Chapter 6

Numerical interpolation, extrapolation and fi tting of data

6.1 Introduction

Numerical interpolation and extrapolation is perhaps one of the most used tools in numerical applications to physics. The often encountered situation is that of a function f at a set of points $x_1 \dots x_n$ where an analytic form is missing. The function f may represent some data points from experiment or the result of a lengthy large-scale computation of some physical quantity that cannot be cast into a simple analytical form.

We may then need to evaluate the function f at some point x within the data set $x_1 \dots x_n$, but where x differs from the tabulated values. In this case we are dealing with interpolation. If x is outside we are left with the more troublesome problem of numerical extrapolation. Below we will concentrate on two methods for interpolation and extrapolation, namely polynomial interpolation and extrapolation and the qubic spline interpolation approach.

6.2 Interpolation and extrapolation

6.2.1 Polynomial interpolation and extrapolation

Let us assume that we have a set of $N+1$ points $y_0 = f(x_0), y_1 = f(x_1), \dots, y_N = f(x_N)$ where none of the x_i values are equal. We wish to determine a polynomial of degree n so that

$$P_N(x_i) = f(x_i) = y_i, \qquad i = 0, 1, \dots, N \tag{6.1}$$

for our data points. If we then write P_n on the form

$$P_N(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_N(x - x_0)\dots(x - x_{N-1}), \tag{6.2}$$

then Eq. (6.1) results in a triangular system of equations

$$\begin{array}{llll} a_0 & = f(x_0) & & \\ a_0 + & a_1(x_1 - x_0) & = f(x_1) & \\ a_0 + & a_1(x_2 - x_0) + & a_2(x_2 - x_0)(x_2 - x_1) & = f(x_2) \\ \dots & \dots & \dots & \dots \end{array} . \tag{6.3}$$

The coefficients $a_0, \ldots, a_N$ are then determined in a recursive way, starting with $a_0, a_1, \ldots$.

The classic of interpolation formulae was created by Lagrange and is given by

$$P_N(x) = \sum_{i=0}^{N} \prod_{k \neq i} \frac{x - x_k}{x_i - x_k} y_i. \tag{6.4}$$

If we have just two points (a straight line) we get

$$P_1(x) = \frac{x - x_0}{x_1 - x_0} y_1 + \frac{x - x_1}{x_0 - x_1} y_0, \tag{6.5}$$

and with three points (a parabolic approximation) we have

$$P_3(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} y_2 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} y_1 + \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} y_0 \tag{6.6}$$

and so forth. It is easy to see from the above equations that when $x = x_i$ we have that $f(x) = f(x_i)$ It is also possible to show that the approximation error (or rest term) is given by the second term on the right hand side of

$$f(x) = P_N(x) + \frac{\omega_{N+1}(x) f^{(N+1)}(\xi)}{(N+1)!}. \tag{6.7}$$

The function $\omega_{N+1}(x)$ is given by

$$\omega_{N+1}(x) = a_N(x - x_0) \ldots (x - x_N), \tag{6.8}$$

and $\xi = \xi(x)$ is a point in the smallest interval containing all interpolation points x_j and x. The algorithm we provide however (the code POLINT in the program library) is based on divided differences. The recipe is quite simple. If we take $x = x_0$ in Eq. (6.2), we then have obviously that $a_0 = f(x_0) = y_0$. Moving a_0 over to the left-hand side and dividing by $x - x_0$ we have

$$\frac{f(x) - f(x_0)}{x - x_0} = a_1 + a_2(x - x_1) + \cdots + a_N(x - x_1)(x - x_2) \ldots (x - x_{N-1}), \tag{6.9}$$

where we hereafter omit the rest term

$$\frac{f^{(N+1)}(\xi)}{(N+1)!}(x - x_1)(x - x_2) \ldots (x - x_N). \tag{6.10}$$

The quantity

$$f_{0x} = \frac{f(x) - f(x_0)}{x - x_0}, \tag{6.11}$$

is a divided difference of first order. If we then take $x = x_1$, we have that $a_1 = f_{01}$. Moving a_1 to the left again and dividing by $x - x_1$ we obtain

