..., n.

If n = 1 $$\begin{cases} P_0(x) = a_1 \\ P_1(x) = a_0 + (x - x_0)\, P_0(x) \end{cases}$$

If n = 2 $$\begin{cases} P_0(x) = a_2 \\ P_1(x) = a_1 + (x - x_1)\, P_0(x) \\ P_2(x) = a_0 + (x - x_0)\, P_1(x) \end{cases}$$

If n = 3 $$\begin{cases} P_0(x) = a_3 \\ P_1(x) = a_2 + (x - x_2)\, P_0(x) \\ P_2(x) = a_1 + (x - x_1)\, P_1(x) \\ P_3(x) = a_0 + (x - x_0)\, P_2(x) \end{cases}$$

Hence, the k=3 polynomial can be written as

$$P_3(x) = a_0 + (x - x_0)\, \{a_1 + (x - x_1)\, [a_2 + (x - x_2)\, a_3]\}$$

Interpolation. Polynomial interpolation, Newton's method

The coefficients a_k are determined by **forcing the polynomial to pass through each data point $y_i = P_k(x_i)$** with i = 0, 1, .., n and k = n

This yields the simultaneous equations:

$$\begin{aligned} y_0 &= a_0 \\ y_1 &= a_0 + (x_1 - x_0)\, a_1 \\ y_2 &= a_0 + (x_2 - x_0)\, a_1 + (x_2 - x_0)(x_2 - x_1)\, a_2 \\ &\vdots \\ y_n &= a_0 + (x_n - x_0)\, a_1 + (x_n - x_0)(x_n - x_1)\, a_2 + \ldots + (x_n - x_0)\ldots(x_n - x_{n-1})\, a_n \end{aligned}$$

If we write the divided differences as

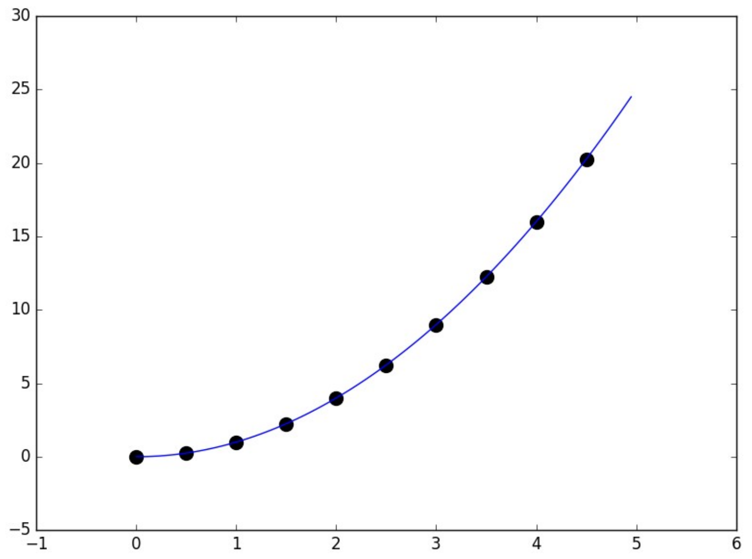
$$\begin{aligned} \nabla y_i &= \frac{y_i - y_0}{x_i - x_0}, \quad i = 1, 2, \ldots, n \\ \nabla^2 y_i &= \frac{\nabla y_i - \nabla y_1}{x_i - x_1}, \quad i = 2, \ldots, n \\ \nabla^3 y_i &= \frac{\nabla^2 y_i - \nabla^2 y_2}{x_i - x_2}, \quad i = 3, \ldots, n \\ &\vdots \\ \nabla^n y_n &= \frac{\nabla^{n-1} y_i - \nabla^{n-1} y_{n-1}}{x_i - x_{n-1}} \end{aligned}$$

The solution of the simultaneous equations becomes

$$a_0 = y_0, \quad a_1 = \nabla y_1, \quad a_2 = \nabla^2 y_2, \quad \ldots, \quad a_n = \nabla^n y_n$$

Interpolation. Polynomial interpolation, Newton’s method

Example script: let’s read it together



```
import numpy as np
import matplotlib.pyplot as plt

def coeffts(xData,yData):
    #Computes the coefficients of Newton polynomial.
    n = len(xData)
    # Number of data points108
    a = yData.copy()
    for k in range(1,n):
        for j in range(k,n):
            a[j] = (a[j]-a[k-1])/(xData[j] - xData[k-1])
        #or more simply
        #a[k:n] = (a[k:n] - a[k-1])/(xData[k:n] - xData[k-1])
    return a

def evalPoly(a,xData,x):
    n = len(xData) - 1
    # Degree of polynomial
    p = a[n]
    for k in range(1,n+1):
        p = a[n-k] + (x -xData[n-k])*p
    return p

#Evaluates Newton polynomial p at x. The coefficient
#vector { a } can be computed by the function coeffts.
def newtonPoly(xData,yData,x):
    a = coeffts(xData,yData)
    p = evalPoly(a,xData,x)
    return p

#main
xData=np.zeros(10,float)
yData=np.zeros(10,float)
x=np.zeros(100,float)
for i in range(len(xData)):
    xData[i]=float(i*0.5)
    yData[i]=float(xData[i]**2)
for i in range(len(x)):
    x[i]=float(i*0.5/10.)

y=newtonPoly(xData,yData,x)


fig, ax1= plt.subplots()
ax1.scatter(xData,yData, marker=’o’, edgecolor=’black’,\
 facecolors=’black’, s=100,zorder=1)
ax1.plot(x,y)
fig.tight_layout()
plt.show()
```

Interpolation. Limitations of polynomial interpolation

Polynomial interpolation should be carried out with the fewest feasible number of data points.

