1 Interpolation

1.1 Introduction

In practice one often meets a situation where the function of interest, $f(x)$, is only represented by a discrete set of tabulated points,

$$\{x_i, y_i \doteq f(x_i) \mid i = 1 \dots n\},$$

obtained, for example, by sampling, experimentation, or extensive numerical calculations.

Interpolation means constructing a (smooth) function, called *interpolating function* or *interpolant*, which passes exactly through the given points and hopefully approximates the tabulated function 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/integrals involving the tabulated function.

1.2 Polynomial interpolation

Polynomial interpolation uses a polynomial as the interpolating function. Given a table of n points, $\{x_i, y_i\}$, where no two x_i are the same, one can construct a polynomial $P(x)$ of the order $n-1$ which passes exactly through the points: $P(x_i) = y_i$. This polynomial can be intuitively written in the *Lagrange form*,

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

The Lagrange interpolating polynomial always exists and is unique.

Table 1: Polynomial interpolation in C

```
double polinterp(int n, double *x, double *y, double z) {
   double s=0,p;
   for(int i=0;i<n;i++) {
      p=1; for(int k=0;k<n;k++) if(k!=i) p*=(z-x[k])/(x[i]-x[k]);
      s+=y[i]*p; }
   return s; }
```

Higher order interpolating polynomials are susceptible to the *Runge's phenomenon:* erratic oscillations close to the end-points of the interval, see Figure 1.

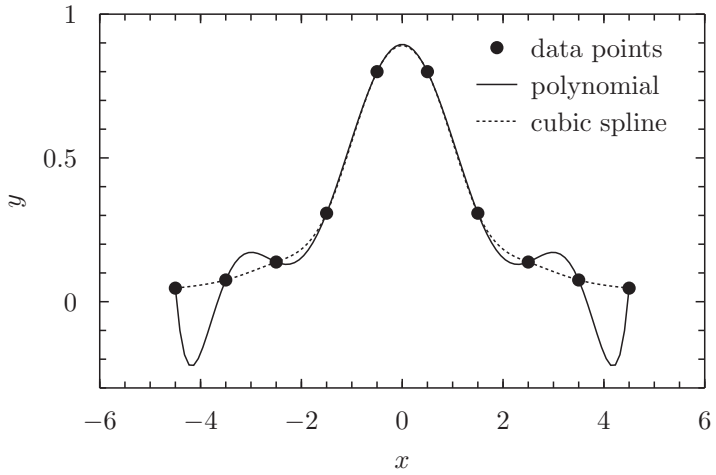


Figure 1: Lagrange interpolating polynomial, solid line, showing the Runge's phenomenon: large oscillations at the edges. For comparison the dashed line shows a cubic spline.

1.3 Spline interpolation

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

$$S(x) = s_i(x) \text{ if } x \in [x_i, x_{i+1}] \Big|_{i=1,\ldots,n-1}, \tag{2}$$

where $s_i(x)$ is a polynomial of a given order k. Spline interpolation avoids the problem of Runge's phenomenon. Originally, "spline" was a term for elastic rulers that were bent to pass through a number of predefined points. These were used to make technical drawings by hand.

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

$$s_i(x_i) = y_i\,,\ s_i(x_{i+1}) = y_{i+1} \Big|_{i=1,\ldots,n-1}, \tag{3}$$

together with its $k-1$ derivatives,

$$\left.\begin{array}{r} s_i'(x_{i+1}) = s_{i+1}'(x_{i+1})\,, \\ s_i''(x_{i+1}) = s_{i+1}''(x_{i+1})\,, \\ \vdots \\ s_i^{(k-1)}(x_{i+1}) = s_{i+1}^{(k-1)}(x_{i+1})\,. \end{array}\right| \; i = 1, \ldots, n-2 \tag{4}$$

Continuity conditions (3) and (4) make $kn + n - 2k$ linear equations for the $(n-1)(k+1) = kn + n - k - 1$ coefficients of $n-1$ 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 has a nice feature that it (sort of) minimizes the total curvature of the interpolating function. This makes the cubic splines look good.