$$\frac{f_{0x} - f_{01}}{x - x_1} = a_2 + \cdots + a_N(x - x_2) \ldots (x - x_{N-1}). \tag{6.12}$$

and the quantity

$$f_{01x} = \frac{f_{0x} - f_{01}}{x - x_1}, \tag{6.13}$$

is a divided difference of second order. We note that the coefficient

$$a_1 = f_{01}, \tag{6.14}$$

is determined from f_{0x} by setting $x = x_1$. We can continue along this line and define the divided difference of order $k+1$ as

$$f_{01\ldots kx} = \frac{f_{01\ldots(k-1)x} - f_{01\ldots(k-1)k}}{x - x_k}, \tag{6.15}$$

meaning that the corresponding coefficient a_k is given by

$$a_k = f_{01\ldots(k-1)k}. \tag{6.16}$$

With these definitions we see that Eq. (6.7) can be rewritten as

$$f(x) = a_0 + \sum_{k=1} N f_{01\ldots k}(x - x_0)\ldots(x - x_{k-1}) + \frac{\omega_{N+1}(x) f^{(N+1)}(\xi)}{(N+1)!}. \tag{6.17}$$

If we replace $x_0, x_1, \ldots, x_k$ in Eq. (6.15) with $x_{i+1}, x_{i+2}, \ldots, x_k$, that is we count from $i+1$ to k instead of counting from 0 to k and replace x with x_i, we can then construct the following recursive algorithm for the calculation of divided differences

$$f_{x_i x_{i+1}\ldots x_k} = \frac{f_{x_{i+1}\ldots x_k} - f_{x_i x_{i+1}\ldots x_{k-1}}}{x_k - x_i}. \tag{6.18}$$

Assuming that we have a table with function values $(x_j, f(x_j) = y_j)$ and need to construct the coefficients for the polynomial $P_N(x)$. We can then view the last equation by constructing the following table for the case where $N = 3$.

$$\begin{array}{llllll} x_0 & y_0 & & & & \\ & & f_{x_0x_1} & & & \\ x_1 & y_1 & & f_{x_0x_1x_2} & & \\ & & f_{x_1x_2} & & f_{x_0x_1x_2x_3} & . \\ x_2 & y_2 & & f_{x_1x_2x_3} & & \\ & & f_{x2x_3} & & & \\ x_3 & y_3 & & & & \end{array} \tag{6.19}$$

The coefficients we are searching for will then be the elements along the main diagonal. We can understand this algorithm by considering the following. First we construct the unique polynomial of order zero which passes through the point x_0, y_0. This is just a_0 discussed above. Therafter we construct the unique polynomial of order one which passes through both x_0y_0 and x_1y_1. This corresponds to the coefficient a_1 and the tabulated value $f_{x_0x_1}$ and together with a_0 results in the polynomial for a straight line. Likewise we define polynomial coefficients for all other couples of points such as $f_{x_1x_2}$ and $f_{x_2x_3}$. Furthermore, a coefficient like $a_2 = f_{x_0x_1x_2}$ spans now three points, and adding together $f_{x_0x_1}$ we obtain a polynomial which represents three points, a parabola. In this fashion we can continue till we have all coefficients. The function POLINT included in the library is based on an extension of this algorithm, knowns as Neville's algorithm. It is based on equidistant interpolation points. The error provided by the call to the function POLINT is based on the truncation error in Eq. (6.7).

Exercise 6.1
Use the function $f(x) = x^3$ to generate function values at four points $x_0 = 0$, $x_1 = 1$, $x_2 = 5$ and $x_3 = 6$. Use the above described method to show that the interpolating polynomial becomes $P_3(x) = x + 6x(x-1) + x(x-1)(x-5)$. Compare the exact answer with the polynomial P_3 and estimate the rest term.

6.3 Richardson's deferred extrapolation method

Here we present an elegant method to improve the precision of our mathematical truncation, without too many additional function evaluations. We will again study the evaluation of the first and second derivatives of $\exp(x)$ at a given point $x = \xi$. In Eqs. (3.1) and (3.2) for the first and second derivatives, we noted that the truncation error goes like $O(h^{2j})$.

