1 Interpolation

In practice one often meets a situation where a function of interest is only given as a set of tabulated discrete points $\{x_i, y_i\}$, $i = 1 \ldots n$, for example as a result of a numerical integration of a differential equation. *Interpolation* means constructing a (smooth) function, called interpolating function, which passes exactly through the given points and hopefully approximates the unknown function of interest in between the tabulated points. One can use the interpolating function for different practical needs, like estimating the unknown function between the tabulated points, differentiating, integrating etc. Interpolation is a specific case of curve fitting, in which the function must go exactly through the data points.

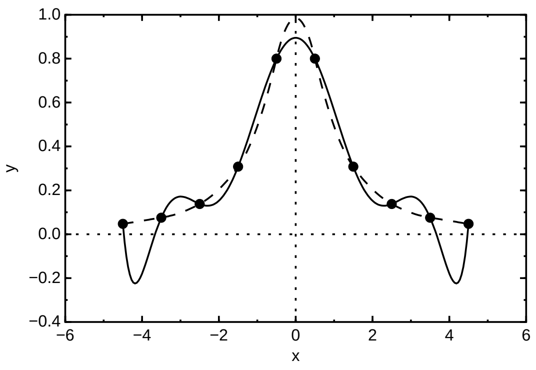

Figure 1: Lagrange interpolating polynomial (solid line) showing the Runge phenomenon: large oscillations at the end-points. Dashed line shows a quadratic spline.

1.1 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 of the order $n-1$ which passes exactly through the points. This polynomial can be intuitively written in the *Lagrange form*,

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

```
function pinterp(x,y,z){
    for(var s=0,i=0; i<x.length; i++){
        for(var p=1,k=0; k<x.length; k++){
            if(k!=i) p*=(z-x[k])/(x[i]-x[k])}
        s+=y[i]*p}
    return s
}
```

Higher order interpolating polynomials, say larger than 5, are susceptible to the *Runge phenomenon*: erratic oscillations close to the end-points of the interval, as illustrated on Figure 1. Therefore when interpolating from a large table one usually uses only the nearest few points instead of all the points in the table.

1.2 Spline interpolation

Spline interpolation uses a *piecewise polynomial* (called *spline*) as the interpolating function: at each interval $[x_i, x_{i+1}]$ the spline, $S(x)$, is represented by a polynomial

$$S_i(x) = \sum_{p=0}^{k} c_{ip} x^p, \; i = 1 \ldots n-1 \tag{2}$$

of a given order k, such that

$$S(x) = S_i(x), \text{ if } x \in [x_i, x_{i+1}] \; . \tag{3}$$

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

$$\begin{aligned} S_i(x_i) &= y_i \quad &, \; i = 1 \ldots n-1 \\ S_i(x_{i+1}) &= y_{i+1} \quad &, \; i = 1 \ldots n-1 \; , \end{aligned} \tag{4}$$

together with its $k-1$ derivatives,

$$\begin{aligned} S_i'(x_{i+1}) &= S_{i+1}'(x_{i+1}) \quad &, \; i = 1 \ldots n-2 \\ S_i''(x_{i+1}) &= S_{i+1}''(x_{i+1}) \quad &, \; i = 1 \ldots n-2 \\ &\ldots \end{aligned} \tag{5}$$

Continuity conditions (4) and (5) make $kn+n-2k$ linear equations for the $kn+n-k-1$ coefficients c_{ik} in (2). The missing $k-1$ equations can be chosen (reasonably) arbitrary.

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 has also a nice feature that it (sort of) minimises the total curvature of the interpolating function. This makes the cubic splines look good.

Quadratic splines are not nearly as good as cubic splines in most respects. Particularly they 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. However quadratic spline is simpler to program.

1.2.1 Linear interpolation

If the spline polynomials are linear the spline is called *linear interpolation.* The continuity conditions (4) can be satisfied by choosing the spline as

$$S_i(x) = y_i + \frac{\Delta y_i}{\Delta x_i}(x - x_i) \; . \tag{6}$$

where

$$\Delta y_i \equiv y_{i+1} - y_i \ , \ \ \Delta x_i \equiv x_{i+1} - x_i \ . \tag{7}$$

1.2.2 Quadratic spline

Quadratic splines are made of second order polynomials, conveniently chosen in the form

$$S_i(x) = y_i + \frac{\Delta y_i}{\Delta x_i}(x - x_i) + a_i(x - x_i)(x - x_{i+1}), \tag{8}$$

which identically satisfies the continuity conditions (4).

Substituting (8) into the continuity condition (5) for the first derivative gives the equations for the coefficient a_i,

$$\frac{\Delta y_i}{\Delta x_i} + a_i \Delta x_i = \frac{\Delta y_{i+1}}{\Delta x_{i+1}} - a_{i+1}\Delta x_{i+1} \ . \tag{9}$$

For the quadratic spline, $k = 2$, one coefficient can be chosen arbitrary, for example $a_1 = 0$. Now the other coefficients can be calculated recursively,

$$a_{i+1} = \frac{1}{\Delta x_{i+1}} \left(\frac{\Delta y_{i+1}}{\Delta x_{i+1}} - \frac{\Delta y_i}{\Delta x_i} - a_i \Delta x_i \right) . \tag{10}$$

Alternatively, one can choose $a_{n-1} = 0$ and make an inverse recursion

$$a_i = \frac{1}{\Delta x_i} \left(\frac{\Delta y_{i+1}}{\Delta x_{i+1}} - \frac{\Delta y_i}{\Delta x_i} - a_{i+1} \Delta x_{i+1} \right) . \tag{11}$$

In practice it is better (for symmetry reasons) to run both recursions and then average the resulting a's.

1.2.3 JavaScript implementation

```
Array.prototype.__iterator__=function()
  {for(let i=0;i<this.length;i++) yield i;}

function qspline(x,y){
//creates a quadratic spline
var n=x.length;
var h=[(x[i+1]-x[i])for(i in x)if(i<n-1)];
var p=[(y[i+1]-y[i])/h[i] for(i in h)];
var a=new Array(n-1);

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

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

this.x=[x[i] for(i in x)];
this.y=[y[i] for(i in y)];
this.p=p;
this.a=a;

//evaluation of the spline at point z:
this.eval=function(z){
  var n=this.x.length;
  if(z<this.x[0] || z>this.x[n-1])
    throw "qspline.eval: out of range";
  var i=0, j=n-1;
//locate the interval for z by bisection:
  while(j-i>1){
    var mid=Math.round( (i+j)/2 );
    if(z>this.x[mid]) i=mid; else j=mid;}
//calculate the inerpolating polynomial:
  return(
    this.y[i]+
    (this.p[i]+this.a[i]*(z-this.x[i+1]))*(
       z-this.x[i]));
  }// end eval
}// end qspline
```

1.3 Integration and differentiation of tabulated functions

Hint: first interpolate the tabulated data and then integrate or differentiate the interpolating function.