CUBIC SPLINE INTERPOLATION: A REVIEW

George Wolberg

Department of Computer Science
Columbia University
New York, NY 10027
wolberg@cs.columbia.edu

September 1988
Technical Report CUCS-389-88

ABSTRACT

The purpose of this paper is to review the fundamentals of interpolating cubic splines. We begin by defining a cubic spline in Section 1. Since we are dealing with interpolating splines, constraints are imposed to guarantee that the spline actually passes through the given data points. These constraints are described in Section 2. They establish a relationship between the known data points and the unknown coefficients used to completely specify the spline. Due to extra degrees of freedom, the coefficients may be solved in terms of the first or second derivatives. Both derivations are given in Section 3. Once the coefficients are expressed in terms of either the first or second derivatives, these unknown derivatives must be determined. Their solution, using one of several end conditions, is given in Section 4. Finally source code, written in C, is provided in Section 5 to implement cubic spline interpolation for uniformly and nonuniformly spaced data points.

1. DEFINITION

A cubic spline $f(x)$ interpolating on the partition $x_0 < x_1 < \cdots < x_{n-1}$ is a function for which $f(x_k) = y_k$. It is a piecewise polynomial function that consists of $n-1$ cubic polynomials f_k defined on the ranges $[x_k, x_{k+1}]$. Furthermore, f_k are joined at x_k $(k=1,...,n-2)$ such that f'_k and f''_k are continuous. An example of a cubic spline passing through n data points is illustrated in Fig. 1.

The k^{th} polynomial piece, f_k, is defined over the fixed interval $[x_k, x_{k+1}]$ and has the cubic form

$$f_k(x) = A_3(x - x_k)^3 + A_2(x - x_k)^2 + A_1(x - x_k) + A_0 \tag{1.1}$$

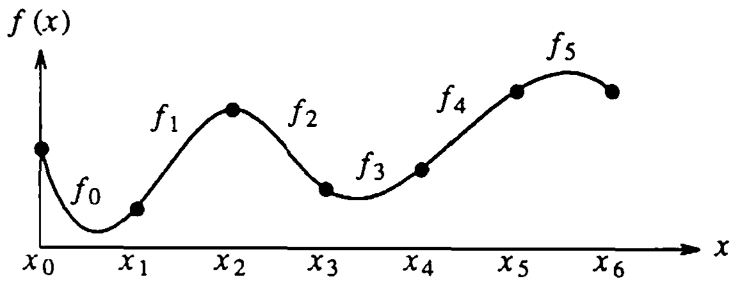

Figure 1: Cubic spline.

2. CONSTRAINTS

Given only the data points (x_k, y_k), we must determine the polynomial coefficients, A, for each partition such that the resulting polynomials pass through the data points and are continuous in their first and second derivatives. These conditions require f_k to satisfy the following constraints

$$\begin{aligned} y_k &= f_k(x_k) &&= A_0 \\ y_{k+1} &= f_k(x_{k+1}) &&= A_3\Delta x_k^3 + A_2\Delta x_k^2 + A_1\Delta x_k + A_0 \end{aligned} \tag{2.1}$$

$$\begin{aligned} y'_k &= f'_k(x_k) &&= A_1 \\ y'_{k+1} &= f'_k(x_{k+1}) &&= 3A_3\Delta x_k^2 + 2A_2\Delta x_k + A_1 \end{aligned} \tag{2.2}$$

$$\begin{aligned} y''_k &= f''_k(x_k) &&= 2A_2 \\ y''_{k+1} &= f''_{k+1}(x_k) &&= 6A_3\Delta x_k + 2A_2 \end{aligned} \tag{2.3}$$

Note that these conditions apply at the data points (x_k, y_k). If the x_k's are defined on a regular grid, they are equally spaced and $\Delta x_k = x_{k+1} - x_k = 1$. This eliminates all of the

Δx_k terms in the above equations. Consequently, Eqs. (2.1) through (2.3) reduce to

$$\begin{aligned} y_k &= A_0 \\ y_{k+1} &= A_3 + A_2 + A_1 + A_0 \end{aligned} \tag{2.4}$$

$$\begin{aligned} y'_k &= A_1 \\ y'_{k+1} &= 3A_3 + 2A_2 + A_1 \end{aligned} \tag{2.5}$$

$$\begin{aligned} y''_k &= 2A_2 \\ y''_{k+1} &= 6A_3 + 2A_2 \end{aligned} \tag{2.6}$$

In the remainder of this paper, we will refrain from making any simplifying assumptions about the spacing of the data points in order to treat the more general case.