Quadratic splines—continuous with the first derivative—are not nearly as good as cubic splines in most respects. In particular they might oscillate unpleasantly when a quick change in the tabulated function is followed by a period where the function is nearly a constant. Cubic splines are somewhat less susceptible to such oscillations.

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

1.3.1 Linear interpolation

Linear interpolation is a spline with linear polynomials. The continuity conditions (3) can be satisfied by choosing the spline in the (intuitive) form

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

where

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

The linear spline can be easily differentiated,

$$s_i'(x) = p_i \ , \tag{7}$$

and integrated,

$$\int_{x_i}^{x<x_{i+1}} s_i(t)dt = y_i(x - x_i) + p_i \frac{(x - x_i)^2}{2} \ . \tag{8}$$

Linear spline is continuous but its derivative is not.

Table 2: Linear interpolation in C

```
#include<assert.h>
double linterp(int n, double* x, double* y, double z){
    assert(n>1 && z>=x[0] && z<=x[n-1]);
    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;}
    double dy=y[i+1]-y[i], dx=x[i+1]-x[i]; assert(dx>0);
    return y[i]+dy/dx*(z-x[i]);
}
```

Note that the search of the interval $[x_i \le x \le x_{i+1}]$ in an ordered array $\{x_i\}$ should be done with the *binary search* algorithm (also called *half-interval search*): the point x is compared to the middle element of the array, if it is less than the middle element, the algorithm repeats its action on the sub-array to the left of the

middle element, if it is greater, on the sub-array to the right. When the remaining sub-array is reduced to two elements, the interval is found (see Table 2). The average number of operations for a binary search is $O(\log n)$.

1.3.2 Quadratic spline

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

$$s_i(x) = y_i + p_i(x - x_i) + c_i(x - x_i)(x - x_{i+1}) \Big|_{i=1,\ldots,n-1} , \tag{9}$$

which identically satisfies the continuity conditions

$$s_i(x_i) = y_i \ , \ s_i(x_{i+1}) = y_{i+1} \Big|_{i=1,\ldots,n-1} . \tag{10}$$

Substituting (9) into the derivative continuity condition,

$$s_i'(x_{i+1}) = s_{i+1}'(x_{i+1}) \Big|_{i=1,\ldots,n-2} , \tag{11}$$

gives $n-2$ equations for $n-1$ unknown coefficients c_i,

$$p_i + c_i \Delta x_i = p_{i+1} - c_{i+1} \Delta x_{i+1} \Big|_{i=1,\ldots,n-2} . \tag{12}$$

One coefficient can be chosen arbitrarily, for example $c_1 = 0$. The other coefficients can now be calculated recursively from (12),

$$c_{i+1} = \frac{1}{\Delta x_{i+1}} \left(p_{i+1} - p_i - c_i \Delta x_i \right) \Big|_{i=1,\ldots,n-2} . \tag{13}$$

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

$$c_i = \frac{1}{\Delta x_i} \left(p_{i+1} - p_i - c_{i+1} \Delta x_{i+1} \right) \Big|_{i=n-2,\ldots,1} . \tag{14}$$

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. This amounts to first running the forward-recursion from $c_1 = 0$ and then the backward recursion from $\frac{1}{2} c_{n-1}$.

The optimized form (9) of the quadratic spline can also be written in the ordinary form suitable for differentiation and integration,

$$s_i(x) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 \ , \text{ where } b_i = p_i - c_i \Delta x_i \ . \tag{15}$$

