Journal of Natural Sciences and Mathematics
Qassim University, Vol. 10, No. 1, pp 71-86 (January 2017/Rabi II 1437H.)

Inverse Cubic Spline By Applying Reversion of Series Method

Dhanya Ramachandran* and Dr.V. Madhukar Mallayya**
**Assistant Professor (Maths) Mar Baselios College Of Engg. & Tech Nalanchira , Trivandrum, dhanyar@gmail.com*
***Professor and Head (Maths) Mohandas College of Engg. & Tech Anad, Trivandrum, mmallayyav@gmail.com*

Abstract. By applying series reversion method a formula for inverse cubic spline interpolation is derived and illustrated and the computer programme for inverse cubic spline interpolation is also given.

Key words: Cubic Spline, Reversion of series, Inverse Cubic spline.

1. Introduction

A Spline function is a function consisting of polynomial bits joined together with certain smoothness conditions. Cubic Spline consisting of third degree polynomial bits is most frequently used where the cubic polynomials are joined together in such a way that the resulting spline function has two continuous derivatives everywhere. Cubic Spline functions are most popularly used because they are smooth functions without having the usual oscillatory behaviour characteristic of higher degree polynomial interpolation. [2],[3]

$$S(x)=\begin{cases} S_1(x), a_1 < x < a_2 \\ S_2(x), a_2 < x < a_3 \\ \quad ' \\ \quad ' \\ \quad ' \\ S_n(x), a_{n-1} < x < a_n \end{cases}$$

Given a set of data points (a_i, y_i), i = 1,2,…,n satisfied by a function $y = f(x)$ a cubic interpolation means fitting a unique series of cubic polynomials between each of the data points. Consider an interval $I = [a,b]$ composed of n sub intervals $I_i = [a_{i-1}, a_i]$, i =2,3,…….,n with $a = a_1 < a_2 < - - - < a_n = b$. A spline S(x) [1] ,[4] of degree m on the interval I is a polynomial having continuous derivatives up to order m-1 and coinciding with $y = f(x)$ on each subinterval $I_i = [a_{i-1}, a_i]$, i = 2,3,……..,n such that

The end conditions for natural cubic spline are $S''(a) = S''(b) = 0$ and $S'(a) = f'(a),\ S'(b) = f'(b).$

Since $S''(x)$ is piecewise linear and continuous

$$S''(x) = M_{i-1}\frac{a_i - x}{h_i} + M_i\frac{x - a_{i-1}}{h_i}$$

where $M_i's$ are constants and $h_i = a_i - a_{i-1}$, i = 2,3,…….,n

The natural cubic spline for $x \in [a_{i-1}, a_i]$ is

$$S_{i-1}(x) = M_{i-1}\frac{(a_i - x)^3}{6h_i} + M_i\frac{(x - a_{i-1})^3}{6h_i} + \left(y_{i-1} - \frac{M_{i-1}h_i^{\,2}}{6}\right)\left(\frac{a_i - x}{h_i}\right) + \left(y_i - \frac{M_i h_i^{\,2}}{6}\right)\left(\frac{x - a_{i-1}}{h_i}\right)$$

i = 2,3,…….,n and the $M_i^{'}s$ are obtained by

$$\mu_i M_{i-1} + 2M_i + \lambda_i M_{i+1} = d_i \text{ , i = 2,3,……,n-1}$$

and

$$2M_1 + \lambda_1 M_2 = d_1$$
$$\mu_n M_{n-1} + 2M_n = d_n$$

Using reversion of series method [5] discussed below a formula for inverse cubic spline interpolation is derived in section 3.