Linear interpolation, using the nearest 2 points, is often sufficient if the data points are closely spaced.

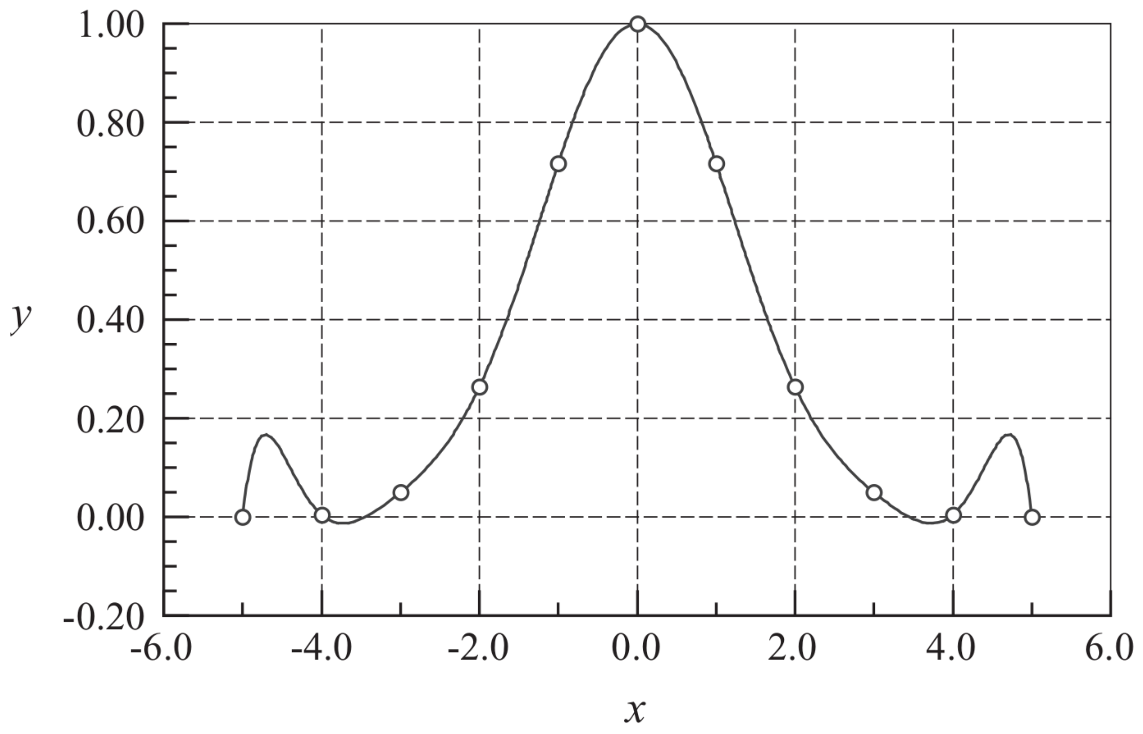
An interpolant intersecting more than 6 points must be viewed with suspicion.
The reason is that the data points that are far from the point of interest do not contribute to the accuracy of the interpolant.

Example of bad interpolation:

* polynomial of degree 10

* 11 data points

* too much **OSCILLATION**

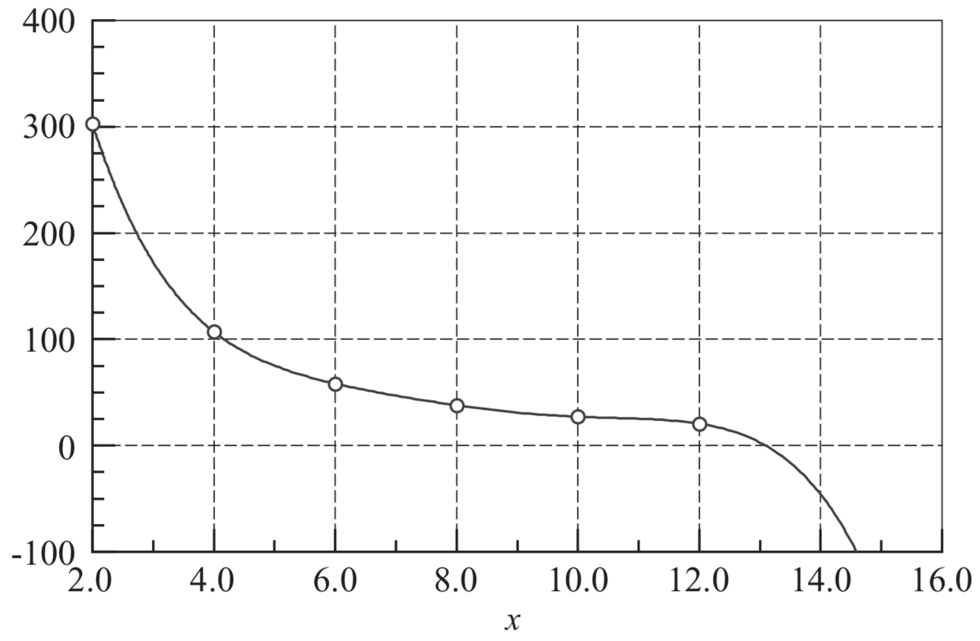


Interpolation. Limitations of polynomial extrapolation

Polynomial extrapolation is even more dangerous.