An implementation of quadratic spline in C is listed in Table 1.3.2

Table 3: Quadratic spline in C

```
#include <stdlib.h>
#include <assert.h>
typedef struct {int n; double *x, *y, *b, *c;} qspline;
qspline* qspline_alloc(int n,double* x,double* y){ //builds qspline
    qspline *s = (qspline*)malloc(sizeof(qspline));//spline
    s->b = (double*)malloc((n-1)*sizeof(double));   // b_i
    s->c = (double*)malloc((n-1)*sizeof(double));   // c_i
    s->x = (double*)malloc(n*sizeof(double));       // x_i
    s->y = (double*)malloc(n*sizeof(double));       // y_i
    s->n = n; for(int i=0;i<n;i++){s->x[i]=x[i]; s->y[i]=y[i];}
    int i; double p[n-1], h[n-1];                   //VLA from C99
    for(i=0;i<n-1;i++){h[i]=x[i+1]-x[i]; p[i]=(y[i+1]-y[i])/h[i];}
    s->c[0]=0;                                      //recursion up:
    for(i=0;i<n-2;i++)s->c[i+1]=(p[i+1]-p[i]-s->c[i]*h[i])/h[i+1];
    s->c[n-2]/=2;                                  //recursion down:
    for(i=n-3;i>=0;i--)s->c[i]=(p[i+1]-p[i]-s->c[i+1]*h[i+1])/h[i];
    for(i=0;i<n-1;i++)s->b[i]=p[i]-s->c[i]*h[i];
    return s; }
double qspline_eval(qspline *s, double z){      //evaluates s(z)
    assert(z>=s->x[0] && z<=s->x[s->n-1]);
    int i=0, j=s->n-1;                          //binary search:
    while(j-i>1){int m=(i+j)/2; if(z>s->x[m]) i=m; else j=m;}
    double h=z-s->x[i];
    return s->y[i]+h*(s->b[i]+h*s->c[i]); }//inerpolating polynomial
void qspline_free(qspline *s){ //free the allocated memory
    free(s->x); free(s->y); free(s->b); free(s->c); free(s); }
```

1.3.3 Cubic spline

Cubic splines are made of third order polynomials,

$$s_i(x) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \ . \tag{16}$$

This form automatically satisfies the first half of continuity conditions (3): $s_i(x_i) = y_i$. The second half of continuity conditions (3), $s_i(x_{i+1}) = y_{i+1}$, and the continuity of the first and second derivatives (4) give a set of equations,

$$\begin{aligned} y_i + b_i h_i + c_i h_i^2 + d_i h_i^3 &= y_{i+1} \ , \ i = 1, \ldots, n-1 \\ b_i + 2c_i h_i + 3d_i h_i^2 &= b_{i+1} \ , \ i = 1, \ldots, n-2 \\ 2c_i + 6d_i h_i &= 2c_{i+1} \ , \ i = 1, \ldots, n-2 \end{aligned} \tag{17}$$

where $h_i \doteq x_{i+1} - x_i$.

The set of equations (17) 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

derivatives at the end-points,

$$S''(x_1) = S''(x_n) = 0 \ , \tag{18}$$

which gives

$$\begin{aligned} c_1 &= 0 \ , \\ c_{n-1} + 3d_{n-1}h_{n-1} &= 0 \ . \end{aligned} \tag{19}$$

Solving the first two equations in (17) for c_i and d_i gives[1]

$$\begin{aligned} c_i h_i &= -2b_i - b_{i+1} + 3p_i \ , \\ d_i h_i^2 &= b_i + b_{i+1} - 2p_i \ , \end{aligned} \tag{20}$$

where $p_i \doteq \Delta y_i / h_i$. The natural conditions (19) and the third equation in (17) then produce the following tridiagonal system of n linear equations for the n coefficients b_i,

$$\begin{aligned} 2b_1 + b_2 &= 3p_1 \ , \\ b_i + \left(2\frac{h_i}{h_{i+1}} + 2\right) b_{i+1} + \frac{h_i}{h_{i+1}} b_{i+2} &= 3\left(p_i + p_{i+1}\frac{h_i}{h_{i+1}}\right) \Big|_{i=1,\ldots,n-2} \ , \\ b_{n-1} + 2b_n &= 3p_{n-1} \ , \end{aligned} \tag{21}$$