Employing the mid-point approximation to the derivative, the various derivatives D of a given function $f(x)$ can then be written as

$$D(h) = D(0) + a_1 h^2 + a_2 h^4 + a_3 h^6 + \ldots, \tag{6.20}$$

where $D(h)$ is the calculated derivative, $D(0)$ the exact value in the limit $h \to 0$ and a_i are independent of h. By choosing smaller and smaller values for h, we should in principle be able to approach the exact value. However, since the derivatives involve differences, we may easily loose numerical precision as shown in the previous sections. A possible cure is to apply Richardson's deferred approach, i.e., we perform calculations with several values of the step h and extrapolate to $h = 0$. The philososphy is to combine different values of h so that the terms in the above equation involve only large exponents for h. To see this, assume that we mount a calculation for two values of the step h, one with h and the other with $h/2$. Then we have

$$D(h) = D(0) + a_1 h^2 + a_2 h^4 + a_3 h^6 + \ldots, \tag{6.21}$$

and

$$D(h/2) = D(0) + \frac{a_1 h^2}{4} + \frac{a_2 h^4}{16} + \frac{a_3 h^6}{64} + \ldots, \tag{6.22}$$

and we can eliminate the term with a_1 by combining

$$D(h/2) + \frac{D(h/2) - D(h)}{3} = D(0) - \frac{a_2 h^4}{4} - \frac{5 a_3 h^6}{16}. \tag{6.23}$$

We see that this approximation to $D(0)$ is better than the two previous ones since the error now goes like $O(h^4)$. As an example, let us evaluate the first derivative of a function f using a step with lengths h and $h/2$. We have then

$$\frac{f_h - f_{-h}}{2h} = f_0' + O(h^2), \tag{6.24}$$

$$\frac{f_{h/2} - f_{-h/2}}{h} = f_0' + O(h^2/4), \tag{6.25}$$

which can be combined, using Eq. (6.23) to yield

$$\frac{-f_h + 8f_{h/2} - 8f_{-h/2} + f_{-h}}{6h} = f_0' - \frac{h^4}{480} f^{(5)}. \tag{6.26}$$

In practice, what happens is that our approximations to $D(0)$ goes through a series of steps

$$\begin{array}{cccc} D_0^{(0)} & & & \\ D_0^{(1)} & D_1^{(0)} & & \\ D_0^{(2)} & D_1^{(1)} & D_2^{(0)} & \\ D_0^{(3)} & D_1^{(2)} & D_2^{(1)} & D_3^{(0)} \\ \ldots & \ldots & \ldots & \ldots \end{array}, \tag{6.27}$$

where the elements in the first column represent the given approximations

$$D_0^{(k)} = D(h/2^k). \tag{6.28}$$

This means that $D_1^{(0)}$ in the second column and row is the result of the extrapolating based on $D_0^{(0)}$ and $D_0^{(1)}$. An element $D_m^{(k)}$ in the table is then given by

$$D_m^{(k)} = D_{m-1}^{(k)} + \frac{D_{m-1}^{(k+1)} - D_{m-1}^{(k)}}{4^m - 1} \tag{6.29}$$

with $m > 0$. I.e., it is a linear combination of the element to the left of it and the element right over the latter.

In Table 3.1 we presented the results for various step sizes for the second derivative of $\exp(x)$ using $f_0'' = \frac{f_h - 2f_0 + f_{-h}}{h^2}$. The results were compared with the exact ones for various x values. Note well that as the step is decreased we get closer to the exact value. However, if it is further increased, we run into problems of loss of precision. This is clearly seen for $h = 0000001$. This means that even though we could let the computer run with smaller and smaller values of the step, there is a limit for how small the step can be made before we loose precision. Consider now the results in Table 6.1 where we choose to employ Richardson's extrapolation scheme. In this calculation we have computed our function with only three possible values for the step size, namely h, $h/2$ and $h/4$ with $h = 0.1$. The agreement with the exact value is amazing! The extrapolated result is based upon the use of Eq. (6.29). We will use this

