Table 1: Javascript implementation of Lagrange polynomial interpolation.

Interpolation

In practice one often meets a situation where the function of interest, $f(x)$, is only represented by a discrete set tabulated points, $\{x_i, y_i = f(x_i) \mid i = 1 \dots n\}$, obtained for example by sampling, experimentation, or extensive numerical calculations.

Interpolation means constructing a (smooth) function, called *interpolating function*, which passes exactly through the given points and hopefully approximates the tabulated function in between the tabulated points. Interpolation is a specific case of *curve fitting* in which the fitting function must go exactly through the data points.

The interpolating function can be used for different practical needs like estimating the tabulated function between the tabulated points and estimating the derivatives and integrals involving the tabulated function.

Polynomial interpolation

Polynomial interpolation uses a polynomial as the interpolating function. Given a table of n points, $\{x_i, y_i\}$, one can construct a polynomial $P^{(n-1)}(x)$ of the order $n-1$ which passes exactly through the points. This polynomial can be intuitively written in the *Lagrange form*,

$$P^{(n-1)}(x) = \sum_{i=1}^{n} y_i \prod_{k \neq i}^{n} \frac{x - x_k}{x_i - x_k} . \tag{1}$$

Higher order interpolating polynomials are susceptible to the *Runge phenomenon* – erratic oscillations close to the end-points of the interval as illustrated on Fig. 1.

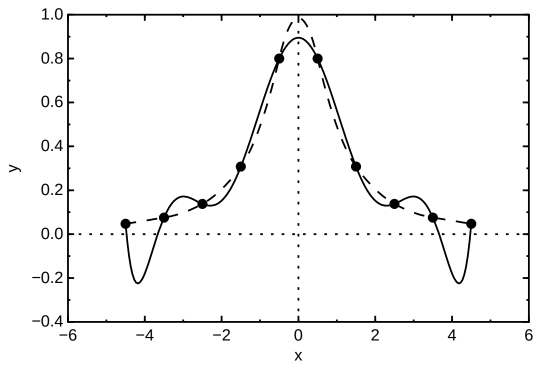

Figure 1: Lagrange interpolating polynomial, solid line, showing the Runge phenomenon: large oscillations at the end-points. Dashed line shows a quadratic spline.

This problem can be avoided by using only the nearest few points instead of all the points in the table (local interpolation) or by using spline interpolation.

Spline interpolation

Spline interpolation uses a *piecewise polynomial*, $S(x)$, called *spline*, as the interpolating function,

$$S(x) = S_i(x) \text{ if } x \in [x_i, x_{i+1}] , \tag{2}$$

where $S_i(x)$ is a polynomial of a given order k.

The spline of the order $k \geq 1$ can be made continuous at the tabulated points,

$$\begin{aligned} S_i(x_i) &= y_i \quad &, \; i = 1, \dots, n-1 \\ S_i(x_{i+1}) &= y_{i+1} \quad &, \; i = 1, \dots, n-1 , \end{aligned} \tag{3}$$

together with its $k-1$ derivatives,

$$\begin{aligned} S_i'(x_{i+1}) &= S_{i+1}'(x_{i+1}) \quad , \ i=1,\dots,n-2 \\ S_i''(x_{i+1}) &= S_{i+1}''(x_{i+1}) \quad , \ i=1,\dots,n-2 \\ &\dots \end{aligned} \tag{4}$$

Continuity conditions (3) and (4) make $kn+n-2k$ linear equations for the $(n-1)(k+1) = kn+n-k-1$ coefficients in $n-1$ polynomials (2) of the order k. The missing $k-1$ conditions can be chosen (reasonably) arbitrarily.

The most popular is the cubic spline, where the polynomials $S_i(x)$ are of third order. The cubic spline is a continuous function together with its first and second derivatives. The cubic spline also has a nice feature that it (sort of) minimizes the total curvature of the interpolating function. This makes the cubic splines look good.

