Interpolation

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,

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

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 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, $\{x_i, y_i\}$, 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 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;
  for(int i=0;i<n;i++) {
    double 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).

Spline interpolation

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

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

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,

$$S_i(x_i) = y_i \text{ , } S_i(x_{i+1}) = y_{i+1} \text{ , } i = 1, \dots, n-1 \text{ ,} \tag{3}$$

together with its $k-1$ derivatives,

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

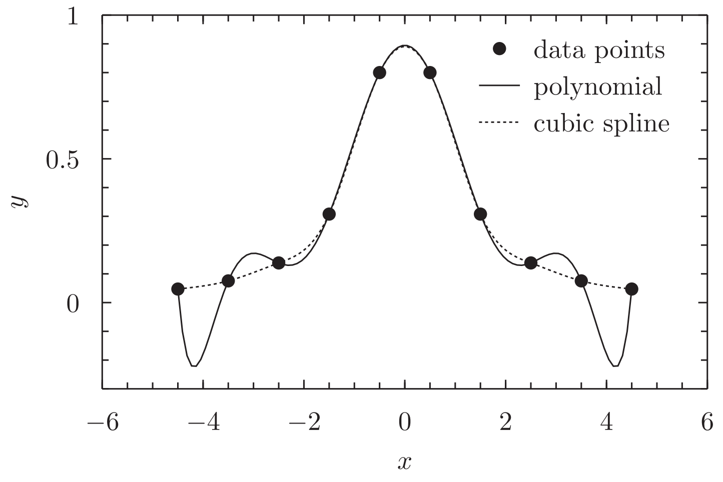

Figure 1: Lagrange interpolating polynomial, solid line, showing the Runge's phenomenon: large oscillations at the edges. For comparison the dashed line shows a cubic spline.

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.

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

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

Table 2: Linear interpolation in C

```
#include<assert.h>
double linterp(int n, double* x, double* y, double z){
  assert(z>=x[0] && z<=x[n-1] && 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;}
  return y[i]+(y[i+1]-y[i])/(x[i+1]-x[i])*(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* algoritm (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. The average number of operations for a binary search is $O(\log n)$.

Quadratic spline

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

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

which identically satisfies the continuity conditions

$$S_i(x_i) = y_i \; , \; S_i(x_{i+1}) = y_{i+1} \Big|_{i=1,\dots,n-1} . \tag{8}$$

Substituting (7) into the derivative continuity condition,

$$S'_i(x_{i+1}) = S'_{i+1}(x_{i+1}) \Big|_{i=1,\dots,n-2} , \tag{9}$$

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

$$p_i + c_i \Delta x_i = p_{i+1} - c_{i+1}\Delta x_{i+1} \Big|_{i=1,\dots,n-2} . \tag{10}$$

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

$$c_{i+1} = \frac{1}{\Delta x_{i+1}} \left(p_{i+1} - p_i - c_i \Delta x_i\right) \Big|_{i=1,\dots,n-2} . \tag{11}$$

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

$$c_i = \frac{1}{\Delta x_i} \left(p_{i+1} - p_i - c_{i+1} \Delta x_{i+1}\right) \Big|_{i=n-2,\dots,1} . \tag{12}$$

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

$$S_i(x) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 \; , \text{ where } b_i = p_i - c_i \Delta x_i \; . \tag{13}$$

An implementation of quadratic spline in C is listed in Table

Cubic spline

Cubic splines are made of third order polynomials,

$$S_i(x) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \; . \tag{14}$$

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,

$$\begin{aligned} y_i + b_i h_i + c_i h_i^2 + d_i h_i^3 &= y_{i+1} \; , \; i = 1, \dots, n-1 \\ b_i + 2c_i h_i + 3d_i h_i^2 &= b_{i+1} \; , \; i = 1, \dots, n-2 \\ 2c_i + 6d_i h_i &= 2c_{i+1} \; , \; i = 1, \dots, n-2 \end{aligned} \tag{15}$$

where

$$h_i = x_{i+1} - x_i \; . \tag{16}$$

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 quadratic spline
  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));//copy of x_i
  s->y = (double*)malloc(n*sizeof(double));//copy of 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); }
```

The set of equations (15) is a set of $3n-5$ linear equations for the $3(n-1)$ unknown coefficients $\{a_i, b_i, c_i \mid i = 1,\dots,n-1\}$. 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,

$$S''(x_1) = S''(x_n) = 0 , \tag{17}$$

which gives

$$\begin{aligned} c_1 &= 0 , \\ c_{n-1} + 3d_{n-1}h_{n-1} &= 0 . \end{aligned} \tag{18}$$