2. Reversion of Series Method

Consider a cubic polynomial $y = a_0 + a_1x + a_2x^2 + a_3x^3$

$$\text{So } y - a_0 = a_1x + a_2x^2 + a_3x^3 \tag{2.1}$$

Let $Y = y - a_0$

$$\text{So } (2.1) \Rightarrow Y = a_1x + a_2x^2 + a_3x^3 \tag{2.2}$$

$$\text{Let the inverted series of (2.2) be } x = A_1Y + A_2Y^2 + A_3Y^3 \tag{2.3}$$

Substituting (2.2) into (2.3) and simplifying we get

$$A_1 = \frac{1}{a_1}$$

$$A_2 = \frac{-a_2}{a_1^{\,3}}$$

$$A_3 = \frac{2a_2^{\,2} - a_3a_1}{a_1^{\,5}}$$

$$\text{So } x = \frac{1}{a_1}Y + \frac{-a_2}{a_1^{\,3}}Y^2 + \frac{2a_2^{\,2} - a_3a_1}{a_1^{\,5}}Y^3 \text{ where } Y = y - a_0 \tag{2.4}$$

3. Inverse Cubic Spline:

The cubic spline equation is

$$S_{i-1}(x) = M_{i-1}\frac{(a_i - x)^3}{6h_i} + M_i\frac{(x - a_{i-1})^3}{6h_i} + \left(y_{i-1} - \frac{M_{i-1}h_i^2}{6}\right)\left(\frac{a_i - x}{h_i}\right) + \left(y_i - \frac{M_i h_i^2}{6}\right)\left(\frac{x - a_{i-1}}{h_i}\right)$$

Where i = 2, 3,……., n

$h_i = a_i - a_{i-1}$, i = 2, 3, 4, …… , n

The $M_i^{'}s$ are determined by $\mu_i M_{i-1} + 2M_i + \lambda_i M_{i+1} = d_i$, i = 2, 3,……,n-1

$$\left.\begin{aligned} 2M_1 + \lambda_1 M_2 &= d_1 \\ \mu_n M_{n-1} + 2M_n &= d_n \end{aligned}\right\} \quad (3.1)$$

Where $\sigma_i = \dfrac{y_i - y_{i-1}}{h_i}$, i = 2,3,……,n

$$\left.\begin{aligned} \lambda_i &= \frac{h_{i+1}}{h_i + h_{i+1}} \\ \mu_i &= 1 - \lambda_i \\ d_i &= \frac{6(\sigma_{i+1} - \sigma_i)}{h_i + h_{i+1}} \end{aligned}\right\} \quad \text{i = 2, 3,……,n-1} \quad (3.2)$$

Let $a_i - x = X_i$

$x - a_{i-1} = h_i - X_i$, i = 2, 3, ….., n

$$S_{i-1}(x) = M_{i-1}\frac{X_i^3}{6h_i} + M_i\frac{(h_i - X_i)^3}{6h_i} + \left(y_{i-1} - \frac{M_{i-1}h_i^2}{6}\right)\left(\frac{X_i}{h_i}\right) + \left(y_i - \frac{M_i h_i^2}{6}\right)\left(\frac{h_i - X_i}{h_i}\right)$$

$$S_{i-1}(x) = \frac{1}{6h_i}\left[M_{i-1}X_i^3 + M_i(h_i - X_i)^3 + \left(6y_{i-1} - M_{i-1}h_i^2\right)X_i + \left(6y_i - M_i h_i^2\right)(h_i - X_i)\right]$$

=

$$\frac{1}{6h_i}\left[(M_{i-1}-M_i)X_i^{\,3}+3M_ih_iX_i^{\,2}-2M_ih_i^{\,2}X_i-M_{i-1}h_i^{\,2}X_i+(6y_{i-1}-6y_i)X_i+6y_ih_i\right]$$

Let

$$M_{i-1}-M_i=\alpha_{i-1}$$
$$y_{i-1}-y_i=\beta_{i-1} \quad , i = 2 , 3 \ldots\ldots,n$$

$$S_{i-1}(x)=$$

$$\frac{1}{6h_i}\left[(\alpha_{i-1})X_i^{\,3}+3M_ih_iX_i^{\,2}+\left\{6\beta_{i-1}-(2M_i+M_{i-1})h_i^{\,2}\right\}X_i+6y_ih_i\right] \quad i = 2, 3 \ldots.,n$$

$$S_{i-1}(x)=\frac{\alpha_{i-1}}{6h_i}X_i^{\,3}+\frac{M_i}{2}X_i^{\,2}+\left[\frac{\beta_{i-1}}{h_i}-(2M_i+M_{i-1})\frac{h_i}{6}\right]X_i+y_i$$