3. SOLVING FOR THE SPLINE COEFFICIENTS

The conditions given above are used to find A_3, A_2, A_1, and A_0 which are needed to define the cubic polynomial piece f_k. Isolating the coefficients, we get

$$\begin{aligned} A_0 &= y_k \\ A_1 &= y'_k \\ A_2 &= \frac{1}{\Delta x_k}\left[3\frac{\Delta y_k}{\Delta x_k} - 2y'_k - y'_{k+1} \right] \\ A_3 &= \frac{1}{\Delta x_k^2}\left[-2\frac{\Delta y_k}{\Delta x_k} + y'_k + y'_{k+1} \right] \end{aligned} \tag{3.1}$$

In the expressions for A_2 and A_3, $k = 0, \ldots, n-2$ and $\Delta y_k = y_{k+1} - y_k$.

3.1. Derivation of A_2

From (2.1),

$$A_2 = \frac{y_{k+1} - A_3 \Delta x_k^3 - y'_k \Delta x_k - y_k}{\Delta x_k^2} \tag{3.2a}$$

From (2.2),

$$2A_2 = \frac{y'_{k+1} - 3A_3 \Delta x_k^2 - y'_k}{\Delta x_k} \tag{3.2b}$$

Finally, A_2 is derived from (3.2a) and (3.2b)

$$\left[3 \times (3.2a) \right] - \left[\frac{\Delta x_k}{\Delta x_k} \times (3.2b) \right] = A_2$$

3.2. Derivation of A_3

From (2.1),

$$A_3 = \frac{y_{k+1} - A_2\Delta x_k^2 - y'_k\Delta x_k - y_k}{\Delta x_k^3} \tag{3.2c}$$

From (2.2),

$$3A_3 = \frac{y'_{k+1} - 2A_2\Delta x_k - y'_k}{\Delta x_k^2} \tag{3.2d}$$

Finally, A_3 is derived from (3.2c) and (3.2d)

$$\left[\frac{\Delta x_k}{\Delta x_k} \times (3.2d)\right] - \left[2 \times (3.2c)\right] = A_3$$

The equations in (3.1) express the coefficients of f_k in terms of x_k, y_k, x_{k+1}, y_{k+1}, (known) and y'_k, y'_{k+1} (unknown). Since the expressions in Eqs. (2.1) through (2.3) present six equations for the four A_i coefficients, the A terms could alternately be expressed in terms of second derivatives, instead of the first derivatives given in Eq. (3.1). This yields

$$\begin{aligned}
A_0 &= y_k \\
A_1 &= \frac{\Delta y_k}{\Delta x_k} - \frac{\Delta x_k}{6}\left[y''_{k+1} + 2y''_k\right] \\
A_2 &= \frac{y''_k}{2} \\
A_3 &= \frac{1}{6\Delta x_k}\left[y''_{k+1} - y''_k\right]
\end{aligned} \tag{3.3}$$

3.3. Derivation of A_1 and A_3

From (2.1),

$$A_1 = \frac{y_{k+1} - A_3\Delta x_k^3 - \frac{y''_k}{2}\Delta x_k^2 - y_k}{\Delta x_k} \tag{3.4a}$$

From (2.3),

$$A_3 = \frac{y''_{k+1} - y''_k}{6\Delta x_k} = \frac{\Delta y''_k}{6\Delta x_k} \tag{3.4b}$$

Plugging Eq. (3.4b) into (3.4a),

$$A_1 = \frac{\Delta y_k}{\Delta x_k} - \frac{\Delta x_k}{6}\left[y''_{k+1} - y''_k\right] - \frac{y''_k}{2}\Delta x_k = \frac{\Delta y_k}{\Delta x_k} - \frac{\Delta x_k}{6}\left[y''_{k+1} + 2y''_k\right] \tag{3.4c}$$

4. EVALUATING THE UNKNOWN DERIVATIVES

Having expressed the cubic polynomial coefficients in terms of data points and derivatives, the unknown derivatives still remain to be determined. They are typically not given explicitly. Instead, we may evaluate them from the given constraints. Although the spline coefficients require *either* the first derivatives or the second derivatives, we shall derive both forms for the sake of completeness.