x	$h = 0.1$	$h = 0.05$	$h = 0.025$	Extrapolat	Error
0.0	1.00083361	1.00020835	1.00005208	1.00000000	0.00000000
1.0	2.72054782	2.71884818	2.71842341	2.71828183	0.00000001
2.0	7.39521570	7.39059561	7.38944095	7.38905610	0.00000003
3.0	20.10228045	20.08972176	20.08658307	20.08553692	0.00000009
4.0	54.64366366	54.60952560	54.60099375	54.59815003	0.00000024
5.0	148.53687797	148.44408109	148.42088912	148.41315910	0.00000064

Table 6.1: Result for numerically calculated second derivatives of $\exp(x)$ using extrapolation. The first three values are those calculated with three different step sizes, h, $h/2$ and $h/4$ with $h = 0.1$. The extrapolated result to $h = 0$ should then be compared with the exact ones from Table 3.1.

method to obtain improved eigenvalues in chapter 12.

6.4 Qubic spline interpolation

Qubic spline interpolation is among one of the mostly used methods for interpolating between data points where the arguments are organized as ascending series. In the library program we supply such a function, based on the so-called qubic spline method to be described below.

A spline function consists of polynomial pieces defined on subintervals. The different subintervals are connected via various continuity relations.

Assume we have at our disposal $n + 1$ points $x_0, x_1, \ldots x_n$ arranged so that $x_0 < x_1 < x_2 < \ldots x_{n-1} < x_n$ (such points are called knots). A spline function s of degree k with $n + 1$ knots is defined as follows

– On every subinterval $[x_{i-1}, x_i)$ s is a polynomial of degree $\leq k$.

– s has $k-1$ continuous derivatives in the whole interval $[x_0, x_n]$.

As an example, consider a spline function of degree $k = 1$ defined as follows

$$s(x) = \begin{cases} s_0(x) = a_0 x + b_0 & x \in [x_0, x_1) \\ s_1(x) = a_1 x + b_1 & x \in [x_1, x_2) \\ \dots & \dots \\ s_{n-1}(x) = a_{n-1} x + b_{n-1} & x \in [x_{n-1}, x_n] \end{cases} \tag{6.30}$$

In this case the polynomial consists of series of straight lines connected to each other at every endpoint. The number of continuous derivatives is then $k - 1 = 0$, as expected when we deal with straight lines. Such a polynomial is quite easy to construct given $n+1$ points $x_0, x_1, \dots x_n$ and their corresponding function values.

The most commonly used spline function is the one with $k = 3$, the so-called qubic spline function. Assume that we have in adddition to the $n + 1$ knots a series of functions values $y_0 = f(x_0), y_1 = f(x_1), \dots y_n = f(x_n)$. By definition, the polynomials s_{i-1} and s_i are thence supposed to interpolate the same point i, i.e.,

$$s_{i-1}(x_i) = y_i = s_i(x_i), \tag{6.31}$$

with $1 \leq i \leq n-1$. In total we have n polynomials of the type

$$s_i(x) = a_{i0} + a_{i1} x + a_{i2} x^2 + a_{i2} x^3, \tag{6.32}$$

yielding $4n$ coefficients to determine. Every subinterval provides in addition the $2n$ conditions

$$y_i = s(x_i), \tag{6.33}$$

and

$$s(x_{i+1}) = y_{i+1}, \tag{6.34}$$

to be fulfilled. If we also assume that s' and s'' are continuous, then

$$s'_{i-1}(x_i) = s'_i(x_i), \tag{6.35}$$

yields $n-1$ conditions. Similarly,

$$s''_{i-1}(x_i) = s''_i(x_i), \tag{6.36}$$

results in additional $n-1$ conditions. In total we have $4n$ coefficients and $4n-2$ equations to determine them, leaving us with 2 degrees of freedom to be determined.