Let $S_{i-1}(x)=y_i+p_iX_i+q_iX_i^{\,2}+r_iX_i^{\,3}$, i = 2, 3,……,n (3.3)

Where $p_i=\left[\frac{\beta_{i-1}}{h_i}-(2M_i+M_{i-1})\frac{h_i}{6}\right]$

$$q_i=\frac{M_i}{2}$$

$$r_i=\frac{\alpha_{i-1}}{6h_i}$$

(3.4)

Applying (2.4) the inverse formula is

$$X_i=\frac{1}{p_i}(y-y_i)-\frac{q_i}{p_i^{\,3}}(y-y_i)^2+\frac{2q_i^{\,2}-p_ir_i}{p_i^{\,5}}(y-y_i)^3 \qquad (3.5)$$

But we have $a_i-x=X_i$

So $x=a_i-X_i$

Thus $S_{i-1}^{-1}(y) = a_i - X_i$ where $y \in [y_{i-1}, y_i]$, i = 2,3,…,n

The inverse cubic spline interpolation formula is $x = S_{i-1}^{-1}(y) = a_i - X_i$ where X_i is given by (3.5), $y \in [y_{i-1}, y_i]$ i = 2,3,……..,n (3.6)

4. Computer Program & Illustration:

A computer program has been developed for inverse cubic spline interpolation using the above formula and is given in the appendix.

To illustrate , Consider the data points (a_i, y_i) = (1,1) , (2,2) , (3,5) , (4,11)

$$h_2 = h_3 = h_4 = 1$$

Using (3.2) $\sigma_2 = 1$, $\sigma_3 = 3$, $\sigma_4 = 6$

$$\lambda_2 = 0.5 , \ \lambda_3 = 0.5$$

$$\mu_2 = 0.5 , \ \mu_3 = 0.5$$

$$d_2 = 6 \ , \ d_3 = 9$$

For natural spline S(x) $M_1 = M_4 = 0$

Using (3.1) $M_2 = 2$ $M_3 = 4$

Using (3.4)
$$p_2 = -1.667 \quad q_2 = 1 \quad r_2 = -0.333$$
$$p_3 = -4.667 \quad q_3 = 2 \quad r_3 = -0.333$$
$$p_4 = -6.667 \quad q_4 = 0 \quad r_4 = 0.667$$

Using (3.5) and (3.6) the inverse cubic spline is

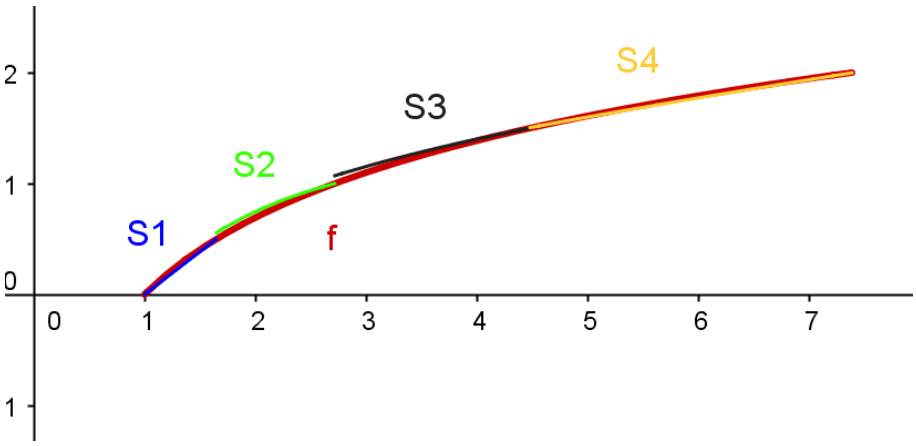
$$S^{-1}(y) = \begin{cases} -0.04312y^3 + 0.0429y^2 + 0.9459y + 0.2817, y \in [1,2] \\ -0.0007y^3 - 0.009y^2 + 0.3584y + 1.5245, y \in [2,5] \\ 0.00034y^3 - 0.0111y^2 + 0.273y + 1.9007, y \in [5,11] \end{cases}$$