Quadratic spline, which is continuous together with its first derivative, is not nearly as good as the cubic spline in most respects. Particularly it might oscillate unpleasantly when a quick change in the tabulated function is followed by a period where the function is nearly a constant. The cubic spline is less susceptible to such oscillations.

Linear spline is simply a *polygon* drawn through the tabulated points.

Linear interpolation

If the spline polynomials are linear the spline is called *linear interpolation*. The continuity conditions (3) can be satisfied by choosing the spline as

$$S_i(x) = y_i + p_i(x-x_i) \ , \tag{5}$$

where

$$p_i = \frac{\Delta y_i}{\Delta x_i} \ , \ \ \Delta y_i \doteq y_{i+1} - y_i \ , \ \ \Delta x_i \doteq x_{i+1} - x_i \ . \tag{6}$$

Quadratic spline

Quadratic spline is made of second order polynomials, conveniently written in the form

$$S_i(x) = y_i + p_i(x-x_i) + c_i(x-x_i)(x-x_{i+1}) \ , \tag{7}$$

which identically satisfies the spline continuity conditions $S_i(x_i) = y_i$ and $S_i(x_{i+1}) = y_{i+1}$. Substituting (7) into the derivative continuity condition, $S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$, gives $n-2$ equations for $n-1$ unknown coefficients c_i,

$$p_i + c_i\Delta x_i = p_{i+1} - c_{i+1}\Delta x_{i+1} \ . \tag{8}$$

One coefficient can be chosen arbitrarily, for example $c_1 = 0$. The other coefficients can now be calculated recursively,

$$c_{i+1} = \frac{1}{\Delta x_{i+1}}\left(p_{i+1} - p_i - c_i\Delta x_i\right) \ , \ i=1,\dots,n-1 \ . \tag{9}$$

Alternatively, one can choose $c_{n-1} = 0$ and make the backward-recursion

$$c_i = \frac{1}{\Delta x_i}\left(p_{i+1} - p_i - c_{i+1}\Delta x_{i+1}\right) \ , \ i=n-2,\dots,1 \ . \tag{10}$$

In practice, unless you know what your c_1 (or c_{n-1}) is, it is better to run both recursions and then average the resulting c's, which simply amounts to first running the forward-recursion from $c_1 = 0$ and then the backward recursion from $c_{n-1}/2$.

The optimized form (7) of the quadratic spline can aslo be written in the ordinary form, suitable for differentiation and integration, as

$$S_i(x) = y_i + b_i(x-x_i) + c_i(x-x_i)^2 \ , \text{ where } b_i = p_i - c_i\Delta x_i \ . \tag{11}$$

Table 2: Quadratic spline in C++, functional style

```
#include<functional> // to be compiled with -std=c++0x
#include<vector>
using namespace std;

function<double(double)> qspline(vector<double>&x,vector<double>&y)
{ // returns the interpolating function (quadratic spline)

int n=x.size(); vector<double> p(n-1),h(n-1),c(n-1);
for(int i=0;i<n-1;i++){ h[i]=x[i+1]-x[i]; p[i]=(y[i+1]-y[i])/h[i];}

c[0]=0; // recursion up
for(int i=0;i<n-2;i++) c[i+1]=(p[i+1]-p[i]-c[i]*h[i])/h[i+1];

c[n-2]/=2; //recursion down
for(int i=n-3;i>=0;i--) c[i]=(p[i+1]-p[i]-c[i+1]*h[i+1])/h[i];

return [x,y,p,c,n](double z){ // anonymous function
   int i=0, j=n-1; // binary search
   while(j-i>1){int m=(i+j)/2; if(z>x[m]) i=m; else j=m;}
   return y[i]+(p[i]+c[i]*(z-x[i+1]))*(z-x[i]);
   };
}
```