Using the last equation we define two values for the second derivative, namely

$$s''_i(x_i) = f_i, \tag{6.37}$$

and

$$s''_i(x_{i+1}) = f_{i+1}, \tag{6.38}$$

and setting up a straight line between f_i and f_{i+1} we have

$$s''_i(x) = \frac{f_i}{x_{i+1} - x_i}(x_{i+1} - x) + \frac{f_{i+1}}{x_{i+1} - x_i}(x - x_i), \tag{6.39}$$

and integrating twice one obtains

$$s_i(x) = \frac{f_i}{6(x_{i+1}-x_i)}(x_{i+1}-x)^3 + \frac{f_{i+1}}{6(x_{i+1}-x_i)}(x-x_i)^3 + c(x-x_i) + d(x_{i+1}-x). \quad (6.40)$$

Using the conditions $s_i(x_i) = y_i$ and $s_i(x_{i+1}) = y_{i+1}$ we can in turn determine the constants c and d resulting in

$$\begin{aligned} s_i(x) = & \quad \frac{f_i}{6(x_{i+1}-x_i)}(x_{i+1}-x)^3 + \frac{f_{i+1}}{6(x_{i+1}-x_i)}(x-x_i)^3 \\ + & \quad (\frac{y_{i+1}}{x_{i+1}-x_i} - \frac{f_{i+1}(x_{i+1}-x_i)}{6})(x-x_i) + (\frac{y_i}{x_{i+1}-x_i} - \frac{f_i(x_{i+1}-x_i)}{6})(x_{i+1}-x). \end{aligned} \quad (6.41)$$

How to determine the values of the second derivatives f_i and f_{i+1}? We use the continuity assumption of the first derivatives

$$s'_{i-1}(x_i) = s'_i(x_i), \quad (6.42)$$

and set $x = x_i$. Defining $h_i = x_{i+1} - x_i$ we obtain finally the following expression

$$h_{i-1}f_{i-1} + 2(h_i + h_{i-1})f_i + h_i f_{i+1} = \frac{6}{h_i}(y_{i+1}-y_i) - \frac{6}{h_{i-1}}(y_i - y_{i-1}), \quad (6.43)$$

and introducing the shorthands $u_i = 2(h_i + h_{i-1})$, $v_i = \frac{6}{h_i}(y_{i+1}-y_i) - \frac{6}{h_{i-1}}(y_i-y_{i-1})$, we can reformulate the problem as a set of linear equations to be solved through e.g., Gaussian elemination, namely

$$\begin{bmatrix} u_1 & h_1 & 0 & \dots & & & & \\ h_1 & u_2 & h_2 & 0 & \dots & & & \\ 0 & h_2 & u_3 & h_3 & 0 & \dots & & \\ \dots & & \dots & \dots & \dots & \dots & \dots & \\ & \dots & & & 0 & h_{n-3} & u_{n-2} & h_{n-2} \\ & & & & & 0 & h_{n-2} & u_{n-1} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \dots \\ f_{n-2} \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \dots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}. \quad (6.44)$$

Note that this is a set of tridiagonal equations and can be solved through only $O(n)$ operations. The functions supplied in the program library are *spline* and *splint*. In order to use qubic spline interpolation you need first to call

```
spline(double x[], double y[], int n, double yp1, double yp2, double y2[])
```

This function takes as input $x[0,..,n-1]$ and $y[0,..,n-1]$ containing a tabulation $y_i = f(x_i)$ with $x_0 < x_1 < .. < x_{n-1}$ together with the first derivatives of $f(x)$ at x_0 and x_{n-1}, respectively. Then the function returns $y2[0,..,n-1]$ which contanin the second derivatives of $f(x_i)$ at each point x_i. n is the number of points. This function provides the qubic spline interpolation for all subintervals and is called only once. Thereafter, if you wish to make various interpolations, you need to call the function

```
splint(double x[], double y[], double y2a[], int n, double x, double *y)
```

which takes as input the tabulated values $x[0,..,n-1]$ and $y[0,..,n-1]$ and the output y2a[0,..,n - 1] from *spline*. It returns the value y corresponding to the point x.