Example of bad extrapolation:

* polynomial of degree 5
* 6 data points



If extrapolation cannot be avoided, it should be done following some good practice:

• plot the data and visually verify that the extrapolated value makes sense in few test cases;

• use a low-order polynomial: a linear or quadratic polynomial is often the best;

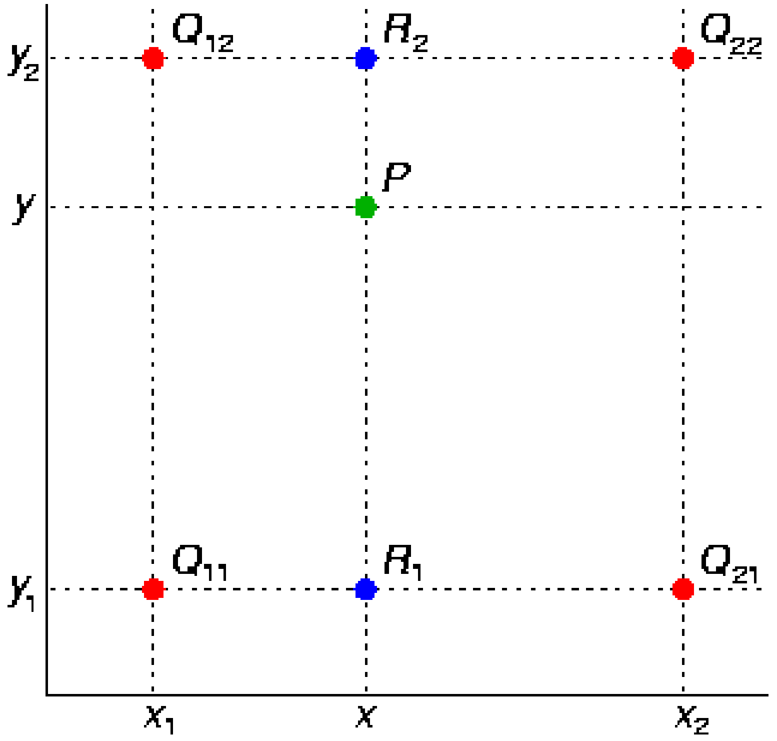
• work with plots of log vs log, which is usually much smoother than the linear axes.

Interpolation. Two dimensional interpolation

We consider the two dimensional linear interpolation case only.

When in 2 dimensions, linear interpolation is called **bilinear interpolation**.

We want to find the value of the unknown function f(x, y) at the point (x, y).
We know the value of f(x, y) at the four points
Q11 = (x1, y1), Q12 = (x1, y2),
Q21 = (x2, y1), and Q22 = (x2, y2)



We first interpolate along the x direction:

$$f(x,y1) \sim \frac{x2-x}{x2-x1} f(Q11) + \frac{x-x1}{x2-x1} f(Q21)$$
$$f(x,y2) \sim \frac{x2-x}{x2-x1} f(Q12) + \frac{x-x1}{x2-x1} f(Q22)$$

We then interpolate along the y axis:

$$f(x,y) \sim \frac{y2-y}{y2-y1} f(x,y1) + \frac{y-y1}{y2-y1} f(x,y2)$$
$$= \frac{y2-y}{y2-y1}\left[\frac{x2-x}{x2-x1} f(Q11) + \frac{x-x1}{x2-x1} f(Q21)\right] + \frac{y-y1}{y2-y1}\left[\frac{x2-x}{x2-x1} f(Q12) + \frac{x-x1}{x2-x1} f(Q22)\right]$$
$$= \frac{\{(y2-y)\,[(x2-x)\,f(Q11)+(x-x1)\,f(Q21)]+(y-y1)\,[(x2-x)\,f(Q12)+(x-x1)\,f(Q22)]\}}{(x2-x1)\,(y2-y1)}$$

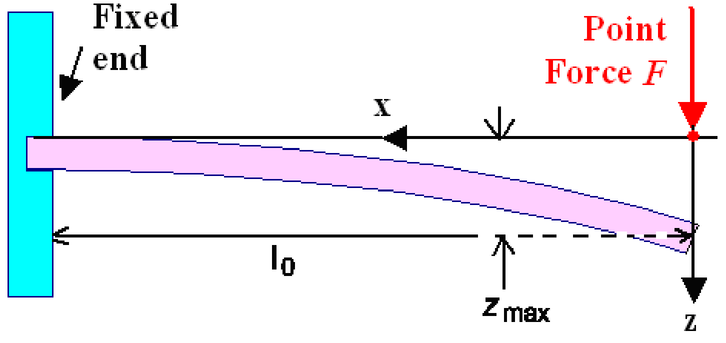
We obtain the same result if we interpolate first along y and then along x.

