A UNIFIED SOFTWARE PACKAGE FOR

CUBIC SPLINE INTERPOLATION

By

PRAVEEN MOTAPOTU

Bachelor of Technology

Jawaharlal Nehru Technological University

Hyderabad, AP, India

2003

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2010

A UNIFIED SOFTWARE PACKAGE FOR

CUBIC SPLINE INTERPOLATION

Thesis Approved:

Dr. John P. Chandler

Thesis Adviser

Dr. K. M. George

Dr. Nohpill Park

Dr. Mark E. Payton

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere gratitude to my graduate adviser Dr. John Chandler for his insightful guidance, constant encouragement, unparalleled support, and immeasurable kindness throughout my thesis work. I honestly appreciate the time he has spent and the efforts he has put, in helping me complete my thesis. This research would not have been possible, if it wasn't for his outstanding advice.

I also wish to express my thanks to Dr. K. M. George and Dr. Nohpill Park for serving on my graduate committee and helping me with their valuable opinions and suggestions.

I am so thankful to my friends who kept their constant encouragement all throughout these times. Their support has helped me to stand tall during the tough times in research. I am also very grateful to have my uncle Venkata Samineni and his family who kept providing me with unconditional care and relief when necessary.

Finally, but never the last, I wish to thank my parents Sri Satyanarayana and Smt Padmavathi for their love, freedom and support provided to me. Their constant words of encouragement, patience, and belief including all kinds of financial and emotional support helped me greatly in my research.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

The term “Spline” comes from the field of mechanics, where a physical spline is the flexible spring steel strip used by draftsmen to go over a set of prescribed points [1]. Weights are attached to that strip at those given data points, so that it is free to slip and straightens out as much as it can. This results in minimizing the stored potential energy ($P.E.$) of the bending. $P.E. = C\int_{-\infty}^{+\infty} K^2(l)dl$ where C is a constant that describes the stiffness of the spring steel strip and the curvature $K = y^{"}/|1 + (y')^2|^{3/2}$. A trace along the spline results in the required physical spline [2]. The shape formed by this spline is natural without any stress between the points. The mathematical model of this physical spline helps us in mechanizing the process, resulting in spline functions. Out of these polynomial spline functions, those functions that have a fixed degree of three are obviously termed as cubic splines. The background of splines along with the cubic equation of the spline between two nodes or knots is discussed in detail at the beginning of Chapter 2.

1.1 Motivation

Every cubic spline has unique boundary conditions, which is explained further in the next chapter. Based on these conditions, it is classified into one of the different types of cubic splines. Currently, cubic spline interpolation is performed independently by using the existing routines in different programming languages. With the constant growing needs of cubic splines in various fields, users would like to interpolate their data and compare the resulting splines based on the different types of cubic splines. However, there is no known software that interpolates all the different types of cubic splines for a given set of data. This difficulty led me to the development of a unified software package for cubic spline interpolation.

1.2 Cubic Spline Terminology

As mentioned earlier, cubic splines are classified into different types. Some of them being discussed include *Natural cubic spline, Complete cubic spline, Clamped cubic spline, Not-a-knot cubic spline,* and *Shape-preserving cubic spline*. Based on the specified *end conditions*, they are mostly categorized into one of these types. In order to interpolate a cubic spline, some data is necessary consisting of points and their function values, commonly known as *nodes* or *knots*.

The initial process of cubic spline interpolation is to calculate the *coefficients* of the cubic spline which include all the coefficients of the linear, quadratic and cubic terms of the cubic equation. This is achieved after eventually solving a *tridiagonal* system, which is explained in detail in the next chapter. Evaluating the first and second

derivatives of the cubic spline is also necessary, depending on the requirements and the type of the cubic spline being interpolated. The final step is to evaluate the spline value at any specified point within the range of nodes, by using the spline coefficients calculated in the previous step. Obviously, at any given node, the spline value is equal to the actual function value specified in the data set.

1.3 Thesis Outline

A detailed literature review along with the background of cubic splines is discussed in Chapter 2. It also covers the various types of cubic splines, and their classification based on the various end conditions. The description of the problem being solved and the development of the code to perform cubic spline interpolation are explained in Chapter 3. The whole process of interpolating the cubic spline including all the necessary steps along with their respective Java routines follows the development. The various different tests performed along with the results obtained comprise the majority of Chapter 4. The various graphs of cubic spline curves obtained for some common functions such as $\sin x$, $\cos x$, e^x and $\frac{1}{(1+x^2)}$ are explained by comparing the different types of the cubic splines. Finally, the conclusions of the thesis and the scope for future work are laid out in Chapter 5.

CHAPTER II

LITERATURE REVIEW

2.1 Background of Splines

If we represent the spline with $s(x)$, then the second derivative $s''(x)$ of the spline approximates the curvature and dx approximates the differential arc length [3]. Thus the stored potential energy of such a *linearized spline* is proportional to $\int s''(x)^2\, dx$. When the knots $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ are given, the linearized interpolating spline $s(x)$ is such that $s(x_i) = y_i$ $(i = 1, 2, \ldots, n)$ and such that $\int_{x_1}^{x_n} (s''(x))^2\, dx$ is minimized. Additionally, $s(x)$ is a cubic polynomial between each adjacent pair of knots, and adjacent polynomials join continuously with continuous first and second derivatives of the spline at the knots. Hence, the cubic spline equation between two knots is defined as

$$s(x) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \quad x_i \le x \le x_{i+1}. \text{ [3]}$$

where b_i, c_i, and d_i, $i = 1, 2, \ldots, n - 1$ are the coefficients that are calculated and stored for the evaluation of the spline values later on.

As the end conditions determine the type of a cubic spline, the scope for having different end conditions can be explained with the help of the parameters of a cubic spline curve. There will be $n-1$ intervals between the n nodes and so there will be $n-1$ cubic equations with 4 parameters each, making the total number of parameters that need to be determined as $4n-4$ [3]. As stated earlier, since the spline has continuous first and second derivatives at each of the $n-2$ interior nodes, every x_i has $3(n-2)$ conditions on s. As $s(x_i)=y_i$ for each of the n nodes, adding that many conditions makes it $4n-6$ parameters. Hence, all we need is two more conditions to completely determine the cubic spline, and these are obviously known as *end conditions*.

2.2 Natural cubic spline

The two end conditions that define the natural cubic spline are $s''(x_1)=0$ and $s''(x_n)=0$ [3]. It got its name because of its similarity to the physical spline used by draftsmen in the mechanical field. The physical spline straightens out as much as it can at the last points. In a similar manner, the natural cubic spline also becomes a straight line at the end nodes because its second derivatives are zero [4].

Shampine and Allen [2] have stated the following theorem to describe the smoothness of the spline by considering the natural cubic spline.

Theorem: Let f be twice continuously differentiable on $[a,b]$ and let $s(x)$ be the natural cubic spline interpolating $f(x)$ at $a=x_0<x_1<\ldots x_n=b$. If we define

$$h=\max_{0\le i\le n-1}(x_{i+1}-x_i),$$

then

$$\max_{a\le x\le b}|f(x)-s(x)| \le h^{3/2}\left\{\int_a^b [f''(t)]^2 dt\right\}^{1/2},$$

$$\max_{a\le x\le b}|f'(x)-s'(x)| \le h^{1/2}\left\{\int_a^b [f''(t)]^2 dt\right\}^{1/2}.$$

This theorem tells us that with the increase in the number of nodes, s' and s converge uniformly to f' and f respectively, for all $x \in [a, b]$. Because s is so smooth, the natural cubic spline is very useful for interpolation and numerical differentiation.

2.3 Complete and clamped cubic splines

The complete cubic spline is defined by the end conditions $s'(x_1) \approx f'(x_1)$ and $s'(x_n) \approx f'(x_n)$. But, instead of requesting the values $f'(x_1)$ and $f'(x_n)$, the four data points nearest each end are interpolated with the cubics and their slopes are used in solving the required tridiagonal system [4].

Clamped cubic spline uses the same end conditions as the complete cubic spline which are $s'(x_1) = f'(x_1)$ and $s'(x_n) = f'(x_n)$. However, the clamped cubic spline requests those $f'(x_1)$ and $f'(x_n)$ values from the user, instead of using the approximations of the complete cubic spline.

Even though both of these cubic splines are dependent on the set of end conditions, their interpolated spline curves are distinct in their own respects. This is due

to the fact that, in a clamped cubic spline the end conditions are the first derivatives of the actual function, which is not the case for the complete cubic spline.

2.4 Not-a-knot cubic spline

The not-a-knot cubic spline utilizes the condition that the third derivative of the spline is continuous at x_2 and x_{n-1} [5]. By the name itself, it indicates that nothing is specified at the traditional end points other than the fact that this spline interpolates the data at the end points.

Behforooz [6] presented an idea to extend the concept of not-a-knot spline to all the interior knots and to obtain a piecewise interpolatory cubic polynomial. It serves as a shortcut by eliminating the necessity of solving the tridiagonal system. However, for more general purposes, the conventional not-a knot cubic spline is implemented here.