Solving the first two equations in (15) for c_i and d_i gives[1]

$$\begin{aligned} c_i h_i &= -2b_i - b_{i+1} + 3p_i , \\ d_i h_i^2 &= b_i + b_{i+1} - 2p_i , \end{aligned} \tag{19}$$

where $p_i \doteq \frac{\Delta y_i}{h_i}$. The natural conditions (18) and the third equation in (15) then produce the following tridiagonal system of n linear equations for the n coefficients b_i,

$$\begin{aligned} 2b_1 + b_2 &= 3p_1 , \\ b_i + (2\frac{h_i}{h_{i+1}} + 2)b_{i+1} + \frac{h_i}{h_{i+1}} b_{i+2} &= 3(p_i + p_{i+1}\frac{h_i}{h_{i+1}}) \Big|_{i=1,\dots,n-2} , \\ b_{n-1} + 2b_n &= 3p_{n-1} , \end{aligned} \tag{20}$$

or, in the matrix form,

$$\begin{pmatrix} D_1 & Q_1 & 0 & 0 & \dots \\ 1 & D_2 & Q_2 & 0 & \dots \\ 0 & 1 & D_3 & Q_3 & \dots \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ \dots & \dots & 0 & 1 & D_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} B_1 \\ \vdots \\ \vdots \\ B_n \end{pmatrix} \tag{21}$$

[1]introducing an auxiliary coefficient b_n.

where the elements D_i at the main diagonal are

$$D_1 = 2\ ,\ D_{i+1} = 2\frac{h_i}{h_{i+1}} + 2\,\Big|_{i=1,\dots,n-2}\ ,\ D_n = 2\ , \tag{22}$$

the elements Q_i at the above-main diagonal are

$$Q_1 = 1\ ,\ Q_{i+1} = \frac{h_i}{h_{i+1}}\,\Big|_{i=1,\dots,n-2}\ , \tag{23}$$

and the right-hand side terms B_i are

$$B_1 = 3p_1\ ,\ B_{i+1} = 3\left(p_i + p_{i+1}\frac{h_i}{h_{i+1}}\right)\,\Big|_{i=1,\dots,n-2}\ , B_n = 3p_{n-1}\ . \tag{24}$$

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

$$\begin{pmatrix} \tilde{D}_1 & Q_1 & 0 & 0 & \dots \\ 0 & \tilde{D}_2 & Q_2 & 0 & \dots \\ 0 & 0 & \tilde{D}_3 & Q_3 & \dots \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ \dots & \dots & 0 & 0 & \tilde{D}_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} \tilde{B}_1 \\ \vdots \\ \vdots \\ \tilde{B}_n \end{pmatrix}\ , \tag{25}$$

where

$$\tilde{D}_1 = D_1\ ,\ \tilde{D}_i = D_i - Q_{i-1}/\tilde{D}_{i-1}\,\Big|_{i=2,\dots,n}\ , \tag{26}$$

and

$$\tilde{B}_1 = B_1\ ,\ \tilde{B}_i = B_i - \tilde{B}_{i-1}/\tilde{D}_{i-1}\,\Big|_{i=2,\dots,n}\ . \tag{27}$$

The triangular system (25) can be solved by a run of back-substitution,

$$b_n = \frac{1}{\tilde{D}_n}\tilde{B}_n\ ,\ b_i = \frac{1}{\tilde{D}_i}(\tilde{B}_i - Q_i b_{i+1})\,\Big|_{i=n-1,\dots,1}\ . \tag{28}$$

An implementation of quadratic spline in C is listed in Table

Akima sub-spline interpolation

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

$$\mathcal{A}(x)\Big|_{x\in[x_i,x_{i+1}]} = a_i + b_i(x-x_i) + c_i(x-x_i)^2 + d_i(x-x_i)^3 \doteq A_i(x)\ . \tag{29}$$

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 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. (29) are determined by the values of the derivatives $\mathcal{A}'_i \doteq \mathcal{A}'(x_i)$ of the sub-spline through the continuity conditions for the sub-spline and its first derivative,

$$A_i(x_i) = y_i,\ A'_i(x_i) = \mathcal{A}'_i,\ A_i(x_{i+1}) = y_{i+1},\ A'_i(x_{i+1}) = \mathcal{A}'_{i+1}. \tag{30}$$

Inserting (29) into (30) and solving for the coefficients gives

$$a_i = y_i,\ b_i = \mathcal{A}'_i,\ c_i = \frac{3p_i - 2\mathcal{A}'_i - \mathcal{A}'_{i+1}}{\Delta x_i},\ d_i = \frac{\mathcal{A}'_i + \mathcal{A}'_{i+1} - 2p_i}{(\Delta x_i)^2}, \tag{31}$$