Using the computer program $S^{-1}(1.5)$ is evaluated and the output is recorded as given below

```
Enter the number of tabular values : 4
Enter a1:1
Enter a2:2
Enter a3:3
Enter a4:4
Enter y1:1
Enter y2:2
Enter y3:5
Enter y4:11
Enter the y value: 1.5
Using Inverse Spline method, x Value is 1.83333
```

5. Graphical Representation:

The graph of the curve $f(x) = \log x$ in $[1,7.3819]$ and its inverse cubic spline S_1 in $[1,1.6487]$, S_2 in $[1.6487,2.7183]$, S_3 in $[2.7183,4.4817]$ and S_4 in $[4.4817,7.3819]$ is given below



6. Conclusion:

The above graph shows that the inverse cubic splines S_1, S_2, S_3, S_4 of $f(x) = \log x$ derived by (3.6) almost coincides with the actual curve f . Hence the formula (3.6) for inverse cubic spline interpolation gives a fairly accurate value for the argument corresponding to a given entry.

Appendix :

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cassert>
#include <math.h>
#include <fstream>
#include <iterator>
#include <sstream>
#include <stdlib.h>
using namespace std;
class InverseSpline {
public:
        double calculateInverseSpline(std::vector<double> X,
std::vector<double> Y,
                        double y, int n) {
                double inverseSpline = 0.0;
                int interval = findInterval(Y, y);
                if (interval > 1 && interval <= n) {
                        double hValue = calculateHValue(X, interval);
                        double betaValue = calculateBetaValue(Y, interval);

                        double** mValues = calculateMValues(Y, X, n);

                        double alphaValue = calculateAlphaValue(mValues,
interval, hValue,
                                        n);
                        double ri_Value = calculateRiValue(alphaValue,
hValue);
                        double qi_Value = calculateQiValue(mValues,
interval);
                        double pi_Value = calculatePiValue(mValues,
betaValue, hValue,
                                        interval);
```

```
                        double variable1 = (1 / pi_Value)*(y- Y[interval]);
                        double variable2 = (pow((y -
Y[interval]),2))*(qi_Value/(pow(pi_Value,3)));

                        double variable3 = (((2* pow(qi_Value,2)) -
(pi_Value*ri_Value))/(pow(pi_Value,5)))*(pow(y- Y[interval],3));
                        inverseSpline = X[interval] - (variable1 - variable2 +
variable3);

                }
                return inverseSpline;
        }
private:
        int findInterval(std::vector<double> Y, double yValue) {

                std::sort(Y.begin(), Y.end());
                vector<double>::iterator k;
                vector<double>::iterator pos;

                k = upper_bound(Y.begin(), Y.end(), yValue);
                pos = find(Y.begin(), Y.end(), *k);
                return (pos - Y.begin()) + 1;
        }

        double calculateHValue(std::vector<double> X, int index) {
                return X[index] - X[index - 1];
        }

        double calculateBetaValue(std::vector<double> Y, int index) {
                return Y[index - 1] - Y[index];
        }

        double calculateSigmaValue(std::vector<double> Y, int index,
                        double hValue) {
                return ((Y[index] - Y[index - 1]) / hValue);
        }

        double calculateLamdaValue(std::vector<double> X, int index, int n) {
                if (index >= 2 && index < n) {
                        double hValue1 = calculateHValue(X, index);
                        double hValue2 = calculateHValue(X, index + 1);
                        return (hValue2 / (hValue1 + hValue2));
                }
                return 0.0;
```