CHAPTER III

DEVELOPMENT OF THE SOFTWARE PACKAGE

3.1 Problem of distinct and independent routines

In order to interpolate a cubic spline for given data, users have to use one of the existing routines for different types of splines written in various different languages. This procedure is obviously very tedious and difficult. First of all, the users need to modify their data in order to match the input parameters of those routines. Even after interpolating the spline, the results will be outputted in different forms. So, the comparison of those obtained results will be another major task.

Shampine and Allen [2] have presented subroutines in the FORTRAN language to evaluate the natural cubic spline. One of the input parameters for them is "Index" which has the order of nodes in increasing order. However, Shampine, Allen and Pruess [4] have provided the subroutines in C language to evaluate the complete cubic spline. They have a parameter called "Flag", which reports the status of the order of nodes and the invalid number of nodes. Bradie [7] provided the archives of source code for clamped and not-a-knot cubic splines. But it has all the interpolation routines, both useful and

ignorable types under one header file. All these instances provide a clear picture of the problem being faced and project light into the need for a unified software package.

3.2 Software package development

The unified software package for the cubic spline interpolation in various types is developed using the Java language. The entire source code present in the package, including the main driver routine is listed in Appendix A. In general, the process of interpolating a cubic spline is mainly divided into two major steps. The first one is calculating the coefficients of the cubic spline, whereas the second one is evaluating the spline at the required point using those coefficients.

3.2.1 Calculation of spline coefficients

The routine *cubicSplineCoeffs* calculates the various coefficients required for the cubic spline interpolation. It is called only once for each set of data. It takes in the data consisting of the number of nodes, the interpolating points and their respective function values, along with the type of cubic spline. For the clamped cubic spline, it will also utilize the two input parameters which are the first derivatives of the actual function at the end points. The routine then computes the tridiagonal system, if required, and then eventually calculates the coefficients of the linear, quadratic and the cubic terms of the cubic spline.

Depending upon the type of the cubic spline, this routine calls one of the respective subroutines and deals with the parameter requirements for them accordingly. The main advantage of having one routine for all the types of cubic splines is to avoid the difficulties in dealing with various parameters. In order to calculate, the user needs to supply the necessary parameters and call this routine only once.

3.2.2 Evaluation of the spline

The routine that evaluates the value of the spline for a give point is *cubicSplineValue*. This routine is called several times, once for each point where the spline is to be interpolated. It receives the nodes, the spline coefficients, the type of the cubic spline, and the point of spline evaluation as inputs. Using all of these, it calculates the value of the cubic spline.

Depending upon the number of nodes and the distance between them, a user can either call this routine less or more frequently. For example, if the number of nodes is as low as 5 along with a small range for these nodes, then the user might like to evaluate the spline nearly 20 times between a pair of consecutive nodes, in order to obtain a smooth cubic spline. However, it might be redundant to call this routine that many number of times if the number of nodes itself is a relatively large number such as 17. In that case, evaluating the spline nearly 5 times between a pair of consecutive nodes might still result in a smooth spline.

3.3 Approach

Initially, all the existing subroutines for the various different types of cubic splines are collected. They are then converted into Java, without the loss of methodology. The top level routines *cubicSplineCoeffs* and *cubicSplineValue* are written, incorporating the different individual subroutines into them. The parameters for these two routines are laid down in such a way that, they work for any type of the four implemented types of cubic splines.

The main routine or the driver for this application is constructed with the objective of interpolating the cubic splines for the four functions: $\sin x$, $\cos x$, e^x and $\frac{1}{(1+x^2)}$. These four functions with specific ranges of nodes provide us with a better understanding of cubic spline interpolation. The comparison between the four implemented types of cubic splines, using these four functions is carried out in detail in the next chapter.

CHAPTER IV

TESTS AND RESULTS

4.1 Testing the software package

The software package for cubic spline interpolation, that is developed using Java, is tested for the following four basic functions: $\sin x$, $\cos x$, e^x and $\frac{1}{(1+x^2)}$. The sine and cosine functions are considered as the two fundamental functions when interpolating cubic splines. The fourth function $\frac{1}{(1+x^2)}$ is a famous problem used by Carl Runge [8] to demonstrate that interpolation of a function $f(x)$ by polynomials of increasing degree on a set of uniformly spaced x, does not necessarily converge uniformly to the function being interpolated. A spline, being of fixed degree, does not suffer from this "Runge's phenomenon" [8]. So, some points for all these functions are given as the input data points for the software. The software can be modified with minimal efforts to include more functions or more data points. For a better understanding, the range of $\sin x$ is taken from $[0, \pi]$, $\cos x$ from $[0, \pi]$, e^x from $[0, 1]$, and $\frac{1}{(1+x^2)}$ from $[-5, 5]$. In the tests performed for each function, the number of nodes is taken as 17 $(2^4 + 1)$, which represents the typical number of data points for a typical problem of spline interpolation.

4.2 Results

The results of the software with different types of splines and various functions are explained with the help of graphs. These graphs help us in the comparison of the cubic splines in a convenient way. For each case, two graphs are displayed, one for the cubic spline and the other one for the error between the spline value and the actual function value.

In all the graphs, the curve of the actual function and the curve of the cubic spline almost overlap. The difference between those curves is only visible in the second graphs plotted. In these difference graphs, the difference value on Y axis is extremely low, when compared to the spline values on Y axis in the actual spline graphs. The following sections contain all the various graphs plotted for a typical input of 17 nodes. But, various tests were also performed for different numbers of input nodes, which were not of much importance at this point.

4.3 Sin x function interpolation with 17 nodes

The next four sub sections explain the interpolation of the four different types of splines for the $\sin x$ function, when 17 node values of the function are given as input to the software. Each sub section has two graphs, one for the spline values and function values and the other one showing the difference between these two values. From the second graphs, we can say that natural cubic spline and clamped cubic spline look similar, whereas complete cubic spline and not-a-knot cubic spline look closer.

4.3.1 Natural cubic spline for sin x function

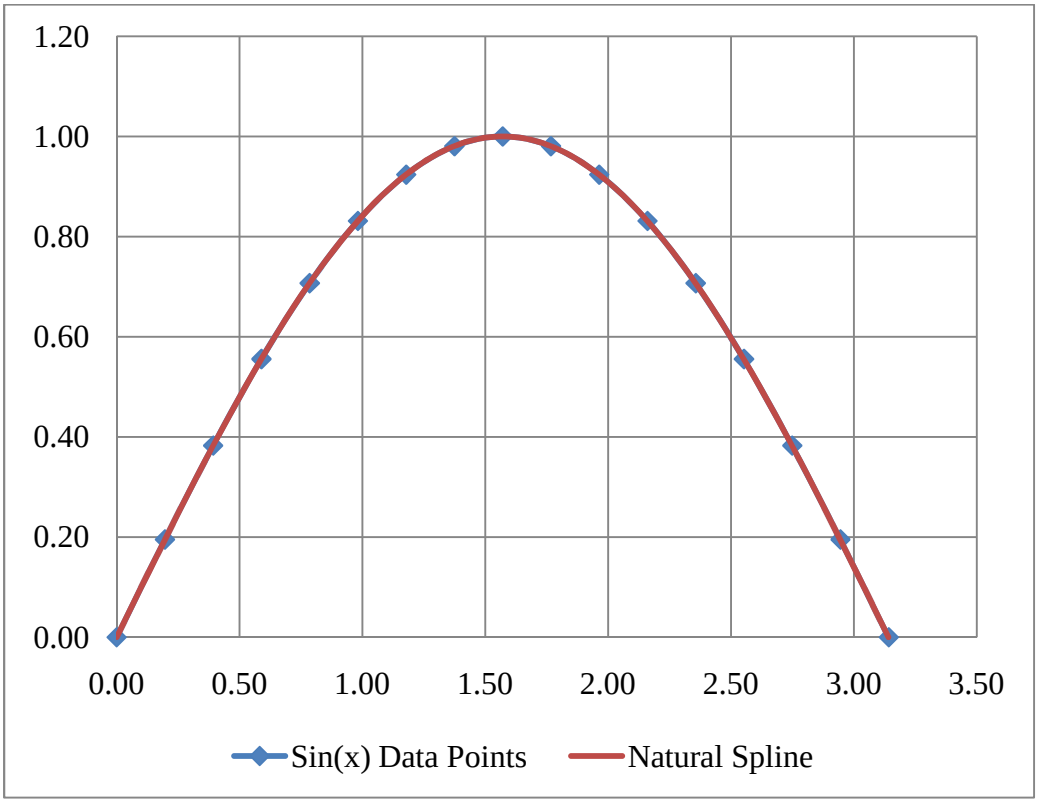

Figure 4.1: Natural cubic spline for sin x with 17 nodes between [0, π]