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

Table 4: Cubic spline in C

```
#include<stdlib.h>
#include<assert.h>
typedef struct {int n; double *x,*y,*b,*c,*d;} cspline;
cspline* cspline_alloc(int n, double *x, double *y)
{// builds natural cubic spline
  cspline* s = (cspline*)malloc(sizeof(cspline));
  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];
  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 cspline_eval(cspline*s,double z){ // evaluation of spline
    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 cspline_free(cspline *s){ //free the allocated memory
  free(s->x);free(s->y);free(s->b);free(s->c);free(s->d);free(s);}
```

In the ordinary qubic spline the derivatives $\mathcal{A}'_i$ 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,

$$\mathcal{A}'_i = \frac{w_{i+1}p_{i-1} + w_{i-1}p_i}{w_{i+1} + w_{i-1}}\,, \quad \text{if} \quad w_{i+1} + w_{i-1} \neq 0\,, \tag{32}$$

$$\mathcal{A}'_i = \frac{p_{i-1} + p_i}{2}\,, \quad \text{if} \quad w_{i+1} + w_{i-1} = 0\,, \tag{33}$$

where the weights w_i are given as

$$w_i = |p_i - p_{i-1}|\,. \tag{34}$$

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

$$\mathcal{A}'_1 = p_1,\ \mathcal{A}'_1 = \frac{1}{2}p_1 + \frac{1}{2}p2,\ \mathcal{A}'_n = p_{n-1},\ \mathcal{A}'_{n-1} = \frac{1}{2}p_{n-1} + \frac{1}{2}p_{n-2}. \tag{35}$$

Table (5) shows a javascript implementation of this algorithm.

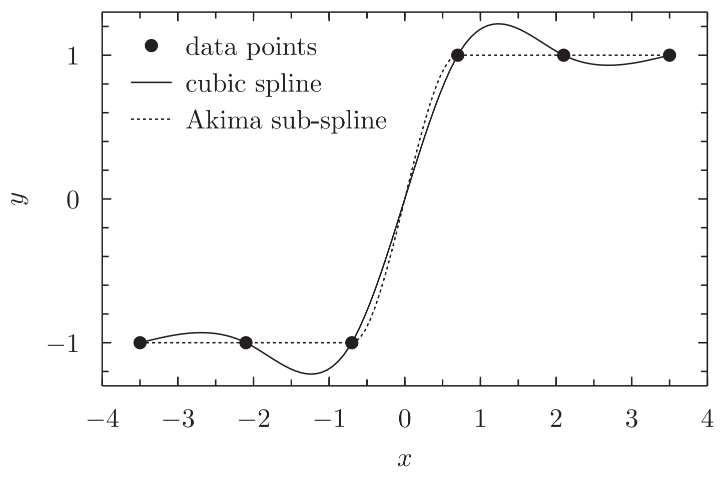

Figure 2: A cubic spline (solid line) showing the typical wiggles, compared to the Akima sub-spline (dashed line) where the wiggles are essentially removed.

Other forms of interpolation

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

Sometimes not only the values of the function are tabulated but also the values of its derivative. This extra information can be taken advantage of when constructing the interpolation function.

Multivariate interpolation

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

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.

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,

$$S(x, y) = a + bx + cy \; , \tag{36}$$

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

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 product of two linear functions of x and y respectively,

$$B(x, y) = (\alpha + \beta x)(\gamma + \delta y) \doteq a + bx + cy + dxy \; , \tag{37}$$

Table 5: Javascript implementation of Akima sub-spline.

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

function akima(x,y){
  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), c=new Array(n-1), d=new Array(n-1);

  A[0]=p[0]; A[1]=(p[0]+p[1])/2;
  A[n-1]=p[n-2]; A[n-2]=(p[n-2]+p[n-3])/2;
  for(var i=2;i<n-2;i++)
  { w1=Math.abs(p[i+1]-p[i]); w2=Math.abs(p[i-1]-p[i-2]);
    if(w1+w2==0) A[i]=(p[i-1]+p[i])/2;
    else A[i]=(w1*p[i-1]+w2*p[i])/(w1+w2);
  }

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

  var spline=function(z)
  { if(z<x[0] || z>x[n-1]) throw "spline: out of range";
    var i=0, j=n-1;
    while(j-i>1){
      var mid=Math.round( (i+j)/2 );
      if(z>x[mid]) i=mid; else j=mid;}
    var dx=z-x[i];
    return y[i]+dx*(A[i]+dx*(c[i]+dx*d[i]));
  }
  return spline;
}
```

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 vertices of the given grid rectangle).