```
	}

	double calculateMewValue(std::vector<double> X, int index, int n) {
		if (index >= 2 && index < n) {
			return (1 - calculateLamdaValue(X, index, n));
		}
		return 0.0;
	}

	double calculateDValue(std::vector<double> Y, std::vector<double> X,
			int index, int n) {
		if (index >= 2 && index < n) {
			double hValue1 = calculateHValue(X, index + 1);
			double hValue2 = calculateHValue(X, index);
			double sigmaIndexValue1 = calculateSigmaValue(Y,
index + 1,
					hValue1);
			double sigmaIndexValue2 = calculateSigmaValue(Y,
index, hValue2);
			return (6 * (sigmaIndexValue1 - sigmaIndexValue2)
					/ (hValue1 + hValue2));
		}
		return 0.0;
	}

	double calculateAlphaValue(double** mValues, int index, double
hValue,
			int n) {
		if (index > 1 && index < n) {
			return ((mValues[index - 2][0] - mValues[index -
1][0]) / 6 * hValue);
		}
		return 0.0;
	}

	double calculateRiValue(double alphaValue, double hValue) {
		return alphaValue / (6 * hValue);
	}

	double calculateQiValue(double** mValues, int index) {
		return (mValues[index - 1][0] / 2);
	}

	double calculatePiValue(double** mValues, double betaValue, double
```

```
hValue,
                        int index) {
                return ((betaValue / hValue)
                                - (2 * mValues[index - 1][0] + mValues[index
- 2][0])
                                                * (hValue / 6));
        }

        double** calculateMValues(std::vector<double> Y, std::vector<double>
X,
                        int n) {
                double** matrix = createMatrix(Y, X, n);
                double** inverseMatrix = inverseMatrixFunc(matrix, n);
                double** dMatrix = new double*[n];
                for (int i = 0; i < n; i++) {
                        dMatrix[i] = new double[1];
                        if (i + 1 > 1 && i + 1 < n) {
                                double dValue = calculateDValue(Y, X, i + 1,
n);
                                dMatrix[i][0] = dValue;
                        }
                }
                double** mValues = multiplyMatrix(inverseMatrix, dMatrix,
n);
                //mValues[0][0] = mValues[n - 1][0] = 0;
                for (int i = 0; i < n; i++) {
                        for (int j = 0; j < 1; j++) {
                                if (i == 0 || i == n - 1) {
                                        mValues[i][j] = 0;
                                }

                        }
                }
                return mValues;
        }

        double** createMatrix(std::vector<double> Y, std::vector<double> X,
int n) {
                double** matrix = new double*[n];
                for (int i = 0; i < n; i++) {
                        matrix[i] = new double[n];
                        for (int j = 0; j < n; j++) {
                                if (i == j) {
                                        matrix[i][j] = 2;
```

```
                                        } else if (j == i + 1) {
                                                matrix[i][j] =
calculateLamdaValue(X, j, n);
                                        } else if (j == i - 1) {
                                                matrix[i][j] = calculateMewValue(X,
i + 1, n);
                                        } else {
                                                matrix[i][j] = 0;
                                        }
                                }

                        }
                        return matrix;

        }

        double** inverseMatrixFunc(double** matrix, int n) {
                double** inverseMatrix = new double*[n];
                double Det = Determinant(matrix, n);
                double** b = coFactor(matrix, n);
                double** d = Transpose(b, n);
                if (Det == 0.0) {
                        cout << "Inverse does not exist (Determinant=0).\n"
<< endl;
                } else {
                        for (int i = 0; i < n; i++) {
                                inverseMatrix[i] = new double[n];
                                for (int j = 0; j < n; j++) {
                                        inverseMatrix[i][j] = d[i][j] / Det;
                                }
                        }
                }

                return inverseMatrix;
        }
        double Determinant(double **matrix, int n) {
                int i, j, j1, j2;
                double det = 0.0;
                double** m = new double*[n];
                if (n < 1) {  //Error

                } else if (n == 1) {  //Shouldn't get used
                        det = matrix[0][0];
                } else if (n == 2) {
```

```
                    det = matrix[0][0] * matrix[1][1] - matrix[1][0] *
matrix[0][1];
                } else {
                    det = 0;
                    for (j1 = 0; j1 < n; j1++) {
                        m = new double*[n];
                        for (i = 0; i < n - 1; i++)
                            m[i] = new double[n - 1];
                        for (i = 1; i < n; i++) {