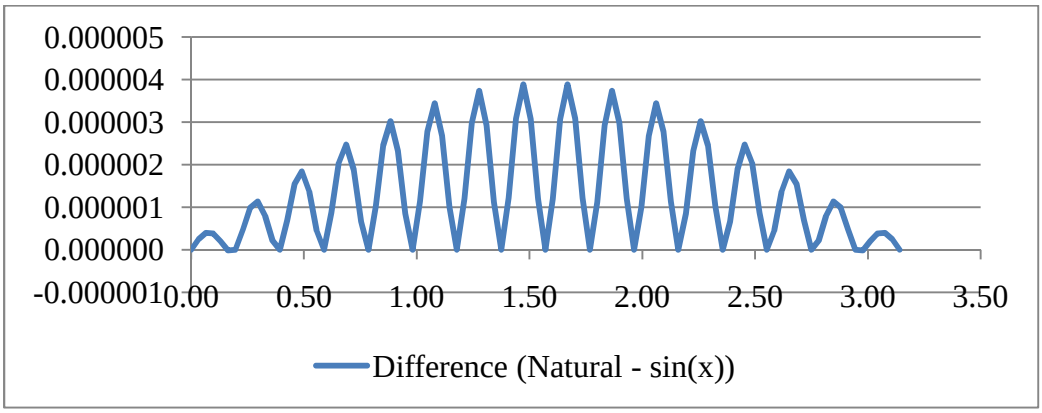

Figure 4.2: Difference between natural spline value and sin x for Figure 4.1

4.3.2 Complete cubic spline for sin x function

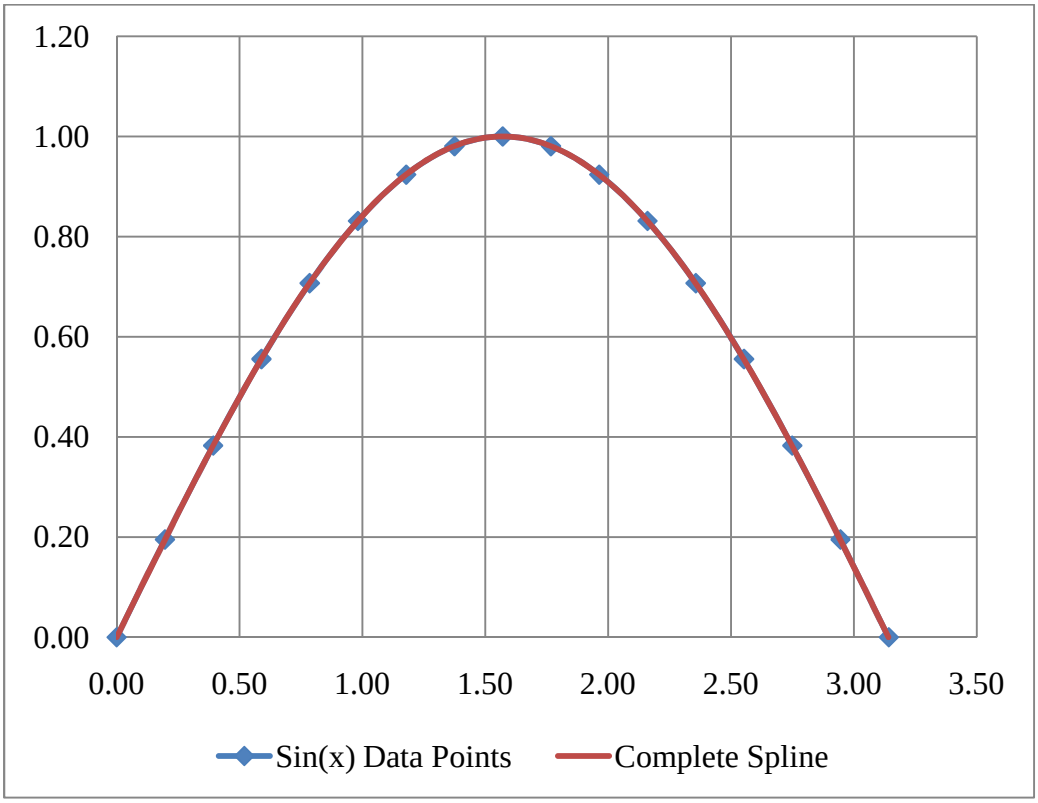

Figure 4.3: Complete cubic spline for sin x with 17 nodes between [0, π]

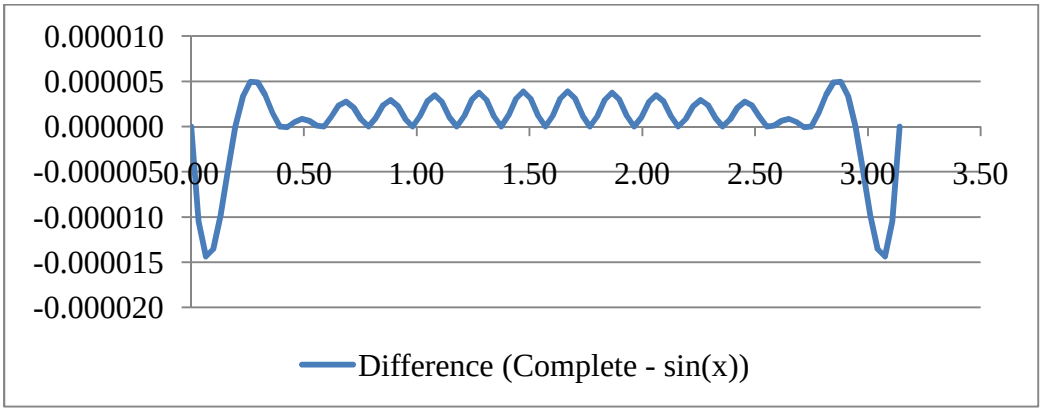

Figure 4.4: Difference between complete spline value and sin x for Figure 4.3

4.3.3 Clamped cubic spline for sin x function

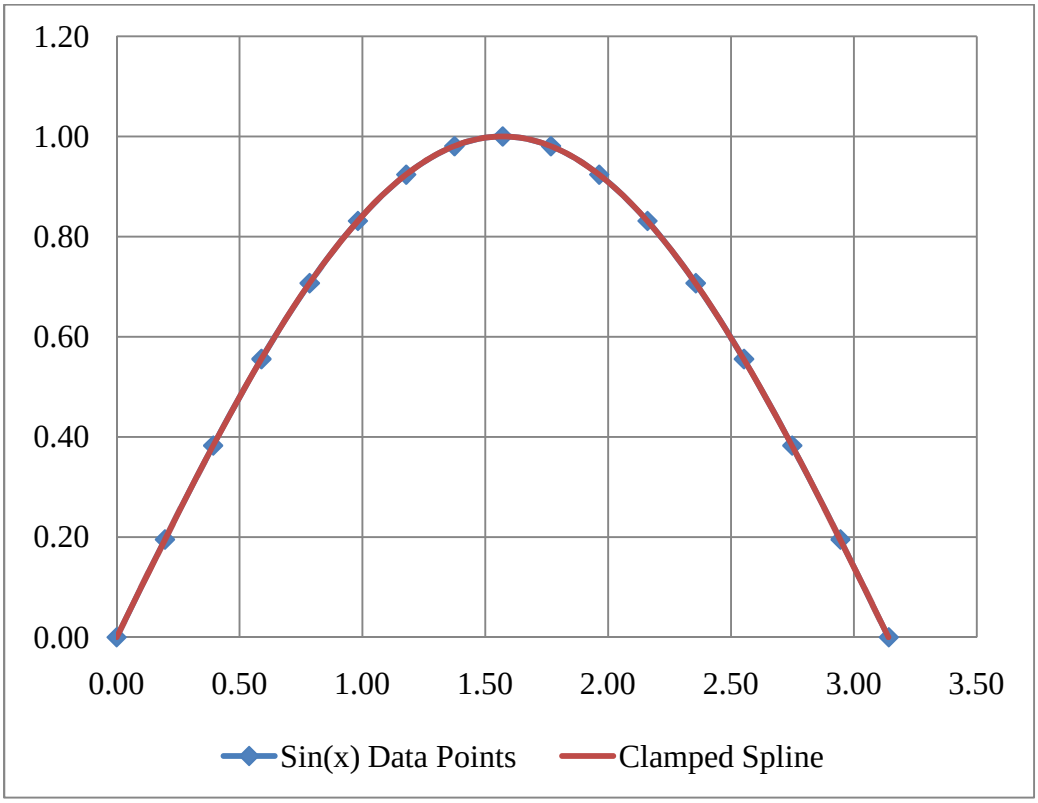

Figure 4.5: Clamped cubic spline for sin x with 17 nodes between [0, π]