4.1. First Derivatives

We begin by deriving the expressions for the first derivatives using Eqs. (2.1) through (2.3). Recall that the A coefficients expressed in terms of y' made use of Eqs. (2.1) and (2.2). We therefore use the remaining constraint, given in Eq. (2.3), to express the desired relation. Constraint Eq. (2.3) defines the second derivative of f_k at the endpoints of its interval. By establishing that $f''_{k-1}(x_k) = f''_k(x_k)$, we enforce the continuity of the second derivative across the intervals and give rise to a relation for the first derivatives.

$$6A_3^{k-1}\Delta x_{k-1} + 2A_2^{k-1} = 2A_2^k \tag{4.1}$$

Note that the superscripts refer to the interval of the coefficient. Plugging Eq. (3.1) into Eq. (4.1) yields

$$\frac{1}{\Delta x_{k-1}}\left[-12\frac{\Delta y_{k-1}}{\Delta x_{k-1}} + 6y'_{k-1} + 6y'_k\right] + \frac{1}{\Delta x_{k-1}}\left[6\frac{\Delta y_{k-1}}{\Delta x_{k-1}} - 4y'_{k-1} - 2y'_k\right] = \frac{1}{\Delta x_k}\left[6\frac{\Delta y_k}{\Delta x_k} - 4y'_k - 2y'_{k+1}\right]$$

$$\frac{1}{\Delta x_{k-1}}\left[-6\frac{\Delta y_{k-1}}{\Delta x_{k-1}} + 2y'_{k-1} + 4y'_k\right] = \frac{1}{\Delta x_k}\left[6\frac{\Delta y_k}{\Delta x_k} - 4y'_k - 2y'_{k+1}\right]$$

After collecting the y' terms on one side, we have Eq. (4.2):

$$y'_{k-1}\left[\frac{1}{\Delta x_{k-1}}\right] + y'_k\left[2\left[\frac{1}{\Delta x_{k-1}} + \frac{1}{\Delta x_k}\right]\right] + y'_{k+1}\left[\frac{1}{\Delta x_k}\right] = 3\left[\frac{\Delta y_{k-1}}{\Delta x_{k-1}^2} + \frac{\Delta y_k}{\Delta x_k^2}\right]$$

Equation (4.2) yields a matrix of $n-2$ equations in n unknowns. We can reduce the need for division operations by multiplying both sides by $\Delta x_{k-1}\,\Delta x_k$. This gives us the following system of equations, with $1 \le k \le n-2$. For notational convenience, we let $h_k = \Delta x_k$ and $r_k = \Delta y_k / \Delta x_k$.

$$\begin{bmatrix} h_1 & 2(h_0+h_1) & h_0 & & \\ & h_2 & 2(h_1+h_2) & h_1 & \\ & & \cdot & \cdot & \\ & & \cdot & \cdot & \\ & & h_{n-2} & 2(h_{n-3}+h_{n-2}) & h_{n-3} \end{bmatrix} \begin{bmatrix} y'_0 \\ y'_1 \\ y'_2 \\ \cdot \\ \cdot \\ y'_{n-2} \\ y'_{n-1} \end{bmatrix} = \begin{bmatrix} 3(r_0h_1 + r_1h_0) \\ 3(r_1h_2 + r_2h_1) \\ \cdot \\ \cdot \\ 3(r_{n-3}h_{n-2} - r_{n-2}h_{n-3}) \end{bmatrix}$$

When the two *end* tangent vectors y'_0 and y'_{n-1} are specified, then the system of equations becomes determinable. One of several boundary conditions described later may be selected to yield the remaining two equations in the matrix.

4.2. Second Derivatives

An alternate, but equivalent, course of action is to determine the spline coefficients by solving for the unknown second derivatives. This procedure is virtually identical to the approach given above. Note that while there is no particular benefit in using second derivatives rather than first derivatives, it is presented here for generality.