or, in the matrix form,

$$\begin{pmatrix} D_1 & Q_1 & 0 & 0 & \ldots \\ 1 & D_2 & Q_2 & 0 & \ldots \\ 0 & 1 & D_3 & Q_3 & \ldots \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ \ldots & \ldots & 0 & 1 & D_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} B_1 \\ \vdots \\ \vdots \\ \vdots \\ B_n \end{pmatrix} \tag{22}$$

where the elements D_i at the main diagonal are

$$D_1 = 2 \ , \ D_{i+1} = 2\frac{h_i}{h_{i+1}} + 2 \Big|_{i=1,\ldots,n-2} \ , \ D_n = 2 \ , \tag{23}$$

the elements Q_i at the above-main diagonal are

$$Q_1 = 1 \ , \ Q_{i+1} = \frac{h_i}{h_{i+1}} \Big|_{i=1,\ldots,n-2} \ , \tag{24}$$

and the right-hand side terms B_i are

$$B_1 = 3p_1 \ , \ B_{i+1} = 3\left(p_i + p_{i+1}\frac{h_i}{h_{i+1}}\right) \Big|_{i=1,\ldots,n-2} , B_n = 3p_{n-1} \ . \tag{25}$$

[1] introducing an auxiliary coefficient b_n.

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

$$\begin{pmatrix} \tilde{D}_1 & Q_1 & 0 & 0 & \dots \\ 0 & \tilde{D}_2 & Q_2 & 0 & \dots \\ 0 & 0 & \tilde{D}_3 & Q_3 & \dots \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ \dots & \dots & 0 & 0 & \tilde{D}_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} \tilde{B}_1 \\ \vdots \\ \vdots \\ \tilde{B}_n \end{pmatrix} , \tag{26}$$

where

$$\tilde{D}_1 = D_1 \ , \ \tilde{D}_i = D_i - Q_{i-1}/\tilde{D}_{i-1} \ \Big|_{i=2,\dots,n} , \tag{27}$$

and

$$\tilde{B}_1 = B_1 \ , \ \tilde{B}_i = B_i - \tilde{B}_{i-1}/\tilde{D}_{i-1} \ \Big|_{i=2,\dots,n} . \tag{28}$$

The triangular system (26) can be solved by a run of back-substitution,

$$b_n = \tilde{B}_n/\tilde{D}_n \ , \ b_i = (\tilde{B}_i - Q_i b_{i+1})/\tilde{D}_i \ \Big|_{i=n-1,\dots,1} . \tag{29}$$

A C-implementation of cubic spline is listed in Table 1.3.3

1.3.4 Sub-splines

Sub-splines are—like splines—piecewise polynomials. However, unlike splines the sub-splines dispense with the demand of maximal differentiability of the spline—hence the name. Instead of maximal differentiability the sub-splines use one (or more) of their free parameters to achieve some other goals. Typically the sub-splines try to minimize the unpleasant wiggles that splines are prone to when the interpolated data contain a sub-set of points that are close to a line or an outlier.

Akima sub-spline [**?**] is an interpolating function in the form of a piecewise cubic polynomial, similar to the cubic spline,

$$S(x)\Big|_{x\in[x_i,x_{i+1}]} = a_i + b_i(x-x_i) + c_i(x-x_i)^2 + d_i(x-x_i)^3 \doteq A_i(x) \, . \tag{30}$$

However, unlike the cubic spline, Akima sub-spline dispenses with the demand of maximal differentiability of the spline—in this case, the continuity of the second derivative—hence the name *sub-spline*. Instead of achieving maximal differentiability Akima sub-splines try to reduce the wiggling which the ordinary splines are typically prone to (see Figure 2).