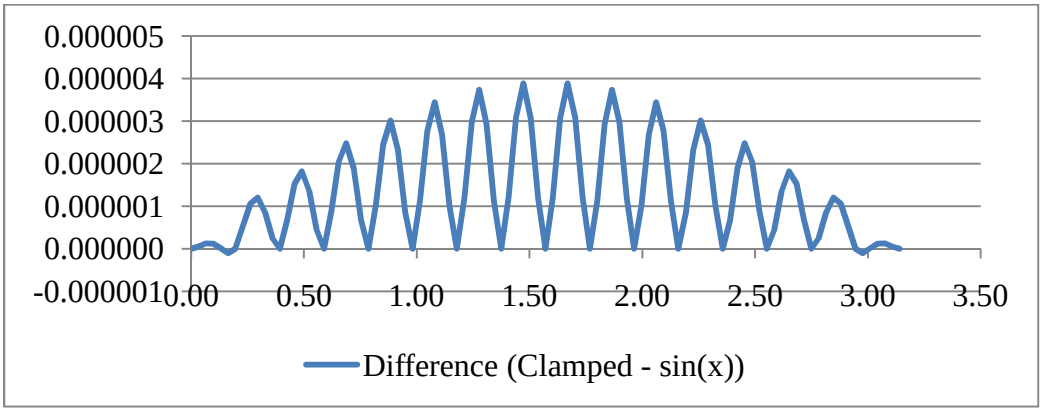

Figure 4.6: Difference between clamped spline value and sin x for Figure 4.5

4.3.4 Not-a-knot cubic spline for sin x function

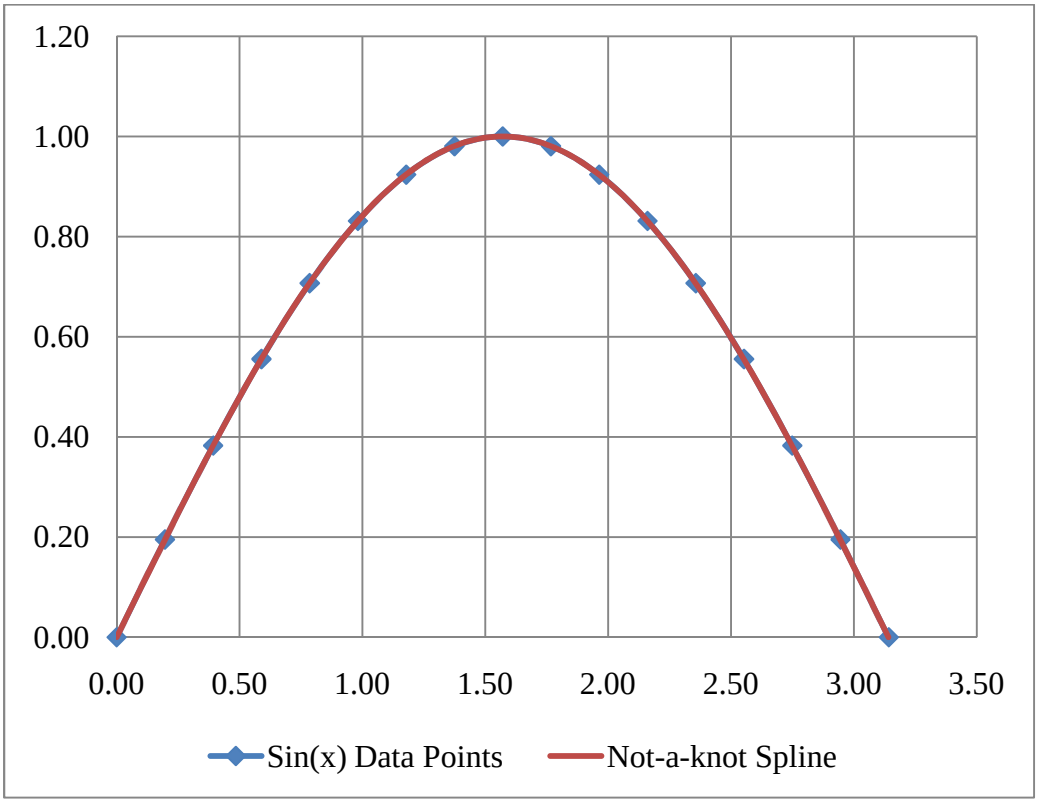

Figure 4.7: Not-a-knot cubic spline for sin x with 17 nodes between [0, π]

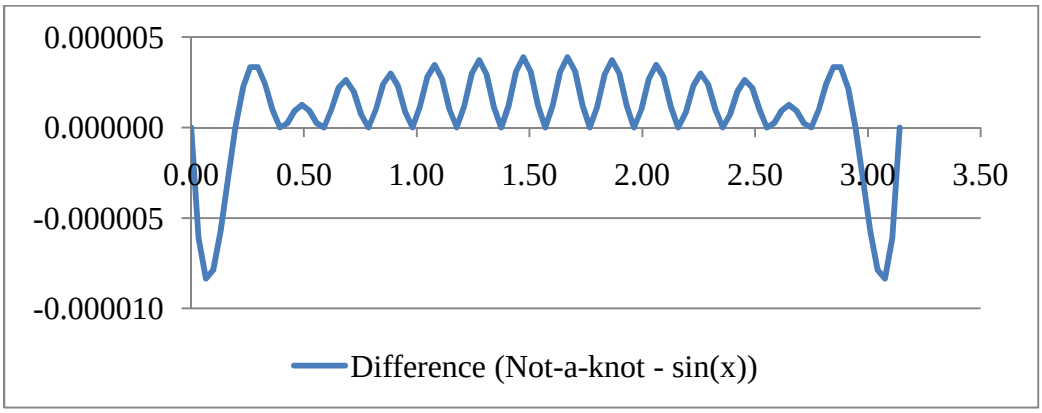

Figure 4.8: Difference between not-a-knot spline value and sin x for Figure 4.7

For $\sin x$ function, out of the 4 splines, both the natural cubic spline and the clamped cubic spline are the closest to the actual function. The maximum difference they achieves is less than 4.0×10^{-6}, as we can see from Figures 4.2 and 4.6 above.

4.4 Cos x function interpolation with 17 nodes

These next four sub sections explain the four types of splines for the $\cos x$ function, when 17 node values of the function are given as input to the software. The complete spline and the not-a-knot spline are similar in this case. The natural cubic spline is different at the ends when compared to those two splines. However, the clamped cubic spline is quite different to all the above three splines.

4.4.1 Natural cubic spline for cos x function

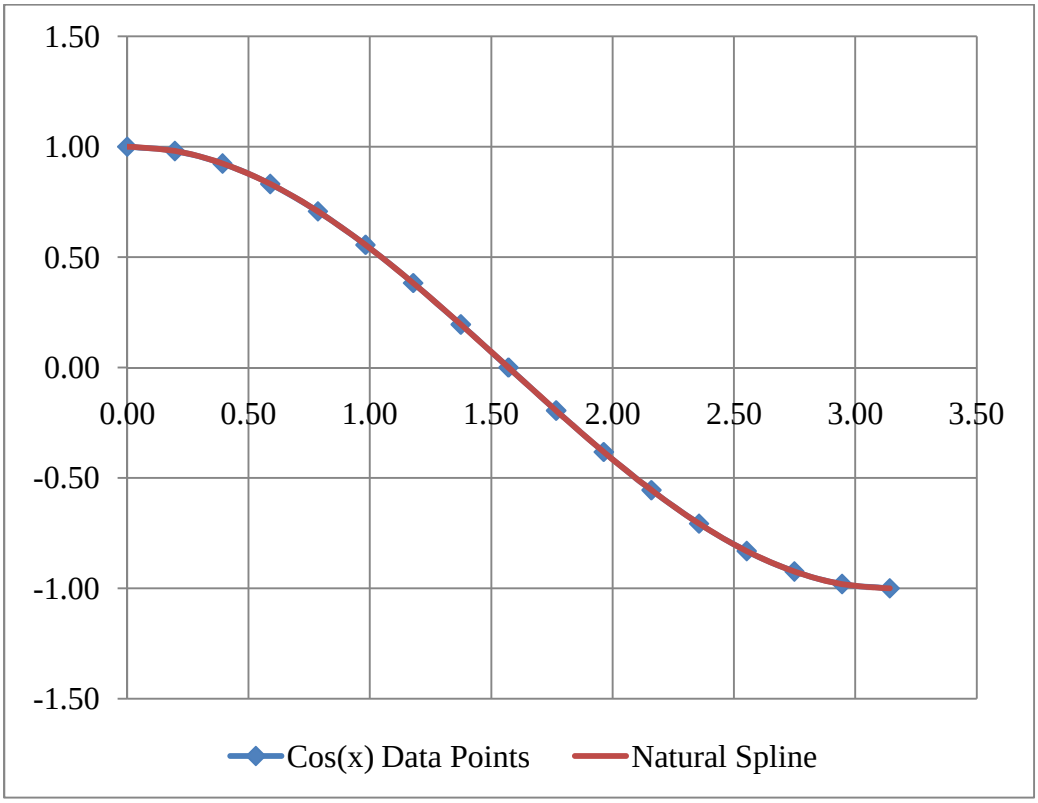

Figure 4.9: Natural cubic spline for cos x with 17 nodes between [0, π]