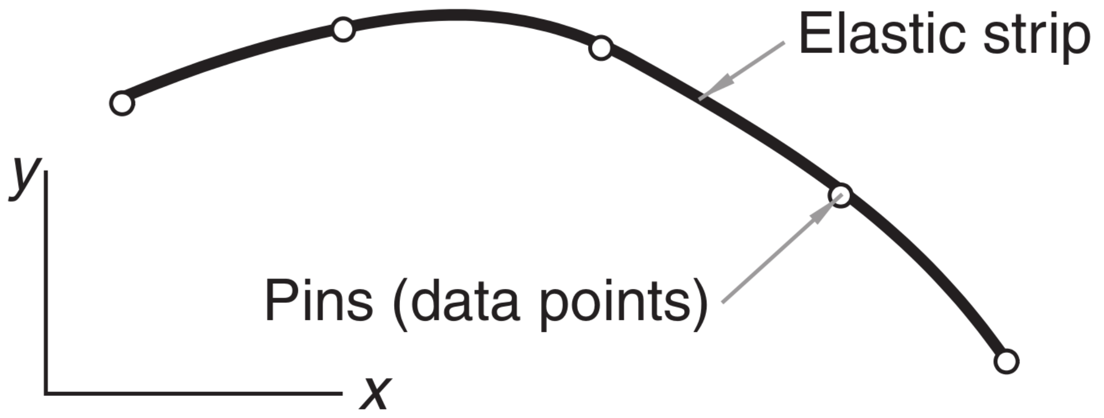
This algorithm works only if the grid points are uniformly spaced.

Interpolation. Cubic spline

Possibly the best interpolation method, especially if there are many data points.

It works like a **thin, elastic beam that is attached with pins to the data points**
The pins, i.e. the data points, are called the **knots**





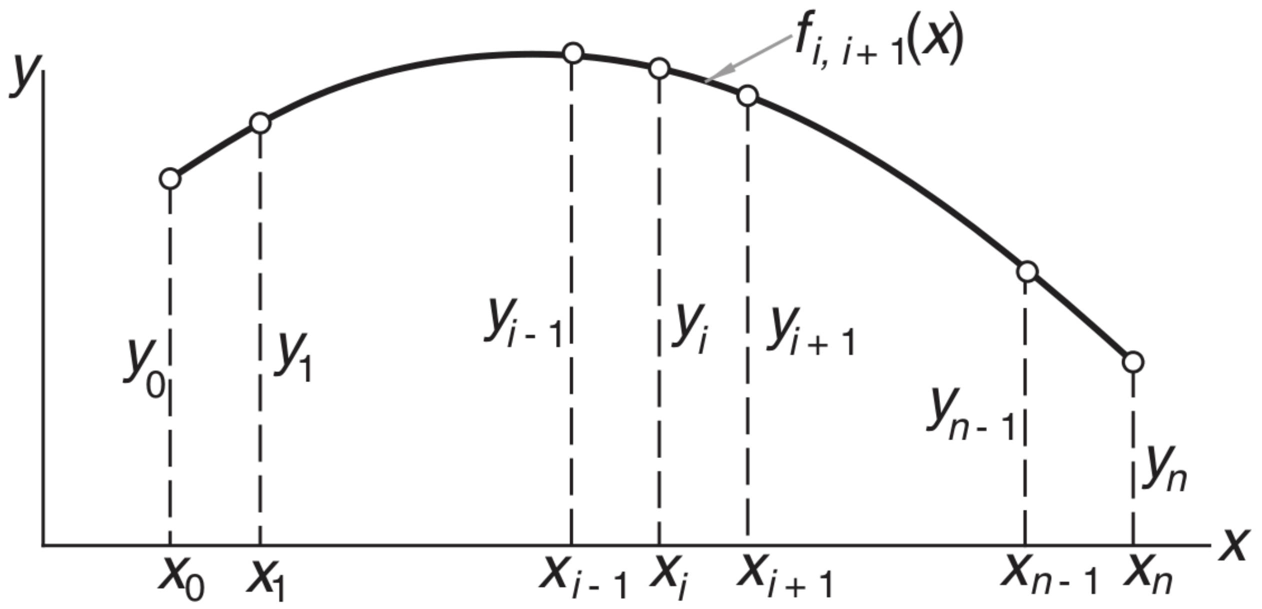
The beam bends under the exerted transverse force,
but does not break: has smooth junctions,
no discontinuities in 1st and 2nd derivatives at the knots

Each segment of the spline curve is a cubic polynomial with smooth junctions:
the first and second derivatives are continuous at the knots.

→ STIFFER than the polynomial but oscillates much less between knots

Interpolation. Cubic spline

Cubic spline that spans n + 1 knots:



$f_{i,i+1}(x)$:= cubic polynomial that spans the segment between knots i and i + 1

The spline is a **piecewise cubic curve**, put together from the **n cubics** $f_{0,1}(x)$, $f_{1,2}(x)$, ..., $f_{n-1,n}(x)$, all of which have **different coefficients**.