First let us note that the coefficients $\{a_i, b_i, c_i, d_i\}$ in eq. (30) are determined by the values of the derivatives $S'_i \doteq S'(x_i)$ of the sub-spline through the continuity conditions for the sub-spline and its first derivative,

$$A_i(x_i) = y_i, \ A'_i(x_i) = S'_i, \ A_i(x_{i+1}) = y_{i+1}, \ A'_i(x_{i+1}) = S'_{i+1}. \tag{31}$$

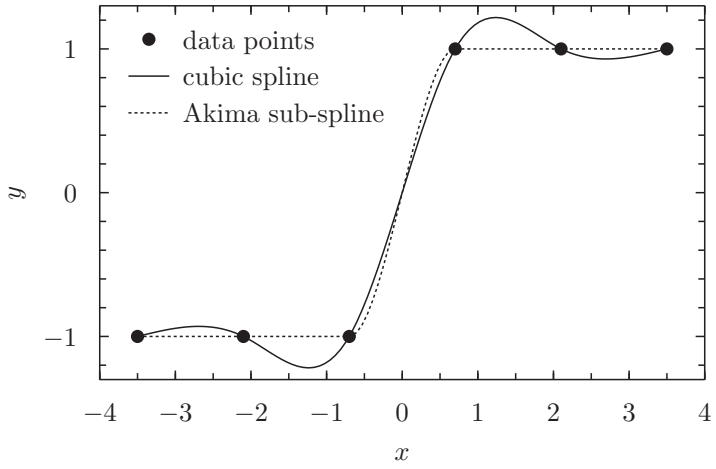


Figure 2: A cubic spline (solid line) showing the typical wiggles, compared to the Akima sub-spline (dashed line) where the wiggles are essentially removed.

Indeed, inserting (30) into (31) and solving for the coefficients gives

$$a_i = y_i,\ b_i = S'_i,\ c_i = \frac{3p_i - 2S'_i - S'_{i+1}}{\Delta x_i},\ d_i = \frac{S'_i + S'_{i+1} - 2p_i}{(\Delta x_i)^2}, \tag{32}$$

where $p_i \doteq \Delta y_i/\Delta x_i$, $\Delta y_i \doteq y_{i+1} - y_i$, $\Delta x_i \doteq x_{i+1} - x_i$.

In the ordinary cubic spline the derivatives S'_i are determined by the continuity condition of the second derivative of the spline. Sub-splines do without this continuity condition and can instead use the derivatives as free parameters to be chosen to satisfy some other condition.

Akima suggested to minimize the wiggling by choosing the derivatives as linear combinations of the nearest slopes,

$$S'_i = \frac{w_{i+1}p_{i-1} + w_{i-1}p_i}{w_{i+1} + w_{i-1}}\,, \quad \text{if} \quad w_{i+1} + w_{i-1} \neq 0\,, \tag{33}$$

$$S'_i = \frac{p_{i-1} + p_i}{2}\,, \quad \text{if} \quad w_{i+1} + w_{i-1} = 0\,, \tag{34}$$

where the weights w_i are given as

$$w_i = |p_i - p_{i-1}|\,. \tag{35}$$

The idea is that if three points lie close to a line, the sub-spline in this vicinity has to be close to this line. In other words, if $|p_i - p_{i-1}|$ is small, the nearby derivatives must be close to p_i.

The first two and the last two points need a special prescription, for example (naively) one can simply use

$$S'_1 = p_1,\ S'_2 = \frac{1}{2}p_1 + \frac{1}{2}p_2,\ S'_n = p_{n-1},\ S'_{n-1} = \frac{1}{2}p_{n-1} + \frac{1}{2}p_{n-2}. \tag{36}$$

Table (5) shows a C-implementation of this algorithm.

1.4 Rational function interpolation

As the name suggests, the rational interpolation uses a rational function (a ratio of two polynomials) as the interpolant,

$$r_m^k(x) = \frac{p_0 + p_1 x + \cdots + p_k x^k}{q_0 + q_1 x + \cdots + q_m x^m} \ , \tag{37}$$

where $m > 0$. The rational interpolants are (generally) not susceptible to Runge phenomenon and are infinitely differentiable.