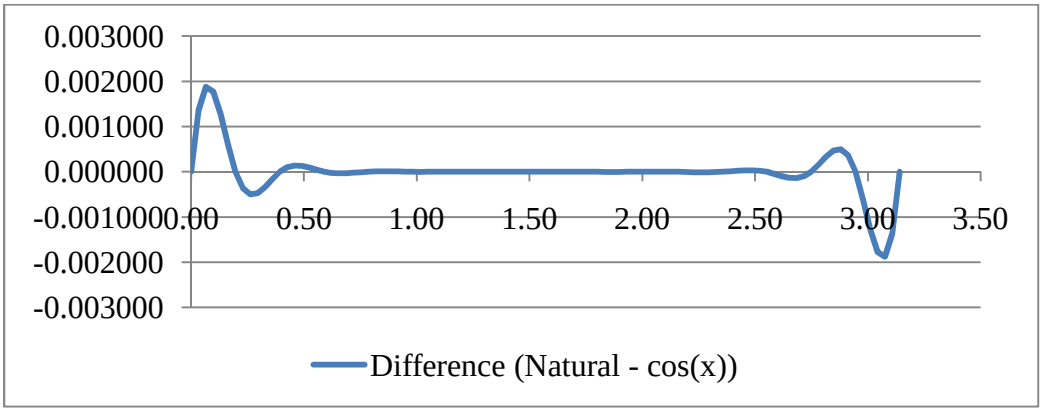

Figure 4.10: Difference between natural spline value and cos x for Figure 4.9

4.4.2 Complete cubic spline for cos x function

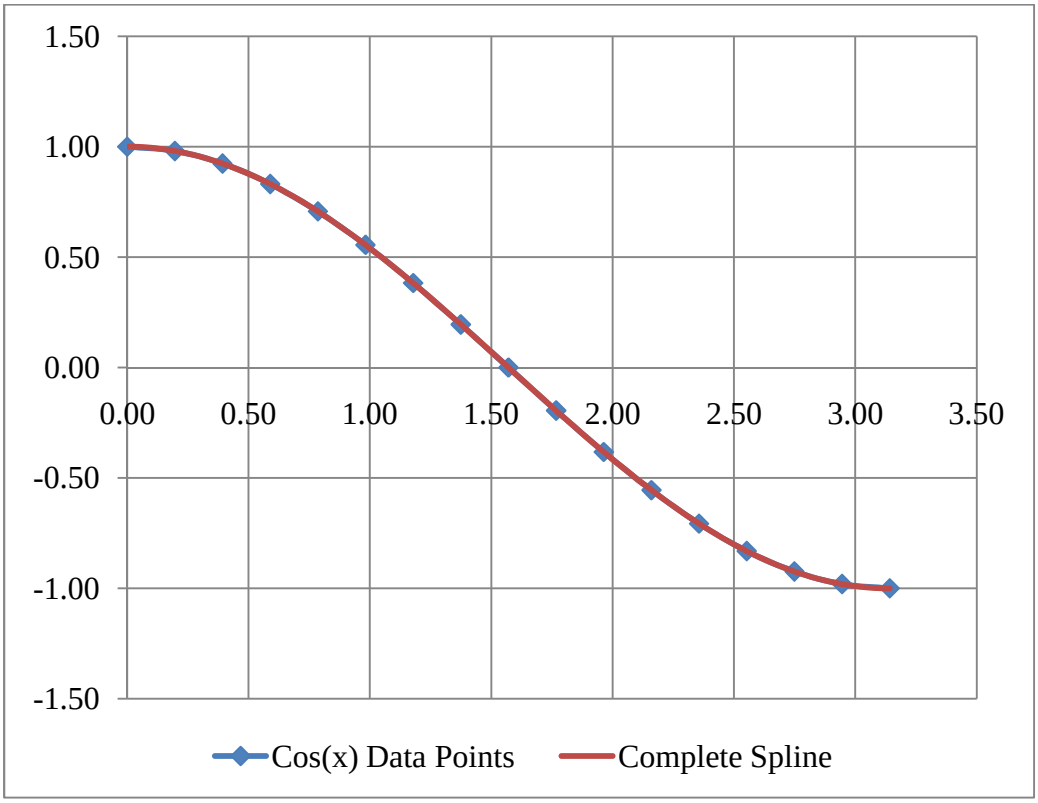

Figure 4.11: Complete cubic spline for cos x with 17 nodes between [0, π]

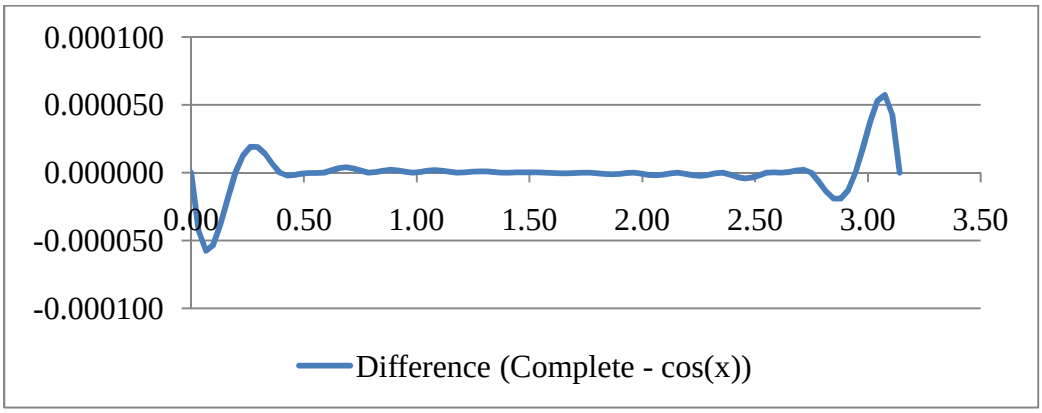

Figure 4.12: Difference between complete spline value and cos x for Figure 4.11

4.4.3 Clamped cubic spline for cos x function

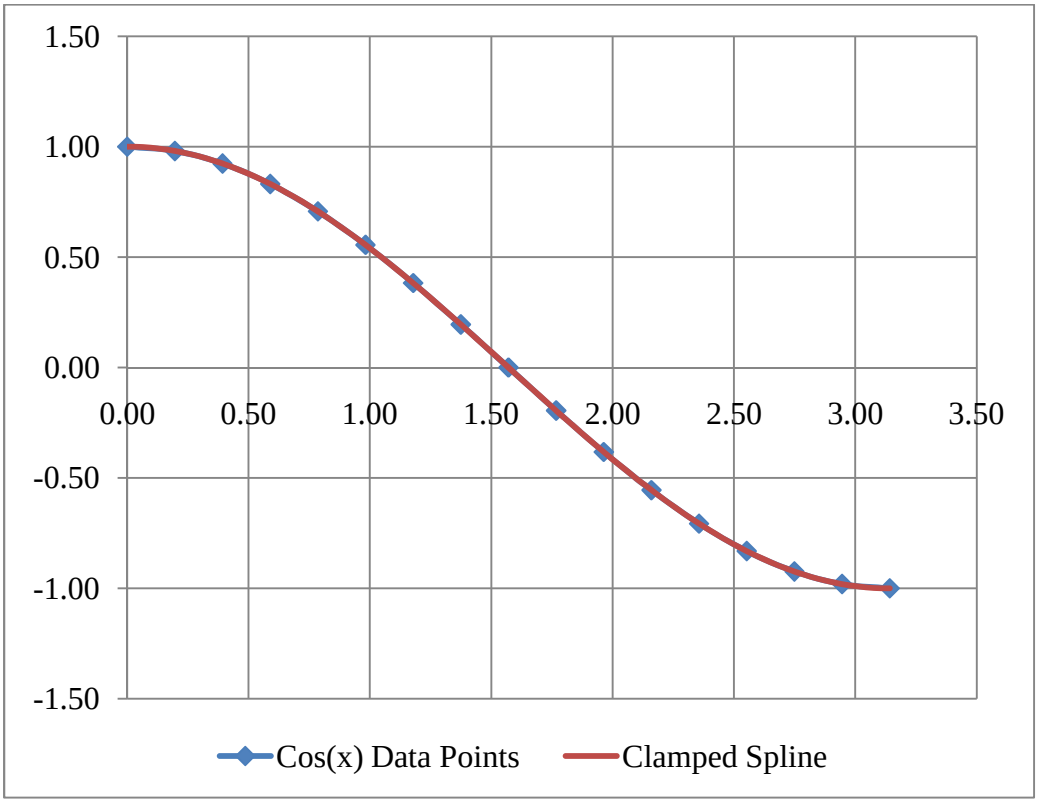

Figure 4.13: Clamped cubic spline for cos x with 17 nodes between [0, π]