Let's call k_i the second derivative of the spline at knot i

Because of the requirement of continuity of second derivatives:

$$f''_{i-1,i}(x_i) = f''_{i,i+1}(x_i) = k_i$$

Interpolation. Cubic spline

Because of the requirement of continuity of second derivatives:

$$f''_{i-1,i}(x_i) = f''_{i,i+1}(x_i) = k_i$$

At this stage the k_i are unknown, apart from $k_0 = k_n = 0$

The starting point for computing the coefficients of $f_{i,i+1}(x)$ is the expression for $f''_{i,i+1}(x)$. Using Lagrange's two-point interpolation, we can write

$$f''_{i,i+1}(x) = k_i\, l_i(x) + k_{i+1}\, l_{i+1}(x)$$

where

$$l_i(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}}, \quad l_{i+1}(x) = \frac{x - x_i}{x_{i+1} - x_i}$$

Hence

$$f''_{i,i+1}(x) = \frac{k_i\,(x - x_{i+1}) - k_{i+1}\,(x - x_i)}{x_i - x_{i+1}}$$

Integrating twice with respect to x we obtain

$$f_{i,i+1}(x) = \frac{k_i\,(x - x_{i+1})^3 - k_{i+1}\,(x - x_i)^3}{6\,(x_i - x_{i+1})} + A\,(x - x_{i+1}) - B\,(x - x_i)$$

Interpolation. Cubic spline

Integrating twice with respect to x we obtain

$$f_{i,i+1}(x) = \frac{k_i\,(x - x_{i+1})^3 - k_{i+1}\,(x - x_i)^3}{6\,(x_i - x_{i+1})} + A\,(x - x_{i+1}) - B\,(x - x_i)$$

where A and B are constants of integration.
Usually, the constants of integration are written in the form C x + D,
but defining C = A − B and D = −A x_{i+1} + B x_i
we end up with two terms that are more handy in the procedure that follows:

Simplifying the notation, we now write $y_i \equiv f_{i,i+1}(x_i)$ and $y_{i+1} \equiv f_{i,i+1}(x_{i+1})$.
Therefore, we can calculate A from y_i and x_i, and B from y_{i+1} and x_{i+1}

$$\frac{k_i\,(x_i - x_{i+1})^3}{6\,(x_i - x_{i+1})} + A\,(x_i - x_{i+1}) = y_i,$$
$$\frac{-k_{i+1}\,(x_{i+1} - x_i)^3}{6\,(x_i - x_{i+1})} - B\,(x_{i+1} - x_i) = y_{i+1}$$

$$\longrightarrow$$

$$A = \frac{y_i}{(x_i - x_{i+1})} - \frac{k_i}{6}\,(x_i - x_{i+1}),$$
$$B = \frac{y_{i+1}}{(x_i - x_{i+1})} - \frac{k_{i+1}}{6}(x_i - x_{i+1})$$

Interpolation. Cubic spline

Substituting back to equation ☺ and rearranging, we have:

$$f_{i,i+1}(x) = \frac{k_i}{6}\left[\frac{(x-x_{i+1})^3}{(x_i-x_{i+1})} - (x_i - x_{i+1})(x - x_{i+1})\right]$$
$$-\frac{k_{i+1}}{6}\left[\frac{(x-x_i)^3}{(x_i-x_{i+1})} - (x_i - x_{i+1})(x - x_i)\right] \qquad ★$$
$$+\frac{y_i(x-x_{i+1}) - y_{i+1}(x-x_i)}{(x_i - x_{i+1})}$$

The only unknown terms in the above equation are the 2nd derivatives at the knots, k_i and k_{i+1}.

They can be obtained from the continuity condition on the slope, i.e. $f'_{i-1,i}(x_i) = f'_{i,i+1}(x_i)$

Calculating the derivative of equation ★ in $(i-1, i)$ and $(i, i+1)$ we obtain

$$k_{i-1}(x_{i-1} - x_i) + 2\,k_i(x_{i-1} - x_{i+1}) + k_{i+1}(x_i - x_{i+1}) = 6\left(\frac{y_{i-1} - y_i}{x_{i-1} - x_i} - \frac{y_i - y_{i+1}}{x_i - x_{i+1}}\right)$$

for $i = 1, 2, \ldots, n-1$

If the data are evenly spaced (i.e., $x_i - x_{i-1} = x_{i+1} - x_i = h$), this further simplifies to

$$k_{i-1} + 4\,k_i + k_{i+1} = \frac{6}{h^2}\left(y_{i-1} - 2\,y_i + y_{i+1}\right)$$

Interpolation. Cubic spline

This gives a system of linear equations that can be solved with one of the methods you already know, to get the k_i terms.

$$k_{i-1}(x_{i-1}-x_i)+2\,k_i(x_{i-1}-x_{i+1})+k_{i+1}(x_i-x_{i+1})=6\left(\frac{y_{i-1}-y_i}{x_{i-1}-x_i}-\frac{y_i-y_{i+1}}{x_i-x_{i+1}}\right)$$

The simplest way to solve it is to observe that the matrix is tridiagonal, i.e. it has non zero elements only on the diagonal (terms i,i) and on the cells (i, i − 1) and (i, i + 1), that is

$$\begin{bmatrix} d_0 & e_0 & 0 & 0 & 0 & 0 \\ c_0 & d_1 & e_1 & 0 & 0 & 0 \\ 0 & c_1 & d_2 & e_2 & 0 & 0 \\ 0 & 0 & c_2 & d_3 & e_3 & 0 \\ 0 & 0 & 0 & c_3 & d_4 & e_4 \\ 0 & 0 & 0 & 0 & c_4 & d_5 \end{bmatrix}$$