As before, we note that the A coefficients expressed in terms of y'' made use of Eqs. (2.1) and (2.3). We therefore use the remaining constraint, given in Eq. (2.2), to express the desired relation. Constraint Eq. (2.2) defines the first derivative of f_k at the endpoints of its interval. By establishing that $f'_{k-1}(x_k) = f'_k(x_k)$ we enforce the continuity of the first derivative across the intervals and give rise to a relation for the second derivatives.

$$3A_3^{k-1}\Delta x_{k-1}^2 + 2A_2^{k-1}\Delta x_{k-1} + A_1^{k-1} = A_1^k \tag{4.3}$$

Again, the superscripts refer to the interval of the coefficient. Plugging Eq. (3.3) into Eq. (4.3) yields

$$\frac{\Delta x_{k-1}}{2}\left[y''_k - y''_{k-1}\right] + y''_{k-1}\Delta x_{k-1} + \left[\frac{\Delta y_{k-1}}{\Delta x_{k-1}} - \frac{\Delta x_{k-1}}{6}\left[y''_k + 2y''_{k-1}\right]\right] = \left[\frac{\Delta y_k}{\Delta x_k} - \frac{\Delta x_k}{6}\left[y''_{k+1} + 2y''_k\right]\right]$$

After collecting the y'' terms on one side, we have

$$y''_{k-1}\left[\Delta x_{k-1}\right] + y''_k\left[2\Delta x_{k-1} + \Delta x_k\right] + y''_{k+1}\left[\Delta x_k\right] = 6\left[\frac{\Delta y_k}{\Delta x_k} - \frac{\Delta y_{k-1}}{\Delta x_{k-1}}\right] \tag{4.4}$$

Equation (4.4) yields the following matrix of $n-2$ equations in n unknowns. Again, for notational convenience we let $h_k = \Delta x_k$ and $r_k = \Delta y_k / \Delta x_k$.

$$\begin{bmatrix} h_0 & 2(h_0+h_1) & h_1 & & & \\ & h_1 & 2(h_1+h_2) & h_2 & & \\ & & \cdot & \cdot & & \\ & & \cdot & \cdot & & \\ & & & h_{n-3} & 2(h_{n-3}+h_{n-2}) & h_{n-2} \end{bmatrix} \begin{bmatrix} y''_0 \\ y''_1 \\ y''_2 \\ \cdot \\ \cdot \\ y''_{n-2} \\ y''_{n-1} \end{bmatrix} = \begin{bmatrix} 6(r_1 - r_0) \\ 6(r_2 - r_1) \\ \cdot \\ \cdot \\ 6(r_{n-2} - r_{n-3}) \end{bmatrix}$$

The system of equations becomes determinable once the boundary conditions are specified.

4.3. Boundary Conditions: Free-end, Cyclic, and Not-A-Knot

A trivial choice for the boundary condition is achieved by setting $y_0'' = y_{n-1}'' = 0$. This is known as the *free-end* condition that results in *natural spline interpolation*. Since $y_0'' = 0$, we know from Eq. (2.6) that $A_2 = 0$. As a result, we derive the following expression from Eq. (3.1).

$$y_0' + \frac{y_1}{2} = \frac{3\Delta y_0}{2\Delta x_0} \tag{4.5}$$

Similarly, since $y_{n-1}'' = 0$, $6A_3 + 2A_2 = 0$, and we derive the following expression from Eq. (3.1).

$$2y_{n-2}' + 4y_{n-1}' = 6\frac{\Delta y_{n-2}}{\Delta x_{n-2}} \tag{4.6}$$

Another condition is called the *cyclic* condition, where the derivatives at the endpoints of the span are set equal to each other.

$$\begin{aligned} y_0' &= y_{n-1}' \\ y_0'' &= y_n'' \end{aligned} \tag{4.7}$$

The boundary condition that we shall consider is the *not-a-knot* condition. This requires y''' to be continuous across x_1 and x_{n-2}. In effect, this extrapolates the curve from the adjacent interior segments [de Boor 78]. As a result, we get

$$\begin{aligned} A_3^0 &= A_3^1 \\ \frac{1}{\Delta x_0^2}\left[-2\frac{\Delta y_0}{\Delta x_0} + y_0' + y_1'\right] &= \frac{1}{\Delta x_1^2}\left[-2\frac{\Delta y_1}{\Delta x_1} + y_1' + y_2'\right] \end{aligned} \tag{4.8}$$

Replacing y_2' with an expression in terms of y_0' and y_1' allows us to remain consistent with the structure of a tridiagonal matrix already derived earlier. From Eq. (4.2), we isolate y_2' and get

$$y_2' = 3\Delta x_1\left[\frac{\Delta y_0}{\Delta x_0^2} + \frac{\Delta y_1}{\Delta x_1^2}\right] - y_0'\frac{\Delta x_1}{\Delta x_0} - 2y_1'\left[\frac{\Delta x_1 + \Delta x_0}{\Delta x_0}\right] \tag{4.9}$$