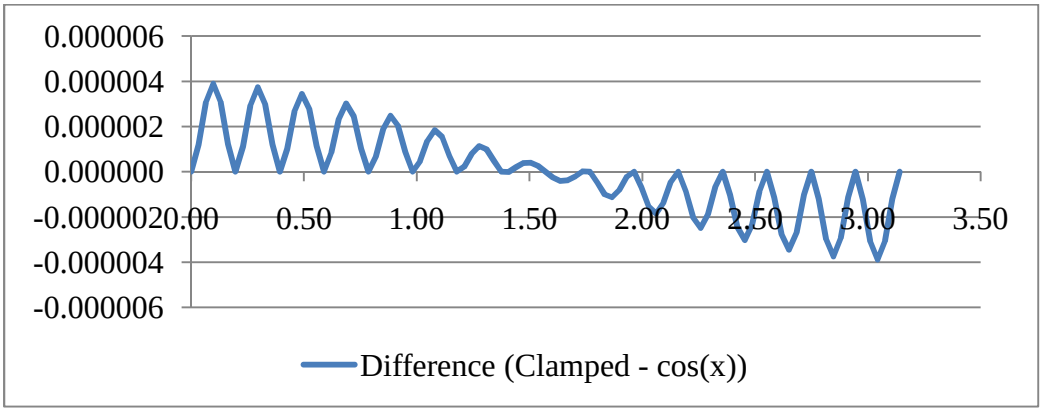

Figure 4.14: Difference between clamped spline value and cos x for Figure 4.13

4.4.4 Not-a-knot cubic spline for cos x function

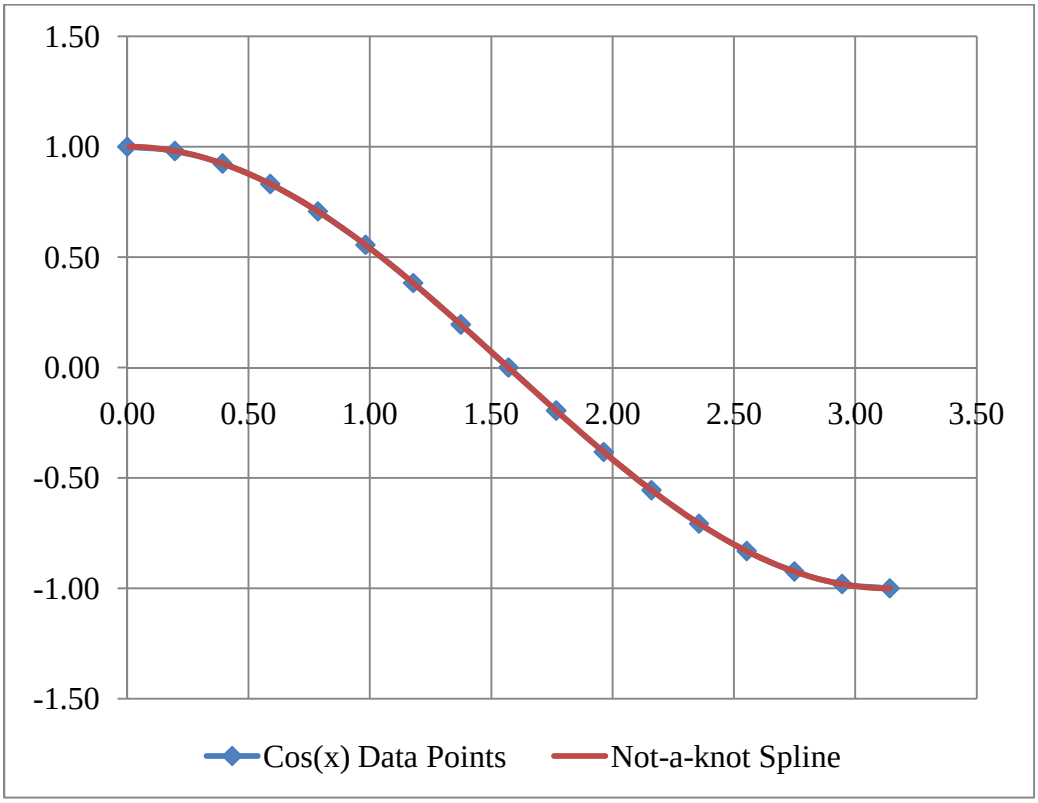

Figure 4.15: Not-a-knot cubic spline for cos x with 17 nodes between [0, π]

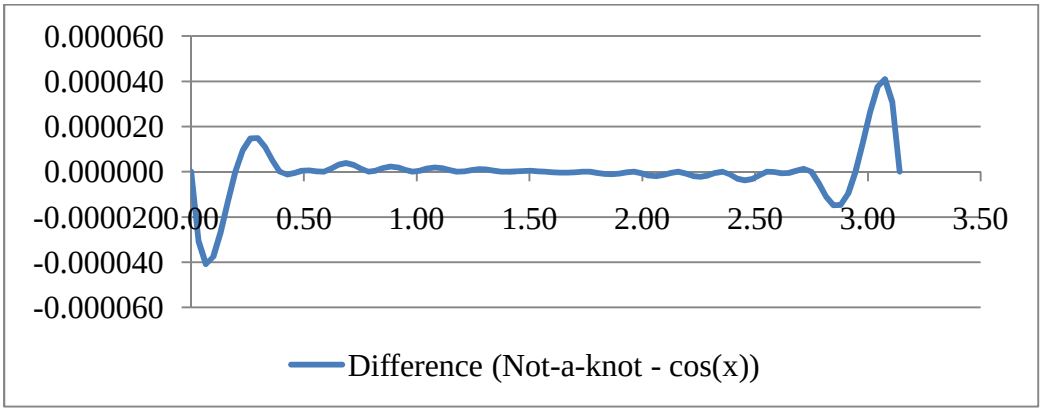

Figure 4.16: Difference between not-a-knot spline value and cos x for Figure 4.15

Out of the 4 splines, the clamped cubic spline is the closest to the actual function. Its maximum difference is less than 4.0×10^{-6}, as we can see from Figure 4.14 above. The natural cubic spline differs greatly at the edges differing by as much as 2.0×10^{-3}. This can be due to the fact that the second derivatives at the edges are considered to be zero, which is the opposite of the actual values.

4.5 e^x function interpolation with 17 nodes

The next four sub sections explain the four types of splines for the e^x function, when 17 node values of the function are given as input to the software. Surprisingly, the clamped cubic spline is different from the rest of the three types of splines. It is also more close to the function except at the ends.

4.5.1 Natural cubic spline for e^x function

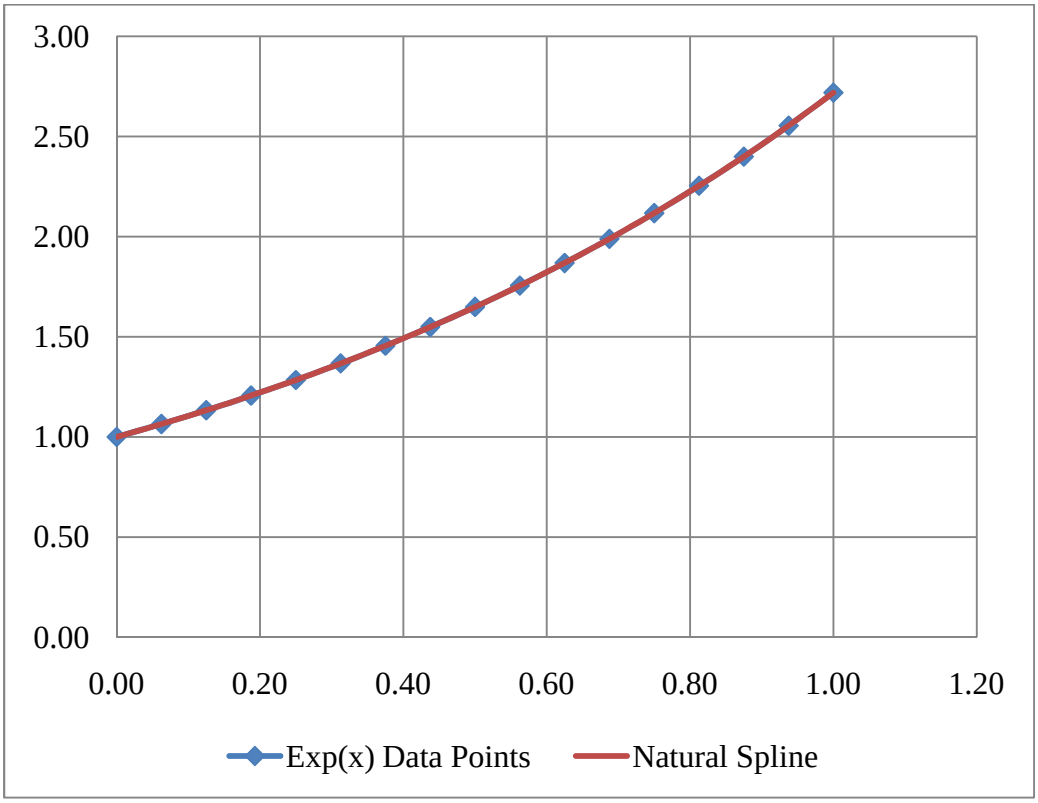

Figure 4.17: Natural cubic spline for e^x with 17 nodes between [0, 1]