One popular family of rational function interpolants is the so called *univariate barycentric interpolation*. One example of this can be illustrated as follows. Suppose the table to interpolate contains only two points, (x_0, y_0) and (x_1, y_1). Then the (linear) interpolant can be intuitively written as

$$F(x) = \frac{x_1 - x}{x_1 - x_0} y_0 + \frac{x - x_0}{x_1 - x_0} y_1 \ . \tag{38}$$

In order to generalize this approach to larger tables one can rewrite it as

$$F(x) = \frac{(x - x_1)y_0 - (x - x_0)y_1}{-(x_1 - x_0)} = \frac{(x - x_1)y_0 - (x - x_0)y_1}{(x - x_1) - (x - x_0)} \ . \tag{39}$$

Now we divide both the numerator and denominator by $(x - x_0)(x - x_1)$,

$$F(x) = \frac{\dfrac{1}{x - x_0} y_0 - \dfrac{1}{x - x_1} y_1}{\dfrac{1}{x - x_0} - \dfrac{1}{x - x_1}} = \frac{\displaystyle\sum_{i=0}^{1} \frac{(-1)^i}{x - x_i} y_i}{\displaystyle\sum_{i=0}^{1} \frac{(-1)^i}{x - x_i}} \tag{40}$$

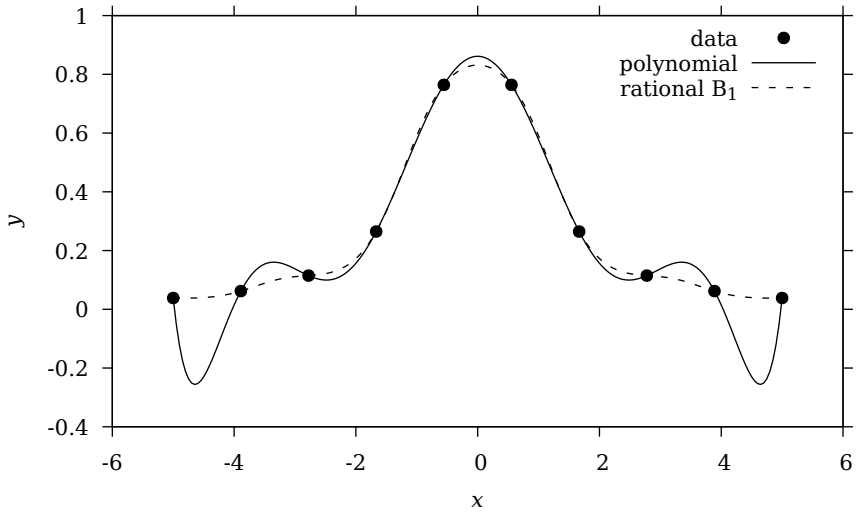
Now, the generalization to a table with $n+1$ points seems as easy as changing the summation limit from 1 to n (first suggested by Berrut in 1988 [**?**]),

$$B_1(x) = \frac{\displaystyle\sum_{i=0}^{n} \frac{(-1)^i}{x - x_i} y_i}{\displaystyle\sum_{i=0}^{n} \frac{(-1)^i}{x - x_i}} \ . \tag{41}$$

In order to see that B_1 indeed goes through the tabulated points we can calculate the limit,

$$\lim_{x \to x_i} B_1(x) = \frac{\dfrac{(-1)^i}{x - x_i} y_i}{\dfrac{(-1)^i}{x - x_i}} = y_i \ . \tag{42}$$

Figure 3: Polynomial interpolant showing the Runge's phenomenon (large oscillations at the edges) compared to rational Berrut interpolant where the oscillations are significantly reduced.