Cubic spline

Cubic splines are made of third order polynomials written e.g. in the form

$$S_i(x) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \ , \tag{12}$$

which automatically satisfies the upper half of continuity conditions (3). The other half of continuity conditions (3) and the continuity of the first and second derivatives (4) give

$$\begin{aligned} y_i + b_i h_i + c_i h_i^2 + d_i h_i^3 &= y_{i+1} \ , \ i = 1, \ldots, n-1 \\ b_i + 2c_i h_i + 3d_i h_i^2 &= b_{i+1} \ , \ i = 1, \ldots, n-2 \\ 2c_i + 6d_i h_i &= 2c_{i+1} \ , \ i = 1, \ldots, n-2 \end{aligned} \tag{13}$$

where

$$h_i = x_{i+1} - x_i \ . \tag{14}$$

The set of equations (13) is a set of $3n - 5$ linear equations for the $3(n - 1)$ unknown coefficients $\{a_i, b_i, c_i \mid i = 1, \ldots, n - 1\}$. Therefore two more equations should be added to the set to find the coefficients. If the two extra equations are also linear, the total system is linear and can be easily solved.

The spline is called *natural* if the extra conditions are given as vanishing second derivative at the end-points,

$$S''(x_1) = S''(x_n) = 0 \ , \tag{15}$$

which gives

$$\begin{aligned} c_1 &= 0 \ , \\ c_{n-1} + 3d_{n-1}h_{n-1} &= 0 \ . \end{aligned} \tag{16}$$

Solving the first two equations in (13) for c_i and d_i gives[1]

$$\begin{aligned} c_i h_i &= -2b_i - b_{i+1} + 3p_i \ , \\ d_i h_i^2 &= b_i + b_{i+1} - 2p_i \ , \end{aligned} \tag{17}$$

where $p_i \doteq \frac{\Delta y_i}{h_i}$. The natural conditions (16) and the third equation in (13) then produce the following tridiagonal system of n linear equations for the n coefficients b_i,

$$\begin{aligned} 2b_1 + b_2 &= 3p_1 \ , \\ b_i + \left(2\frac{h_i}{h_{i+1}} + 2\right)b_{i+1} + \frac{h_i}{h_{i+1}}b_{i+2} &= 3\left(p_i + p_{i+1}\frac{h_i}{h_{i+1}}\right) \ , \ i = 1, \ldots, n-2 \\ b_{n-1} + 2b_n &= 3p_{n-1} \ , \end{aligned} \tag{18}$$

[1]introducing an auxiliary coefficient b_n

or, in the matrix form,

$$\begin{pmatrix} D_1 & Q_1 & 0 & 0 & \dots \\ 1 & D_2 & Q_2 & 0 & \dots \\ 0 & 1 & D_3 & Q_3 & \dots \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ \dots & \dots & 0 & 1 & D_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} B_1 \\ \vdots \\ \vdots \\ B_n \end{pmatrix} \tag{19}$$

where the elements D_i at the main diagonal are

$$D_1 = 2 \; ; \; D_{i+1} = 2\frac{h_i}{h_{i+1}} + 2 \text{ for } i = 1, \dots, n-2 \; ; \; D_n = 2 \; , \tag{20}$$

the elements Q_i at the above-main diagonal are

$$Q_1 = 1 \; ; \; Q_{i+1} = \frac{h_i}{h_{i+1}} \text{ for } i = 1, \dots, n-2 \; , \tag{21}$$

and the right-hand side terms B_i are

$$B_1 = 3p_1 \; ; \; B_{i+1} = 3\left(p_i + p_{i+1}\frac{h_i}{h_{i+1}}\right) \text{ for } i = 1, \dots, n-2 \; ; B_n = 3p_{n-1} \; . \tag{22}$$