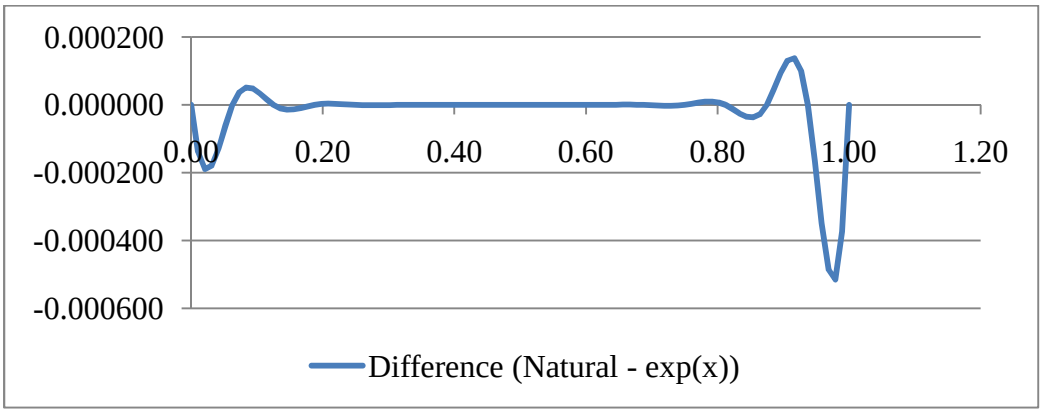

Figure 4.18: Difference between natural spline value and e^x for Figure 4.17

4.5.2 Complete cubic spline for e^x function

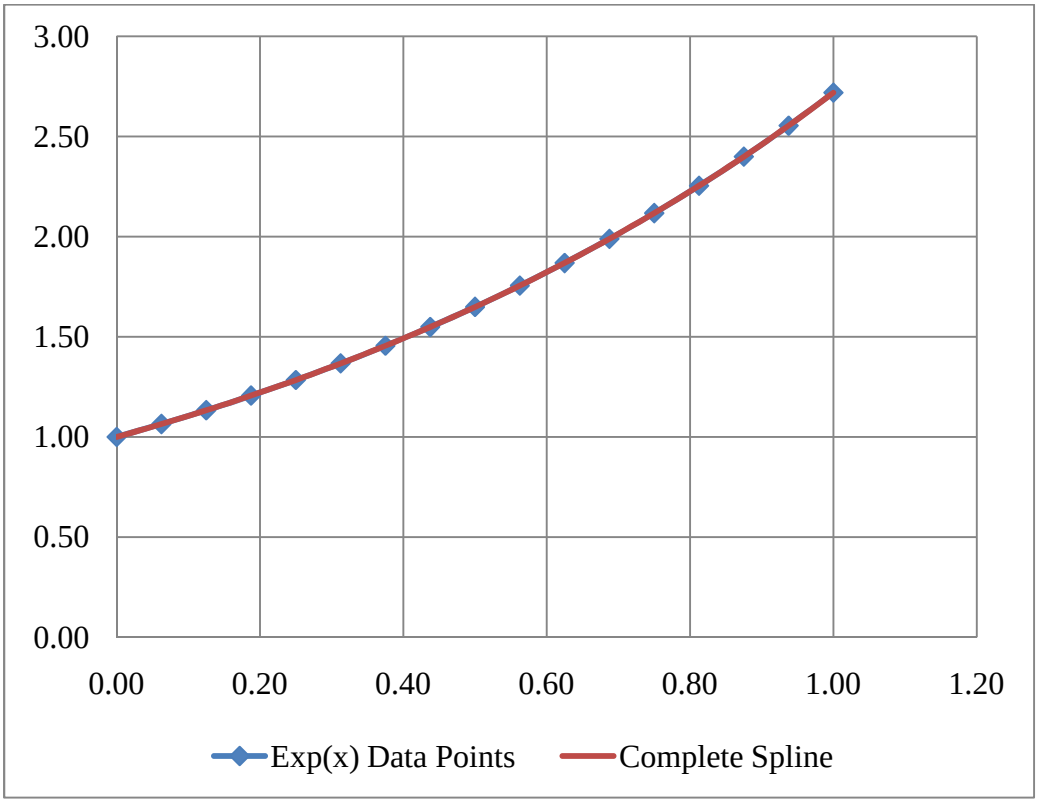

Figure 4.19: Complete cubic spline for e^x with 17 nodes between [0, 1]

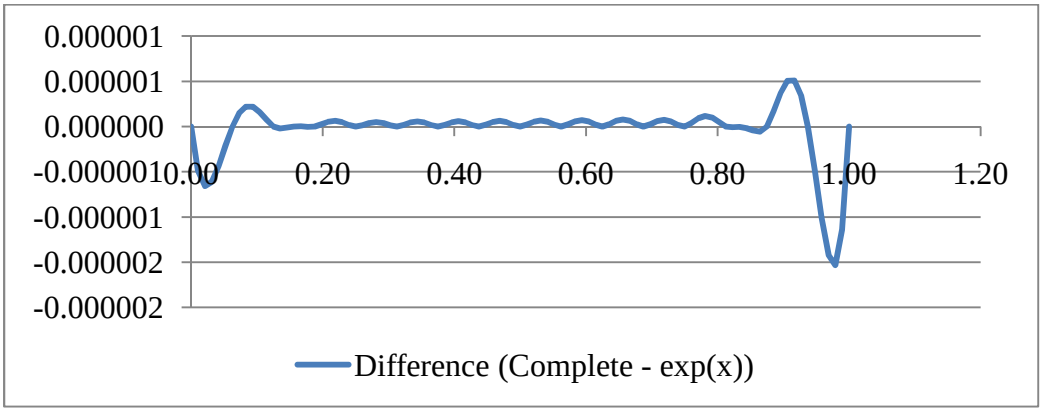

Figure 4.20: Difference between complete spline value and e^x for Figure 4.19

4.5.3 Clamped cubic spline for e^x function

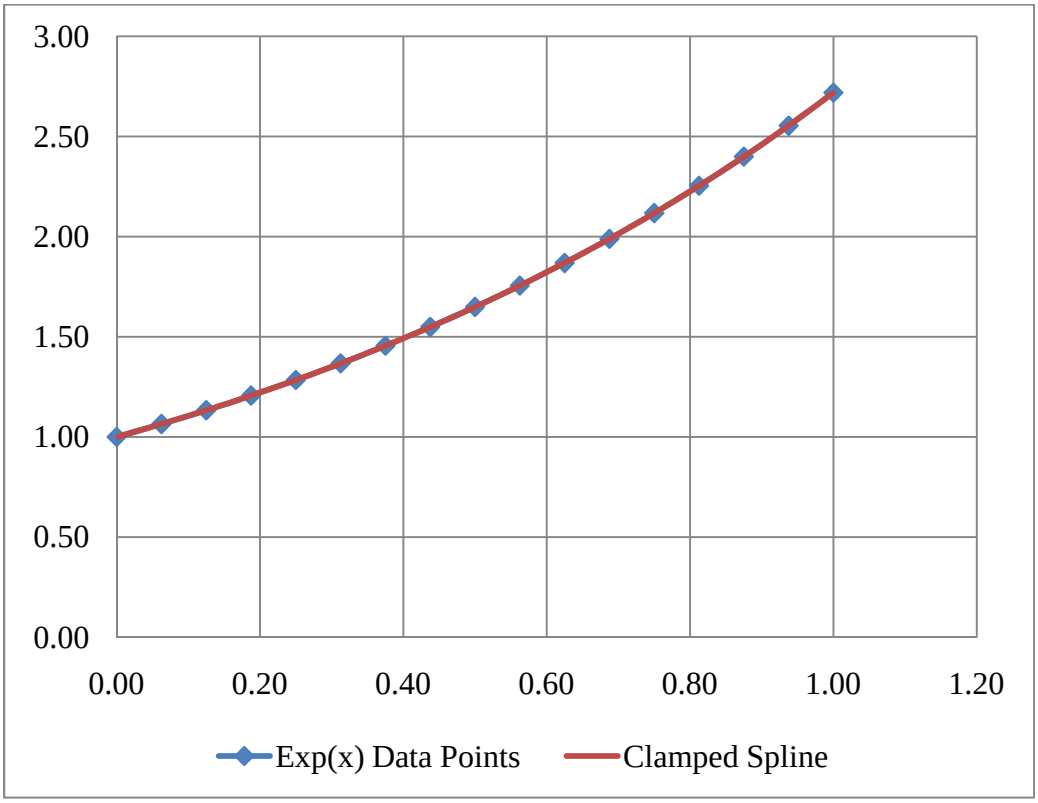

Figure 4.21: Clamped cubic spline for e^x with 17 nodes between [0, 1]