One can show that $B_1(x)$ is indeed a rational function (of the order of at most n by n) by multiplying both the numerator and the denominator by $\prod_i (x - x_i)$. One can also show that $B_1(x)$ has no poles on the real axis as the denominator never vanishes for real arguments. Indeed suppose that $x \in [x_0, x_1]$. The denominator is then given as

$$\sum_{i=0}^{n} \frac{(-1)^i}{x - x_i} = \left(\frac{1}{x - x_0}\right) + \left(\frac{1}{x_1 - x} - \frac{1}{x_2 - x}\right) + \left(\frac{1}{x_3 - x} - \dots\right) + \dots \qquad (43)$$

which is always larger than zero since every term in parentheses is larger than zero (the proof is similar if x falls in any other sub-interval).

Berrut has also suggested a slightly different rational interpolant,

$$B_2(x) = \frac{\dfrac{1}{x - x_0} y_0 + 2 \sum_{i=1}^{n-1} \dfrac{(-1)^i}{x - x_i} y_i + \dfrac{(-1)^n}{x - x_n} y_n}{\dfrac{1}{x - x_0} + 2 \sum_{i=1}^{n-1} \dfrac{(-1)^i}{x - x_i} + \dfrac{(-1)^n}{x - x_n}} \,. \qquad (44)$$

Berrut interpolants are as slow to evaluate as the Lagrange interpolating polynomial—$O(n)$ operations—but are more stable, see the illustration on Figure (3).

1.5 Multivariate interpolation

Interpolation of a function in more than one variable is called *multivariate interpolation*. The function of interest is represented as a set of discrete points in a multidimensional space. The points may or may not lie on a regular grid.

1.5.1 Nearest-neighbor interpolation

Nearest-neighbor interpolation approximates the value of the function at a non-tabulated point by the value at the nearest tabulated point, yielding a piecewise-constant interpolating function. It can be used for both regular and irregular grids.

1.5.2 Piecewise-linear interpolation

Piecewise-linear interpolation is used to interpolate functions of two variables tabulated on irregular grids. The tabulated 2D region is triangulated – subdivided into a set of non-intersecting triangles whose union is the original region. Inside each triangle the interpolating function $S(x, y)$ is taken in the linear form,

$$S(x, y) = a + bx + cy \ , \tag{45}$$

where the three constants are determined by the three conditions that the interpolating function is equal the tabulated values at the three vertexes of the triangle.

1.5.3 Bi-linear interpolation

Bi-linear interpolation is used to interpolate functions of two variables tabulated on regular rectilinear 2D grids. The interpolating function $B(x, y)$ inside each of the grid rectangles is taken as a bilinear function of x and y,

$$B(x, y) = a + bx + cy + dxy \ , \tag{46}$$

where the four constants a, b, c, d are obtained from the four conditions that the interpolating function is equal the tabulated values at the four nearest tabulated points (which are the vertexes of the given grid rectangle).