Namely, our system is

$$\begin{bmatrix} 2(x_0-x_2) & (x_1-x_2) & 0 & 0 & 0 & 0 \\ (x_1-x_2) & 2(x_1-x_3) & (x_2-x_3) & 0 & 0 & 0 \\ 0 & (x_2-x_3) & 2(x_2-x_4) & (x_3-x_4) & 0 & 0 \\ 0 & 0 & (x_3-x_4) & 2(x_3-x_5) & (x_4-x_5) & 0 \\ 0 & 0 & 0 & (x_4-x_5) & 2(x_4-x_6) & (x_5-x_6) \\ 0 & 0 & 0 & 0 & (x_5-x_6) & 2(x_5-x_7) \end{bmatrix}\begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \end{bmatrix} =$$

$$\begin{bmatrix} 6\left(\frac{y_0-y_1}{x_0-x_1}-\frac{y_1-y_2}{x_1-x_2}\right) \\ 6\left(\frac{y_1-y_2}{x_1-x_2}-\frac{y_2-y_3}{x_2-x_3}\right) \\ 6\left(\frac{y_2-y_3}{x_2-x_3}-\frac{y_3-y_4}{x_3-x_4}\right) \\ 6\left(\frac{y_3-y_4}{x_3-x_4}-\frac{y_4-y_5}{x_4-x_5}\right) \\ 6\left(\frac{y_4-y_5}{x_4-x_5}-\frac{y_5-y_6}{x_5-x_6}\right) \\ 6\left(\frac{y_5-y_6}{x_5-x_6}-\frac{y_6-y_7}{x_6-x_7}\right) \end{bmatrix}$$

Tridiagonal matrices are easy to solve with the LU decomposition

Interpolation. Cubic spline

Steps to take to calculate the cubic spline:

1. Calculate the k_i by solving the system of linear equations (remember that $k_0 = k_n = 0$ and you only have to calculate k_i with i = 1, 2, ..., n − 1);

2. Calculate the interpolant at x from equation

$$
\begin{aligned}
f_{i,i+1}(x) = \frac{k_i}{6}\left[\frac{(x - x_{i+1})^3}{(x_i - x_{i+1})} - (x_i - x_{i+1})(x - x_{i+1})\right] \\
-\frac{k_{i+1}}{6}\left[\frac{(x - x_i)^3}{(x_i - x_{i+1})} - (x_i - x_{i+1})(x - x_i)\right] \\
+\frac{y_i(x - x_{i+1}) - y_{i+1}(x - x_i)}{(x_i - x_{i+1})}
\end{aligned}
$$

Repeat for all the x points you want.

Interpolation. Cubic spline

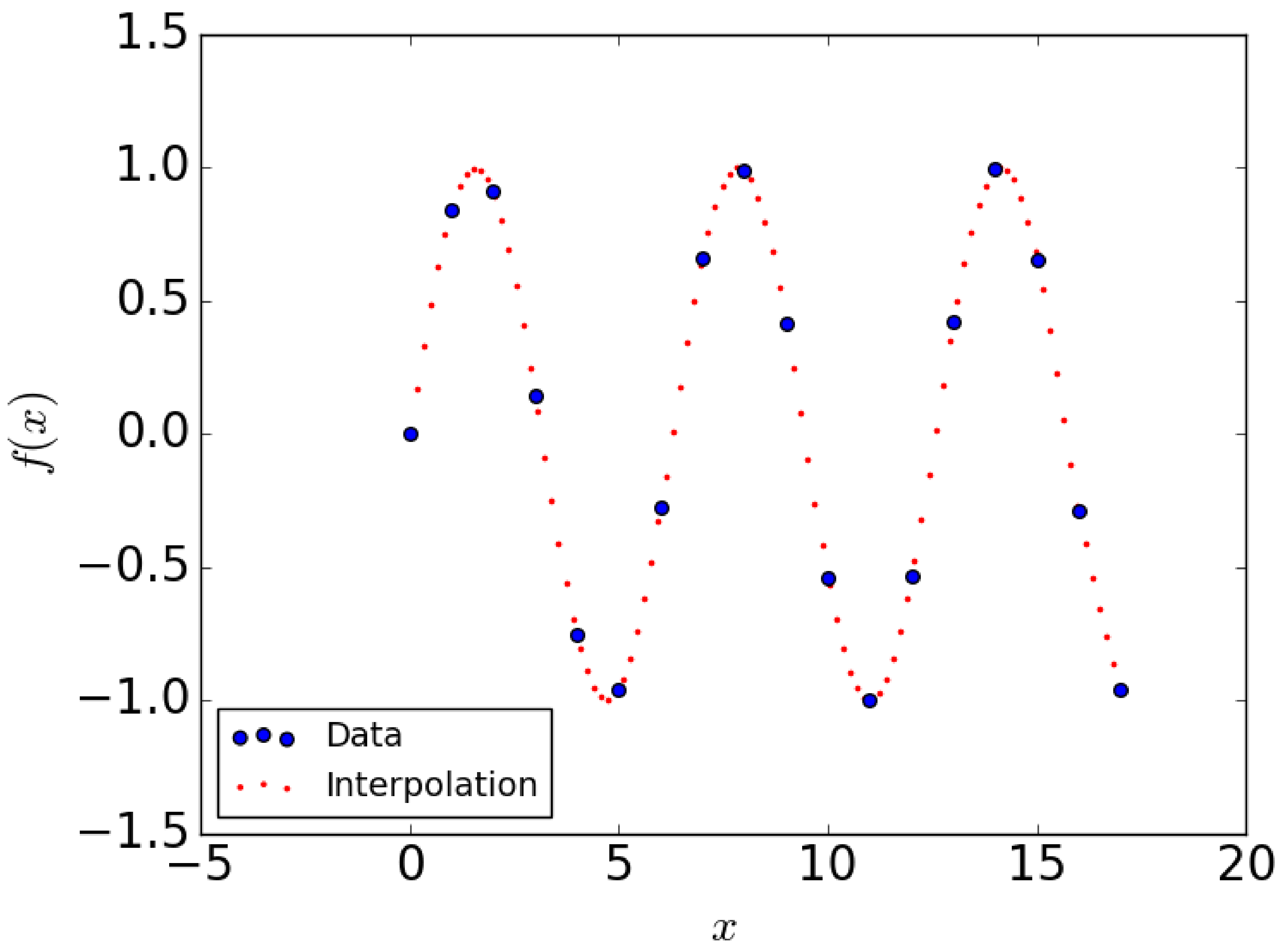
EXERCISE:

Write a script to implement the cubic spline. To calculate the coefficients k_i use the LU decomposition through numpy.linalg.solve. Apply it to the following sample of fake data.

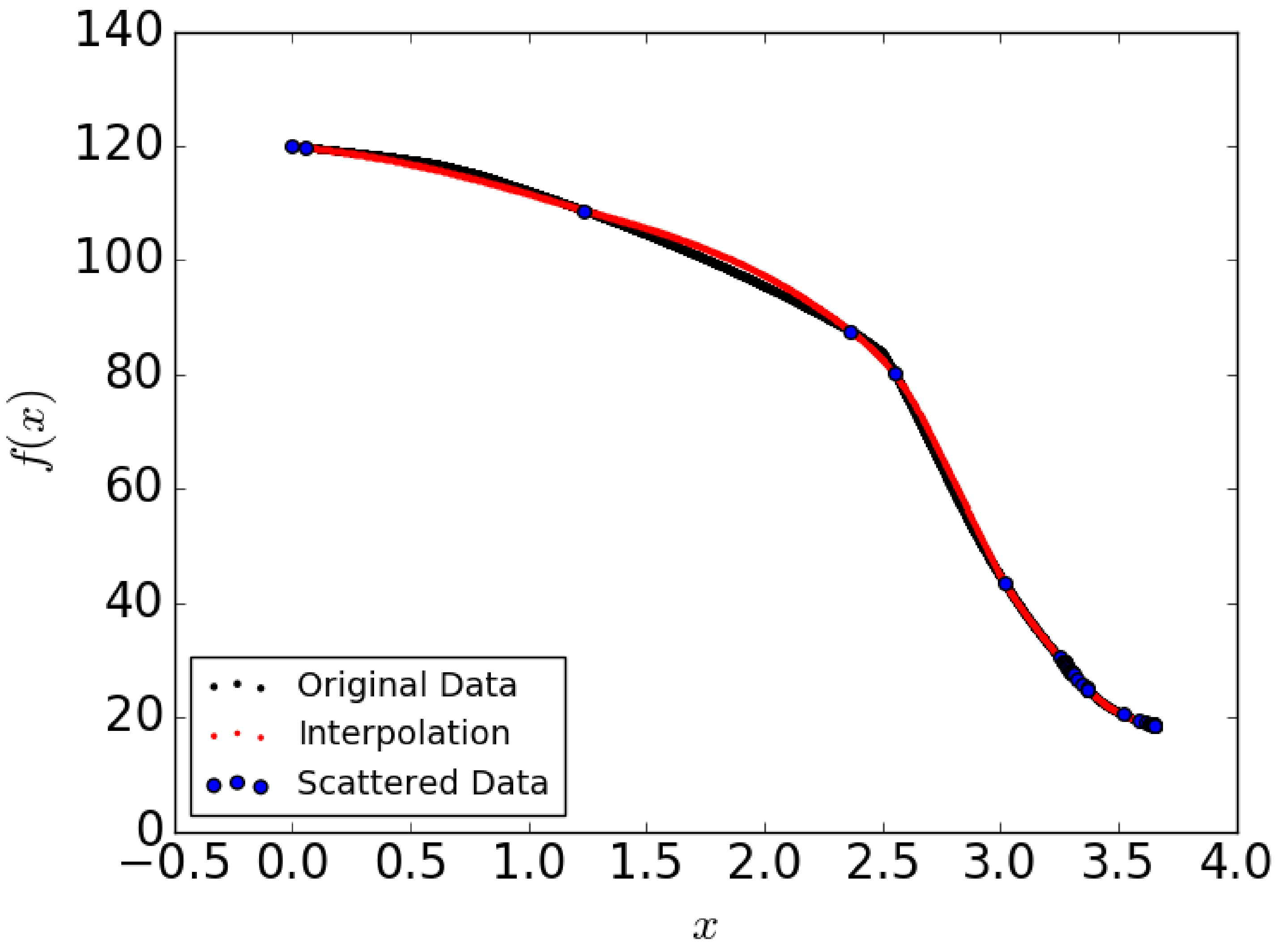
```
import numpy as np
N=18
x=np.arange(0,18,1.)
y=np.sin(x)
```

The result should look as the upper panel of Fig. 62.
Now, apply this script to the file evol_120msun_scattered.dat and reconstruct the evolution of the 120 $M_\odot$ star with the spline. The result should look as the lower panel of Fig. 62.

Interpolation. Cubic spline



Interpolation. Cubic spline



Interpolation. Python modules to interpolate

1. **numpy.interp()**
   returns the one-dimensional piecewise linear interpolant to a function with given discrete data points (xp, yp), evaluated at x.