Substituting this expression into Eq. (4.8) yields

$$y_0'\Delta x_1\left[\Delta x_0 + \Delta x_1\right] + y_1'\left[\Delta x_0 + \Delta x_1\right]^2 = \frac{\Delta y_0}{\Delta x_0}\left[3\Delta x_0\Delta x_1 + 2\Delta x_1^2\right] + \frac{\Delta y_1}{\Delta x_1}\left[\Delta x_0^2\right]$$

Similarly, the last row is derived to be

$$\begin{aligned} y_{n-2}'\left[\Delta x_{n-3} + \Delta x_{n-2}\right]^2 + y_{n-1}'\Delta x_{n-3}\left[\Delta x_{n-3} + \Delta x_{n-2}\right] = \\ \frac{\Delta y_{n-3}}{\Delta x_{n-3}}\left[\Delta x_{n-2}^2\right] + \frac{\Delta y_{n-2}}{\Delta x_{n-2}}\left[3\Delta x_{n-3}\Delta x_{n-2} + 2\Delta\, x_{n-3}^2\right] \end{aligned}$$

The version of this boundary condition expressed in terms of second derivatives is left to the reader as an exercise.

Thus far we have placed no restrictions on the spacing between the data points. Many simplifications are possible if we assume that the points are equispaced, i.e., $\Delta x_k = 1$. This is certainly the case for image reconstruction, where cubic splines can be used to compute image values between regularly spaced samples. The not-a-knot boundary condition used in conjunction with the system of equations given in Eq. (4.2) is shown below. To solve for the polynomial coefficients, the column vector containing the first derivatives must be solved and then substituted into Eq. (3.1).

$$\begin{bmatrix} 2 & 4 & & & & \\ 1 & 4 & 1 & & & \\ & 1 & 4 & 1 & & \\ & & \cdot & & & \\ & & & \cdot & & \\ & & & 1 & 4 & 1 \\ & & & & 4 & 2 \end{bmatrix} \begin{bmatrix} y'_0 \\ y'_1 \\ y'_2 \\ \cdot \\ \cdot \\ y'_{n-2} \\ y'_{n-1} \end{bmatrix} = \begin{bmatrix} -5y_0 + 4y_1 + y_2 \\ 3(y_2 - y_0) \\ 3(y_3 - y_1) \\ \cdot \\ \cdot \\ 3(y_{n-1} - y_{n-3}) \\ -y_{n-3} - 4y_{n-2} + 5y_{n-1} \end{bmatrix} \tag{4.10}$$

5. SOURCE CODE

Below we include two C programs for interpolating cubic splines. The first program, called *ispline*, assumes that the supplied data points are equispaced. The second program, *ispline_gen*, addresses the more general case of irregularly spaced data.