Table 4: Cubic spline in C

```c
#include<stdlib.h>
#include<assert.h>
#include<stdio.h>
typedef struct {int n; double *x,*y,*b,*c,*d;} cubic_spline;
cubic_spline* cubic_spline_alloc(int n, double *x, double *y)
{// builds natural cubic spline
    cubic_spline* s = (cubic_spline*)malloc(sizeof(cubic_spline));
    s->x = (double*)malloc(n*sizeof(double));
    s->y = (double*)malloc(n*sizeof(double));
    s->b = (double*)malloc(n*sizeof(double));
    s->c = (double*)malloc((n-1)*sizeof(double));
    s->d = (double*)malloc((n-1)*sizeof(double));
    s->n = n; for(int i=0;i<n;i++){s->x[i]=x[i]; s->y[i]=y[i];}
    double h[n-1],p[n-1]; // VLA
    for(int i=0;i<n-1;i++){h[i]=x[i+1]-x[i]; assert(h[i]>0);}
    for(int i=0;i<n-1;i++) p[i]=(y[i+1]-y[i])/h[i];
    double D[n], Q[n-1], B[n];// building the tridiagonal system:
    D[0]=2; for(int i=0;i<n-2;i++)D[i+1]=2*h[i]/h[i+1]+2; D[n-1]=2;
    Q[0]=1; for(int i=0;i<n-2;i++)Q[i+1]=h[i]/h[i+1];
    for(int i=0;i<n-2;i++)B[i+1]=3*(p[i]+p[i+1]*h[i]/h[i+1]);
    B[0]=3*p[0]; B[n-1]=3*p[n-2]; //Gauss elimination :
    for(int i=1;i<n;i++){ D[i]-=Q[i-1]/D[i-1]; B[i]-=B[i-1]/D[i-1]; }
    s->b[n-1]=B[n-1]/D[n-1]; //back-substitution :
    for(int i=n-2;i>=0;i--) s->b[i]=(B[i]-Q[i]*s->b[i+1])/D[i];
    for(int i=0;i<n-1;i++){
        s->c[i]=(-2*s->b[i]-s->b[i+1]+3*p[i])/h[i];
        s->d[i]=(s->b[i]+s->b[i+1]-2*p[i])/h[i]/h[i];
    }
    return s;
}
double cubic_spline_eval(cubic_spline *s,double z){
        assert(z>=s->x[0] && z<=s->x[s->n-1]);
        int i=0, j=s->n-1;// binary search for the interval for z :
        while(j-i>1){int m=(i+j)/2; if(z>s->x[m]) i=m; else j=m; }
        double h=z-s->x[i];// calculate the inerpolating spline :
        return s->y[i]+h*(s->b[i]+h*(s->c[i]+h*s->d[i]));
}
void cubic_spline_free(cubic_spline *s){ //free the allocated memory
    free(s->x);free(s->y);free(s->b);free(s->c);free(s->d);free(s);}
```

Table 5: Akima sub-spline in C

```c
#include<assert.h>
#include<stdlib.h>
#include<math.h>
typedef struct {int n; double *x,*y,*b,*c,*d;} akima_spline;
akima_spline* akima_spline_alloc(int n, double *x, double *y){
    assert(n>2); double h[n-1],p[n-1]; /* VLA */
    for(int i=0;i<n-1;i++){h[i]=x[i+1]-x[i]; assert(h[i]>0);}
    for(int i=0;i<n-1;i++) p[i]=(y[i+1]-y[i])/h[i];
    akima_spline *s = (akima_spline*)malloc(sizeof(akima_spline));
    s->x = (double*)malloc(n*sizeof(double));
    s->y = (double*)malloc(n*sizeof(double));
    s->b = (double*)malloc(n*sizeof(double));
    s->c = (double*)malloc((n-1)*sizeof(double));
    s->d = (double*)malloc((n-1)*sizeof(double));
    s->n = n; for(int i=0;i<n;i++){s->x[i]=x[i]; s->y[i]=y[i];}
    s->b[0]  =p[0];    s->b[1]  =(p[0]+p[1])/2;
    s->b[n-1]=p[n-2]; s->b[n-2]=(p[n-2]+p[n-3])/2;
    for(int i=2;i<n-2;i++){
        double w1=fabs(p[i+1]-p[i]), w2=fabs(p[i-1]-p[i-2]);
        if(w1+w2==0) s->b[i]=(p[i-1]+p[i])/2;
        else s->b[i]=(w1*p[i-1]+w2*p[i])/(w1+w2);
    }
    for(int i=0;i<n-1;i++){
        s->c[i]=(3*p[i]-2*s->b[i]-s->b[i+1])/h[i];
        s->d[i]=(s->b[i+1]+s->b[i]-2*p[i])/h[i]/h[i];
    }
    return s;
}
double akima_spline_eval(akima_spline *s, double z){
    assert(z>=s->x[0] && z<=s->x[s->n-1]);
    int i=0, j=s->n-1;
    while(j-i>1){int m=(i+j)/2; if(z>s->x[m]) i=m; else j=m;}
    double h=z-s->x[i];
    return s->y[i]+h*(s->b[i]+h*(s->c[i]+h*s->d[i]));
}
void akima_spline_free(akima_spline *s){
    free(s->x);free(s->y);free(s->b);free(s->c);free(s->d);free(s);}
```