3.2.1 Interpolation

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

$$P_N(x_i) = f(x_i) = y_i, \qquad i = 0, 1, \ldots, N \tag{3.7}$$

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

$$P_N(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_N(x - x_0)\ldots(x - x_{N-1}), \tag{3.8}$$

then Eq. (3.7) results in a triangular system of equations

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

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

$$P_N(x) = \sum_{i=0}^{N} \prod_{k \neq i} \frac{x - x_k}{x_i - x_k} y_i. \tag{3.9}$$

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

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

and with three points (a parabolic approximation) we have

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

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

$$f(x) = P_N(x) + \frac{\omega_{N+1}(x) f^{(N+1)}(\xi)}{(N+1)!}. \tag{3.10}$$

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

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

and $\xi = \xi(x)$ is a point in the smallest interval containing all interpolation points x_j and x. The program we provide below is however based on divided differences. The recipe is quite simple. If we take $x = x_0$ in Eq. (3.8), 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

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

where we hereafter omit the rest term

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

The quantity

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

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

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

and the quantity

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

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

$$a_1 = f_{01},$$

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

$$f_{01\dots kx} = \frac{f_{01\dots(k-1)x} - f_{01\dots(k-1)k}}{x-x_k}, \tag{3.11}$$

meaning that the corresponding coefficient a_k is given by

$$a_k = f_{01\dots(k-1)k}.$$

With these definitions we see that Eq. (3.10) can be rewritten as

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

If we replace $x_0, x_1, \dots, x_k$ in Eq. (3.11) with $x_{i+1}, x_{i+2}, \dots, x_k$, 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

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

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 $N = 3$.

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

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 we provide below included is based on an extension of this algorithm, knowns as Neville's algorithm. The error provided by Neville's algorithm is based on the truncation error in Eq. (3.10).

http://folk.uio.no/mhjensen/compphys/programs/chapter03/cpp/program4.cpp

```
  /*
  ** The function
  **        polint()
  ** takes as input xa[0,..,n-1] and ya[0,..,n-1] together with a given value
  ** of x and returns a value y and an error estimate dy. If P(x) is a polynomial
  ** of degree N - 1 such that P(xa_i) = ya_i, i = 0,..,n-1, then the returned
  ** value is y = P(x).
  */
void polint(double xa[], double ya[], int n, double x, double *y, double *dy)
{
 int    i, m, ns = 1;
 double den,dif,dift,ho,hp,w;
 double *c,*d;

 dif = fabs(x - xa[0]);
 c = new double [n];
 d = new double [n];
 for(i = 0; i < n; i++) {
    if((dift = fabs(x - xa[i])) < dif) {
      ns = i;
  dif = dift;
    }
    c[i] = ya[i];
    d[i] = ya[i];
 }
 *y = ya[ns--];
 for(m = 0; m < (n - 1); m++) {
   for(i = 0; i < n - m; i++) {
      ho = xa[i] - x;
      hp = xa[i + m] - x;
      w = c[i + 1] - d[i];
      if((den = ho - hp) < ZERO) {
        printf("\n\n Error in function polint(): ");
        printf("\nden = ho - hp = %4.1E -- too small\n",den);
        exit(1);
  }
      den = w/den;
      d[i] = hp * den;
      c[i] = ho * den;
    }
    *y += (*dy = (2 * ns < (n - m) ? c[ns + 1] : d[ns--]));
  }
  delete [] d;
  delete [] c;
} // End: function polint()
```

When using this function, you need obviously to declare the function itself.

fulfills the condition of a weak dominance of the diagonal, with $|b_1| > |c_1|$, $|b_n| > |a_n|$ and $|b_k| \geq |a_k| + |c_k|$ for $k = 2,3,\ldots,n-1$. This is a relevant but not sufficient condition to guarantee that the matrix **A** yields a solution to a linear equation problem. The matrix needs also to be irreducible. A tridiagonal irreducible matrix means that all the elements a_i and c_i are non-zero. If these two conditions are present, then **A** is nonsingular and has a unique LU decomposition.

We can obviously extend our boundary value problem to include a first derivative as well

$$-\frac{d^2u(x)}{dx^2} + g(x)\frac{du(x)}{dx} + h(x)u(x) = f(x),$$

with $x \in [a,b]$ and with boundary conditions $u(a) = u(b) = 0$. We assume that f, g and h are continuous functions in the domain $x \in [a,b]$ and that $h(x) \geq 0$. Then the differential equation has a unique solution. We subdivide our interval $x \in [a,b]$ into n subintervals by setting $x_i = a + ih$, with $i = 0,1,\ldots,n+1$. The step size is then given by $h = (b-a)/(n+1)$ with $n \in \mathbb{N}$. For the internal grid points $i = 1,2,\ldots n$ we replace the differential operators with

$$u_i'' \approx \frac{u_{i+1} - 2u_i + u_{i-i}}{h^2}.$$

for the second derivative while the first derivative is given by

$$u_i' \approx \frac{u_{i+1} - u_{i-i}}{2h}.$$

We rewrite our original differential equation in terms of a discretized equation as

$$-\frac{u_{i+1} - 2u_i + u_{i-i}}{h^2} + g_i\frac{u_{i+1} - u_{i-i}}{2h} + h_iu_i = f_i,$$

with $i = 1,2,\ldots,n$. We need to add to this system the two boundary conditions $u(a) = u_0$ and $u(b) = u_{n+1}$. This equation can again be rewritten as a tridiagonal matrix problem. We leave it as an exercise to the reader to find the matrix elements, find the conditions for having weakly dominant diagonal elements and that the matrix is irreducible.