This system can be solved by one run of Gauss elimination and then a run of back-substitution. After a run of Gaussian elimination the system becomes

$$\begin{pmatrix} \tilde{D}_1 & Q_1 & 0 & 0 & \dots \\ 0 & \tilde{D}_2 & Q_2 & 0 & \dots \\ 0 & 0 & \tilde{D}_3 & Q_3 & \dots \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ \dots & \dots & 0 & 0 & \tilde{D}_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} \tilde{B}_1 \\ \vdots \\ \vdots \\ \tilde{B}_n \end{pmatrix} \; , \tag{23}$$

where

$$\tilde{D}_1 = D_1 \; ; \; \tilde{D}_i = D_i - Q_{i-1}/\tilde{D}_{i-1} \; , \; i = 2, \dots, n \tag{24}$$

and

$$\tilde{B}_1 = B_1 \; ; \; \tilde{B}_i = B_i - \tilde{B}_{i-1}/\tilde{D}_{i-1} \; , \; i = 2, \dots, n \tag{25}$$

The triangular system (23) can be solved by a run of back-substitution,

$$b_n = \frac{1}{\tilde{D}_n}\tilde{B}_n \; ; \; b_i = \frac{1}{\tilde{D}_i}(\tilde{B}_i - Q_i b_{i+1}) \; , \; i = n-1, \dots, 1 \; . \tag{26}$$

Other forms of interpolation

Other forms of interpolation can be constructed by choosing a different class of interpolating functions, for example, rational function interpolation, trigonometric interpolation, wavelet interpolation etc.

Sometimes not only the value of the function is given at the tabulated points, but also the derivative. This extra information can be taken advantage of when constructing the interpolation function.

Interpolation of a function in more than one dimension is called *multivariate interpolation.* In two dimension one of the easiest methods is the *bi-linear interpolation* where the function in each tabulated rectangle is approximated as a product of two linear functions,

$$f(x, y) \approx (ax + b)(cy + d) \; , \tag{27}$$

where the constants a, b, c, d are obtained from the condition that the interpolating function is equal the tabulated values at the nearest four tabulated points.

Table 3: Cubic spline in C++, object-oriented style

```
#include<vector>
#include<stdexcept>
using namespace std;

struct cspline
{
int n; vector<double> x,y,b,c,d;
cspline(vector<double>&x,vector<double>&y);
double eval(double z);
};

cspline::cspline(vector<double>&x,vector<double>&y)
: n(x.size()), x(x), y(y), b(n), c(n-1), d(n-1)
{
vector<double> D(n), Q(n-1), B(n), h(n-1), p(n-1);
for(int i=0;i<n-1;i++) { h[i]=x[i+1]-x[i]; p[i]=(y[i+1]-y[i])/h[i]; }

D[0]=2; Q[0]=1; B[0]=3*p[0]; D[n-1]=2; B[n-1]=3*p[n-2];
for(int i=0;i<n-2;i++){
   D[i+1]=2*h[i]/h[i+1]+2;
   Q[i+1]=h[i]/h[i+1];
   B[i+1]=3*(p[i]+p[i+1]*h[i]/h[i+1]);}

for(int i=1;i<n;i++){ // Gauss elimination
   D[i]-=Q[i-1]/D[i-1]; B[i]-=B[i-1]/D[i-1]; }

b[n-1]=B[n-1]/D[n-1]; // back-substitution
for(int i=n-2;i>=0;i--) b[i]=(B[i]-Q[i]*b[i+1])/D[i];

for(int i=0;i<n-1;i++){
   c[i]=(-2*b[i]-b[i+1]+3*p[i])/h[i];
   d[i]=(b[i]+b[i+1]-2*p[i])/h[i]/h[i];}
}

double cspline::eval(double z) // evaluation of spline
{
if(z<x[0] || z>x[n-1]) throw logic_error("eval: out of range");
int i=0, j=n-1; // binary search for the interval for z :
while(j-i>1){ int m=(i+j)/2; if(z>x[m]) i=m; else j=m; }
double h=z-x[i]; // inerpolating spline :
return y[i]+h*(b[i]+h*(c[i]+h*d[i]));
}
```