5.1. Ispline

The function *ispline* takes $Y1$, a list of $len1$ numbers in double-precision, and passes an interpolating cubic spline through that data. The spline is then resampled at $len2$ equal intervals and stored in list $Y2$. It begins by computing the unknown first derivatives at each interval endpoint. It invokes the function *getYD*, which returns the first derivatives in the list *YD*. Along the way, function *tridiag* is called to solve the tridiagonal system of equations shown in Eq. (4.10). Since each derivative is coupled only to its adjacent neighbors on both sides, the equations can be solved in linear time, i.e., $O(n)$. Once *YD* is initialized, it is used together with $Y1$ to compute the spline coefficients. In the interest of speed, the cubic polynomials are evaluated by using Horner's rule for factoring polynomials. This requires three additions and three multiplications per evaluated point.

```
/*********************************************************************
	Interpolating cubic spline function for equispaced points
	Input: Y1 is a list of equispaced data points with len1 entries
	Output: Y2 <- cubic spline sampled at len2 equispaced points
*********************************************************************/
ispline(Y1,len1,Y2,len2)
double *Y1, *Y2;
int len1, len2;
{
	int i, ip, oip;
	double *YD, A0, A1, A2, A3, x, p, fctr;

	/* compute 1st derivatives at each point -> YD */
	YD = (double *) calloc(len1, sizeof(double));
	getYD(Y1,YD,len1);

	/*
	 * p is real-valued position into spline
	 * ip is interval's left endpoint (integer)
	 * oip is left endpoint of last visited interval
	 */
	oip = -1;			/* force coefficient initialization */
	fctr = (double) (len1-1) / (len2-1);
	for(i=p=ip=0; i < len2; i++) {
		/* check if in new interval */
		if(ip != oip) {
			/* update interval */
			oip = ip;

			/* compute spline coefficients */
			A0 = Y1[ip];
			A1 = YD[ip];
			A2 = 3.0*(Y1[ip+1]-Y1[ip]) - 2.0*YD[ip] - YD[ip+1];
			A3 = -2.0*(Y1[ip+1]-Y1[ip]) + YD[ip] + YD[ip+1];
		}
		/* use Horner's rule to calculate cubic polynomial */
		x = p - ip;
		Y2[i] = ((A3*x + A2)*x + A2)*x + A0;

		/* increment pointer */
		ip = (p += fctr);
	}
	cfree((char *) YD);
}
```

```
/**********************************************************************
        YD <- Computed 1st derivative of data in Y (len entries)
        The not-a-knot boundary condition is used
**********************************************************************/
getYD(Y,YD,len)
double *Y, *YD;
int len;
{
        int i;

        YD[0] = -5.0*Y[0] + 4.0*Y[1] + Y[2];
        for(i = 1; i < len-1; i++) YD[i]=3.0*(Y[i+1]-Y[i-1]);
        YD[len-1] = -Y[len-3] - 4.0*Y[len-2] + 5.0*Y[len-1];

        /* solve for the tridiagonal matrix: YD=YD*inv(tridiag matrix) */
        tridiag(YD,len);
}

/**********************************************************************
        Linear time Gauss Elimination with backsubstitution for 141
        tridiagonal matrix with column vector D. Result goes into D
**********************************************************************/
tridiag(D,len)
double *D;
int len;
{
        int i;
        double *C;

        /* init first two entries; C is right band of tridiagonal */
        C = (double *) calloc(len, sizeof(double));
        D[0] = 0.5*D[0];
        D[1] = (D[1] - D[0]) / 2.0;
        C[1] = 2.0;

        /* Gauss elimination; forward substitution */
        for(i = 2; i < len-1; i++) {
                C[i] = 1.0 / (4.0 - C[i-1]);
                D[i] = (D[i] - D[i-1]) / (4.0 - C[i]);
        }
        C[i] = 1.0 / (4.0 - C[i-1]);
        D[i] = (D[i] - 4*D[i-1]) / (2.0 - 4*C[i]);

        /* backsubstitution */
        for(i = len-2; i >= 0; i--) D[i] -= (D[i+1] * C[i+1]);
        cfree((char *) C);
}
```

5.2. Ispline_gen

The function *ispline_gen* takes the data points in $(X1, Y1)$, two lists of $len1$ numbers, and passes an interpolating cubic spline through that data. The spline is then resampled at $len2$ positions and stored in $Y2$. The resampling locations are given by $X2$. The function assumes that $X2$ is monotonically increasing and lies withing the range of numbers in $X1$.