6.5 Spline Interpolation

Cubic spline interpolation is among one of the most 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 cubic spline method to be described below. The linear equation solver we developed in the previous section for tridiagonal matrices can be reused for spline interpolation.

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

$$s(x) = \begin{cases} s_0(x) = a_0x + b_0 & x \in [x_0, x_1) \\ s_1(x) = a_1x + b_1 & x \in [x_1, x_2) \\ \cdots & \cdots \\ s_{n-1}(x) = a_{n-1}x + b_{n-1} & x \in [x_{n-1}, x_n] \end{cases} \tag{6.29}$$

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 $x_0, x_1, \ldots x_n$ and their corresponding function values.

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

$$s_{i-1}(x_i) = y_i = s_i(x_i), \tag{6.30}$$

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

$$s_i(x) = a_{i0} + a_{i1}x + a_{i2}x^2 + a_{i3}x^3, \tag{6.31}$$

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

$$y_i = s(x_i), \tag{6.32}$$

and

$$y_{i+1} = s(x_{i+1}), \tag{6.33}$$

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

$$s'_{i-1}(x_i) = s'_i(x_i), \tag{6.34}$$

yields $n-1$ conditions. Similarly,

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

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

$$s''_i(x_i) = f_i, \tag{6.36}$$

and

$$s''_i(x_{i+1}) = f_{i+1}, \tag{6.37}$$

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

$$s''_i(x) = \frac{f_i}{x_{i+1}-x_i}(x_{i+1}-x) + \frac{f_{i+1}}{x_{i+1}-x_i}(x-x_i), \tag{6.38}$$

and integrating twice one obtains

$$s_i(x) = \frac{f_i}{6(x_{i+1}-x_i)}(x_{i+1}-x)^3 + \frac{f_{i+1}}{6(x_{i+1}-x_i)}(x-x_i)^3 + c(x-x_i) + d(x_{i+1}-x). \tag{6.39}$$

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

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

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

$$s'_{i-1}(x_i) = s'_i(x_i), \quad (6.41)$$

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

$$h_{i-1}f_{i-1} + 2(h_i + h_{i-1})f_i + h_i f_{i+1} = \frac{6}{h_i}(y_{i+1} - y_i) - \frac{6}{h_{i-1}}(y_i - y_{i-1}), \quad (6.42)$$

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

$$\begin{bmatrix} u_1 & h_1 & 0 & \dots & & & \\ h_1 & u_2 & h_2 & 0 & \dots & & \\ 0 & h_2 & u_3 & h_3 & 0 & \dots & \\ \dots & & \dots & \dots & \dots & \dots & \dots \\ & \dots & & 0 & h_{n-3} & u_{n-2} & h_{n-2} \\ & & & & 0 & h_{n-2} & u_{n-1} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \dots \\ f_{n-2} \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \dots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}. \quad (6.43)$$

Note that this is a set of tridiagonal equations and can be solved through only $O(n)$ operations.

It is easy to write your own program for the cubic spline method when you have written a slover for tridiagonal equations. We split the program into two tasks, one which finds the polynomial approximation and one which uses the polynomials approximation to find an interpolated value for a function. These functions are included in the programs of this chapter, see the codes cubicpsline.cpp and cubicsinterpol.cpp. Alternatively, you can solve exercise 6.4!

6.6 Iterative Methods

Till now we have dealt with so-called direct solvers such as Gaussian elimination and LU decomposition. Iterative solvers offer another strategy and are much used in partial differential equations. We start with a guess for the solution and then iterate till the solution does not change anymore.

6.6.1 Jacobi's method

It is a simple method for solving

$$\hat{A}\mathbf{x} = \mathbf{b},$$

where $\hat{A}$ is a matrix and $\mathbf{x}$ and $\mathbf{b}$ are vectors. The vector $\mathbf{x}$ is the unknown.

It is an iterative scheme where we start with a guess for the unknown, and after $k+1$ iterations we have

$$\mathbf{x}^{(k+1)} = \hat{D}^{-1}(\mathbf{b} - (\hat{L} + \hat{U})\mathbf{x}^{(k)}),$$

with $\hat{A} = \hat{D} + \hat{U} + \hat{L}$ and $\hat{D}$ being a diagonal matrix, $\hat{U}$ an upper triangular matrix and $\hat{L}$ a lower triangular matrix.