   It is analogous to our linear interpolation script.
   Reliable, robust, highly recommended.
   Syntax:

   y=numpy.interp(x,xp,yp)

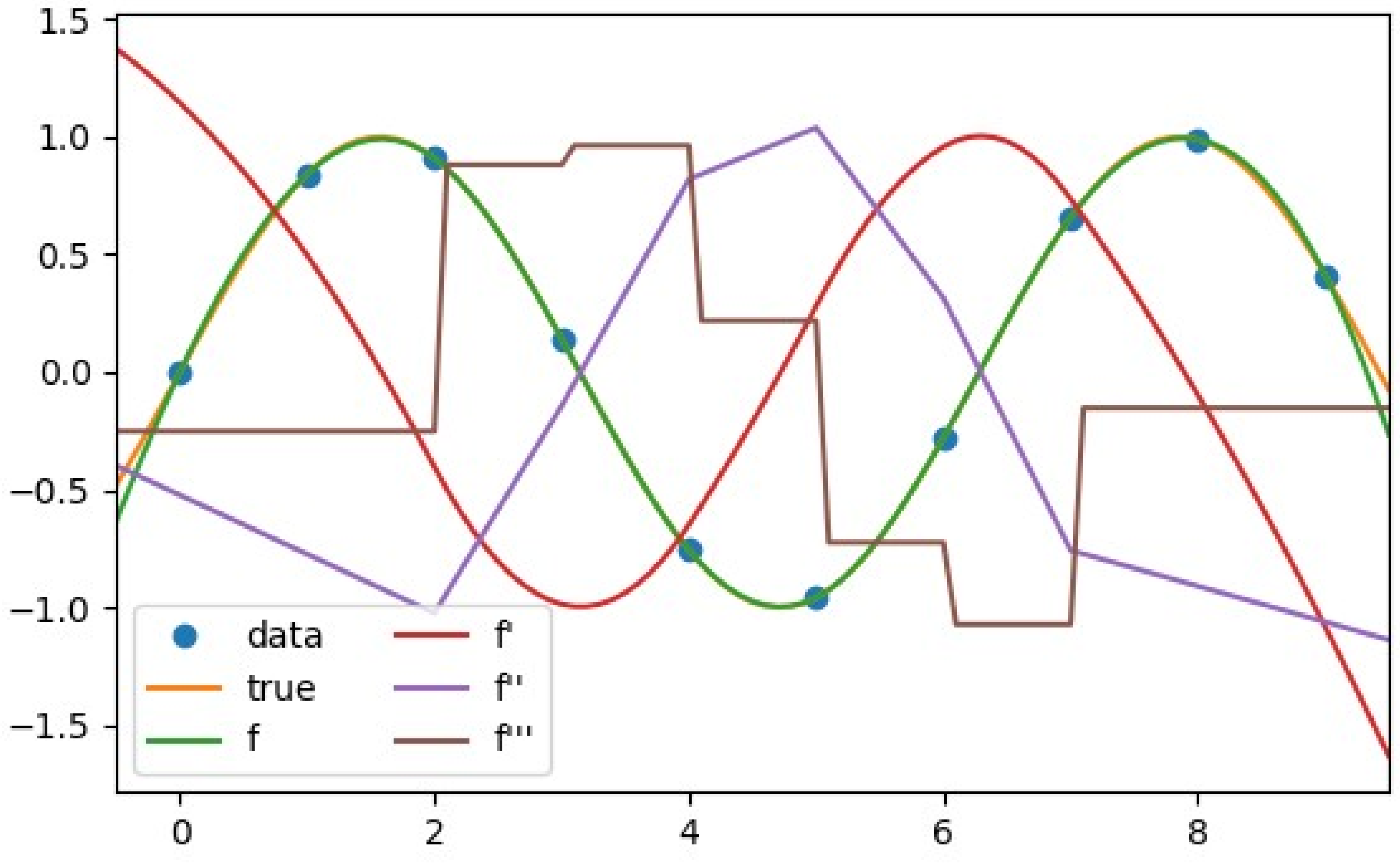
2. **scipy.interpolate** contains several different functions for interpolation.

   Let us just mention **scipy.interpolate.CubicSpline** that interpolates using a cubic spline,
   i.e. interpolates data with a piecewise cubic polynomial which is twice continuously differentiable.

Interpolation. Example of scipy.interpolate.CubicSpline

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

x = np.arange(10)

y = np.sin(x)

cs = CubicSpline(x, y) # x is the array of data,
#y is the array of values of the function in these data

xs = np.arange(-0.5, 9.6, 0.1)

fig, ax = plt.subplots(figsize=(6.5, 4))

ax.plot(x, y, ’o’, label=’data’) #scattered data

ax.plot(xs, np.sin(xs), label=’true’) #true value of the function

ax.plot(xs, cs(xs), label="f") #interpolated value of the function

ax.plot(xs, cs(xs, 1), label="f’") #first derivative of f

ax.plot(xs, cs(xs, 2), label="f’’")  #second derivative of f

ax.plot(xs, cs(xs, 3), label="f’’’")  #third derivative of f

ax.set_xlim(-0.5, 9.5)

ax.legend(loc=’lower left’, ncol=2)

plt.show()
```

Interpolation. Example of scipy.interpolate.CubicSpline


data
true
f
f'
f''
f'''