As before, we begin by computing the unknown first derivatives at each interval endpoint. The function *getYD_gen* is then invoked to return the first derivatives in the list *YD*. Along the way, function *tridiag_gen* is called to solve the tridiagonal system of equations given in Eq. (4.2). Once *YD* is initialized, it is used together with $Y1$ to compute the spline coefficients. Note that in this general case, additional consideration must now be given to determine the polynomial interval in which the resampling point lies.

```
/*******************************************************************
	Interpolating cubic spline function for irregularly-spaced points
	Input:  Y1 is a list of irregular data points (len1 entries)
		Their x-coordinates are specified in X1
	Output: Y2 <- cubic spline sampled according to X2 (len2 entries)
		Assume that X1,X2 entries are monotonically increasing
*******************************************************************/
ispline_gen(X1,Y1,len1,X2,Y2,len2)
double *X1, *Y1, *X2, *Y2;
int len1, len2;
{
	int i, j;
	double *YD, A0, A2, A2, A3, x, dx, dy, p1, p2, p3;

	/* compute 1st derivatives at each point -> YD */
	YD = (double *) calloc(len1, sizeof(double));
	getYD_gen(X1,Y1,YD,len1);

	/* error checking */
	if(X2[0] < X1[0] || X2[len2-1] > X1[len1-1]) {
		fprintf(stderr,"ispline_gen: Out of range0);
		exit();
	}

	/*
	 * p1 is left endpoint of interval
	 * p2 is resampling position
	 * p3 is right endpoint of interval
	 * j  is input index for current interval
	 */
	p3 = X2[0] - 1;		/* force coefficient initialization */
	for(i=j=0; i < len2; i++) {
		/* check if in new interval */
		p2 = X2[i];
```

```
        if(p2 > p3) {
            /* find the interval which contains p2 */
            for(; j<len1 && p2>X1[j]; j++);
            if(p2 < X1[j]) j--;
            p1 = X1[j];          /* update left  endpoint */
            p3 = X1[j+1];        /* update right endpoint */

            /* compute spline coefficients */
            dx = 1.0 / (X1[j+1] - X1[j]);
            dy = (Y1[j+1] - Y1[j]) * dx;
            A0 = Y1[j];
            A2 = YD[j];
            A2 = dx * (3.0*dy - 2.0*YD[j] - YD[j+1]);
            A3 = dx*dx * (-2.0*dy + YD[j] + YD[j+1]);
        }
        /* use Horner's rule to calculate cubic polynomial */
        x = p2 - p1;
        Y2[i] = ((A3*x + A2)*x + A1)*x + A0;
    }
    cfree((char *) YD);
}
```

```
/*********************************************************************
	YD <- Computed 1st derivative of data in X,Y (len entries)
	The not-a-knot boundary condition is used
*********************************************************************/

getYD_gen(X,Y,YD,len)
double *X, *Y, *YD;
int len;
{
	int i;
	double h0, h1, r0, r1, *A, *B, *C;

	/* allocate memory for tridiagonal bands A,B,C */
	A = (double *) calloc(len, sizeof(double));
	B = (double *) calloc(len, sizeof(double));
	C = (double *) calloc(len, sizeof(double));

	/* init first row data */
	h0 =  X[1] - X[0];		h1 =  X[2] - X[1];
	r0 = (Y[1] - Y[0]) / h0;	r1 = (Y[2] - Y[1]) / h1;
	B[0]  = h1 * (h0+h1);
	C[0]  = (h0+h1) * (h0+h1);
	YD[0] = r0*(3*h0*h1 + 2*h1*h1) + r1*h0*h0;

	/* init tridiagonal bands A, B, C, and column vector YD */
	/* YD will later be used to return the derivatives */
	for(i = 1; i < len-1; i++) {
		h0 = X[i] - X[i-1];		h1 = X[i+1] - X[i];
		r0 = (Y[i] - Y[i-1]) / h0;	r1 = (Y[i+1] - Y[i]) / h1;
		A[i] = h1;
		B[i] = 2 * (h0+h1);
		C[i] = h0;
		YD[i] = 3 * (r0*h1 + r1*h0);
	}

	/* last row */
	A[i]  = (h0+h1) * (h0+h1);
	B[i]  = h0 * (h0+h1);
	YD[i] = r0*h1*h1 + r1*(3*h0*h1 + 2*h0*h0);

	/* solve for the tridiagonal matrix: YD=YD*inv(tridiag matrix) */
	tridiag_gen(A,B,C,YD,len);

	cfree((char *) A);
	cfree((char *) B);
	cfree((char *) C);
}

/*********************************************************************
	Gauss Elimination with backsubstitution for general
	tridiagonal matrix with bands A,B,C and column vector D.
```

```
*********************************************************************/
tridiag_gen(A,B,C,D,len)
double *A, *B, *C, *D;
int len;
{
        int i;
        double b, *F;

        F = (double *) calloc(len, sizeof(double));

        /* Gauss elimination; forward substitution */
        b = B[0];
        D[0] = D[0] / b;
        for(i = 1; i < len; i++) {
                F[i] = C[i-1] / b;
                b = B[i] - A[i]*F[i];
                if(b == 0) {
                        fprintf(stderr,"getYD_gen: divide-by-zero0);
                        exit();
                }
                D[i] = (D[i] - D[i-1]*A[i]) / b;
        }

        /* backsubstitution */
        for(i = len-2; i >= 0; i--) D[i] -= (D[i+1] * F[i+1]);

        cfree((char *) F);
}
```