                            j2 = 0;
                            for (j = 0; j < n; j++) {
                                if (j == j1)
                                    continue;
                                m[i - 1][j2] = matrix[i][j];
                                j2++;
                            }
                        }
                        det += pow(-1.0, j1 + 2.0)
                                * (matrix[0][j1] *
Determinant(m, n - 1));

                        for (i = 0; i < n - 1; i++)
                            delete m[i];
                        delete m;
                    }
                }

                return (det);
            }

            /*
             Find the cofactor matrix of a square matrix
             */
            double** coFactor(double **matrix, int n) {
                int i, j, ii, jj, i1, j1;
                double det1 = 0.0;
                double** c = new double*[n];
                double** b = new double*[n];
                for (int i = 0; i < n; i++) {
                    b[i] = new double[n];
                }
                c = (double **) malloc((n - 1) * sizeof(double *));
                for (i = 0; i < n - 1; i++)
```

```
                c[i] = (double*) malloc((n - 1) * sizeof(double));

        for (j = 0; j < n; j++) {

                for (i = 0; i < n; i++) {

                        //  Form the adjoint a_ij
                        i1 = 0;
                        for (ii = 0; ii < n; ii++) {
                                if (ii == i)
                                        continue;
                                j1 = 0;
                                for (jj = 0; jj < n; jj++) {
                                        if (jj == j)
                                                continue;
                                        c[i1][j1] = matrix[ii][jj];
                                        j1++;
                                }
                                i1++;
                        }

                        //Calculate the determinate
                        det1 = Determinant(c, n - 1);
                        //Fill in the elements of the cofactor
                        b[i][j] = pow(-1, i + j + 2) * det1;

                }
        }
        for (int i = 0; i < n - 1; i++)
                free(c[i]);
        free(c);

        return b;
}

/*
 Transpose of a square matrix, do it in place
 */
double** Transpose(double **b, int n) {
        int i, j;
        double** d = new double*[n];
        for (int i = 0; i < n; i++) {
                d[i] = new double[n];
        }
```

```
                for (i = 0; i < n; i++) {
                        for (j = 0; j < n; j++) {
                                d[j][i] = b[i][j];

                        }

                }

                return d;
        }

        double** multiplyMatrix(double** matrix1, double** matrix2, int n) {
                double** mValues = new double*[n];
                for (int i = 0; i < n; i++)
                        for (int j = 0; j < 1; j++)
                                for (int k = 0; k < 1; ++k) {
                                        mValues[i] = new double[1];
                                        mValues[i][j] += matrix1[i][k] *
matrix2[k][j];
                                }
                return mValues;
        }
};

int main() {
        double y;
        int n;
        cout << "Enter the number of tabular values : " ;
        cin >> n;
        std::vector<double> X(n), Y(n);

        for (int i = 0; i < n; i++) {
                cout << "Enter a" << i + 1 << ":" ;
                cin >> X[i];

        }
        for (int j = 0; j < n; j++) {
                cout << "Enter y" << j + 1 << ":" ;
                cin >> Y[j];

        }
        cout << "Enter the y value: " ;
        cin >> y;
```

```
        InverseSpline s;

        double output = s.calculateInverseSpline(X, Y, y, n);
        cout << "Using Inverse Spline method, x Value is " << output << endl;

        return 0;
}
```

References

[1]. Anthony Ralston and Philip Rabinowitz, A First Course In Numerical Analysis , Second Edition, Dover Publications,INC.

[2]. E.Ward Cheney, David R. Kincaid, Numerical Methods and Applications, Cengage Learning India Private Limited.

[3]. Kendall E. Atkinson, An introduction to Numerical Analysis, Second Edition

[4]. P. Thangaraj , Computer Oriented Numerical Methods, Prentice Hall of India Private Limited ,2008.

[5]. Weisstein , Eric W, Series Reversion From Mathworld - A Wolfram web resource , http://mathworld.wolfram.com/SeriesReversion.html