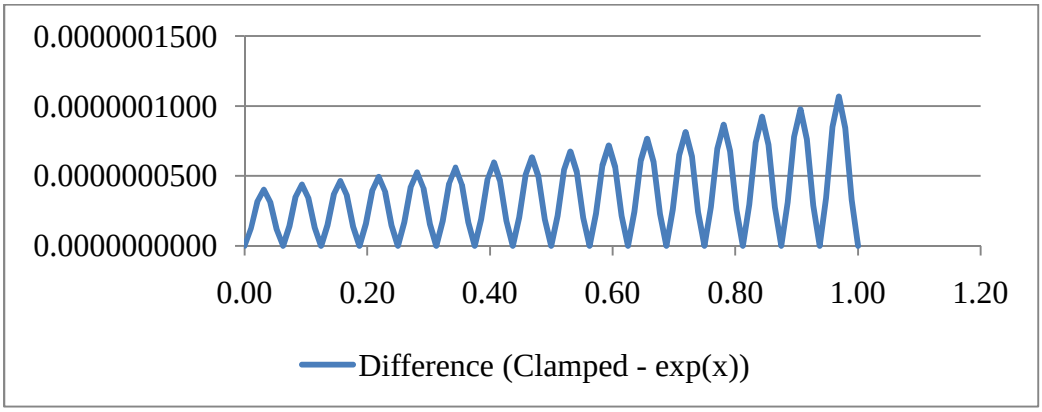

Figure 4.22: Difference between clamped spline value and e^x for Figure 4.20

4.5.4 Not-a-knot cubic spline for e^x function

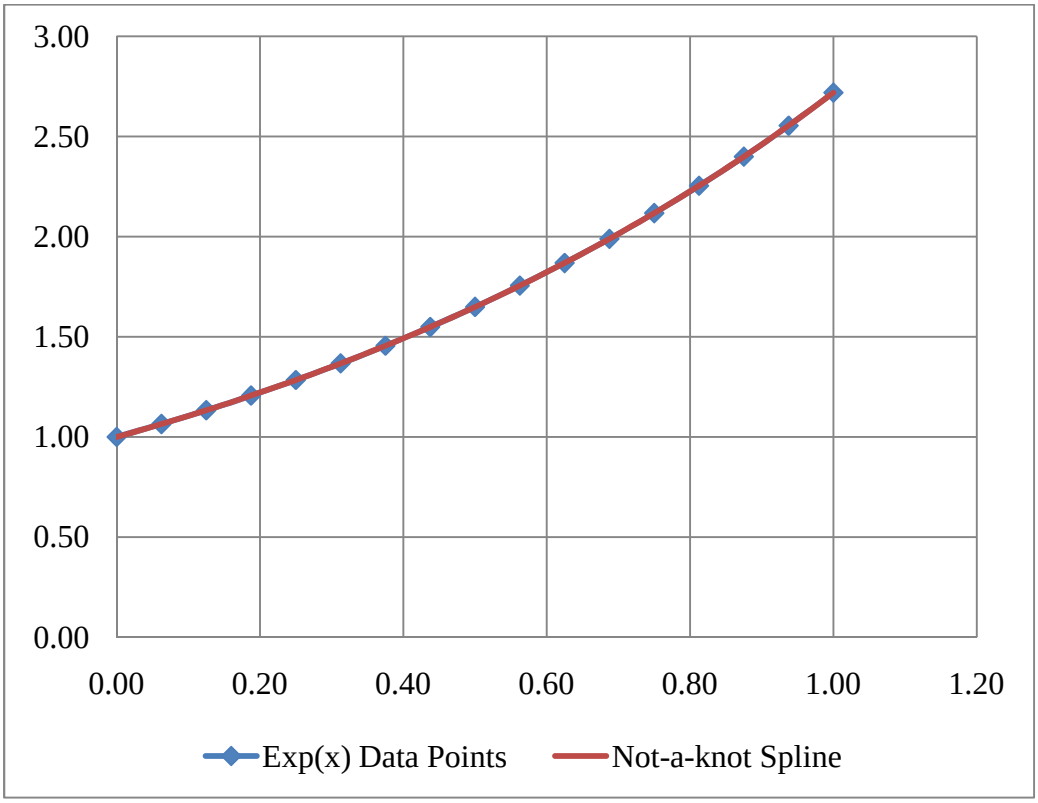

Figure 4.23: Not-a-knot cubic spline for e^x with 17 nodes between [0, 1]

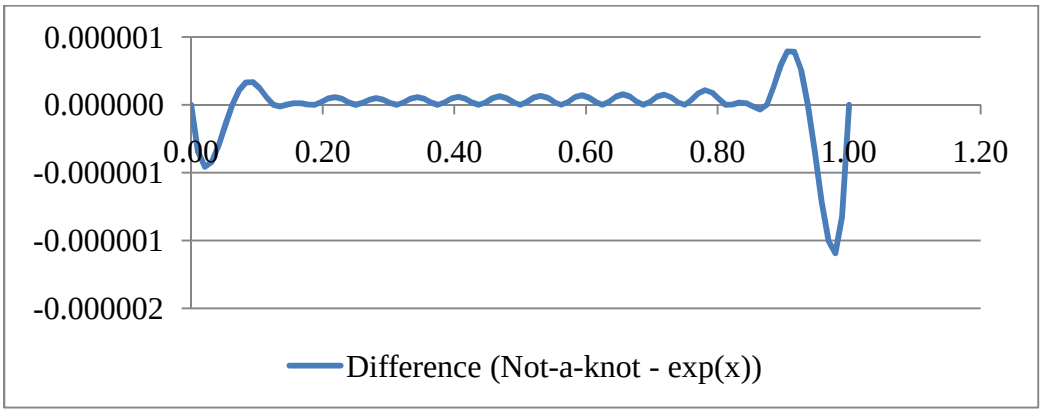

Figure 4.24: Difference between not-a-knot spline and e^x for Figure 4.23

In this case, out of the 4 splines, the clamped cubic spline is the one which is different than the other types of splines. But, it is the one which is the closest to the actual function. The maximum difference it achieves is nearly 1.0×10^{-7}, as we can see from Figure 4.22 above.

4.6 1/(1+x^2) function interpolation with 17 nodes

The next four sub sections explain the four types of splines for the $\frac{1}{(1+x^2)}$ function, when 17 node values of the function are given as input to the software. From the graphs, we can deduce that the natural spline, the complete spline and the not-a-knot spline interpolate similarly, when compared to the clamped cubic spline. Another thing that can said from the earlier discussion about "Runge's phenomenon" is that spline interpolation is usually superior to high degree polynomial interpolation on uniformly spaced x, as a spline, being of fixed degree, does not suffer from this phenomenon [8].

4.6.1 Natural cubic spline for 1/(1+x^2) function

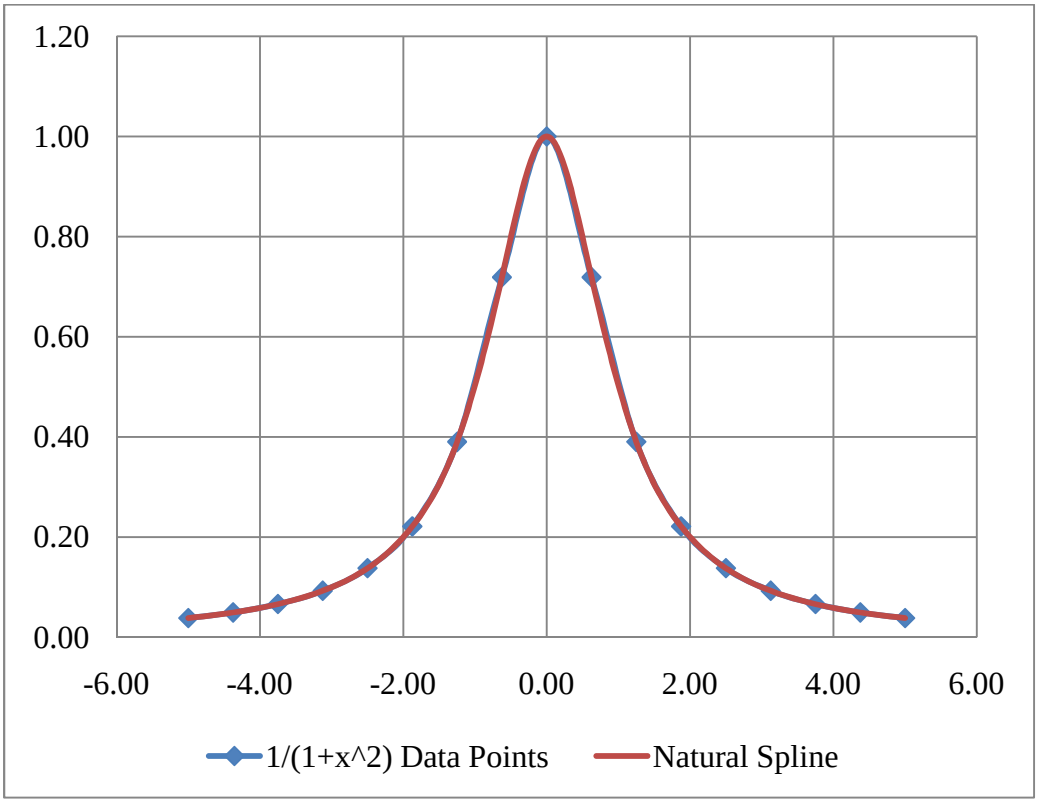

Figure 4.25: Natural cubic spline for 1/(1+x^2) with 17 nodes between [-5, 5]

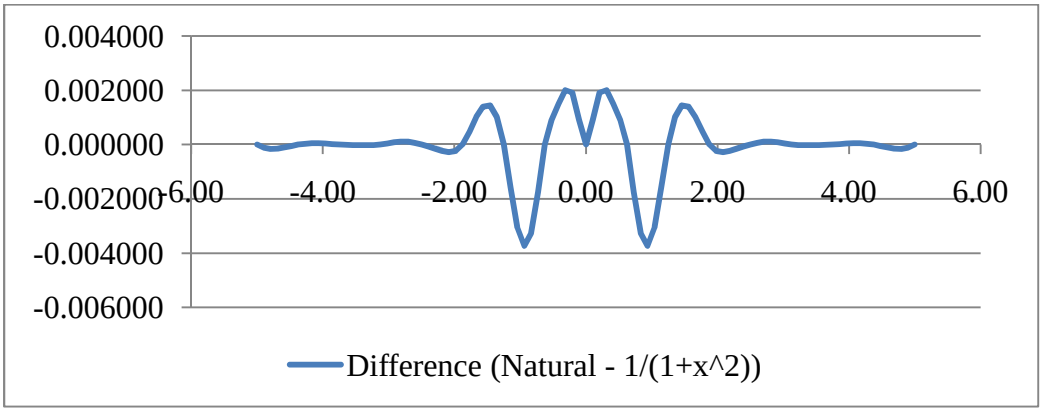

Figure 4.26: Difference between natural spline and 1/(1+x^2) for Figure 4.25

4.6.2 Complete cubic spline for 1/(1+x^2) function

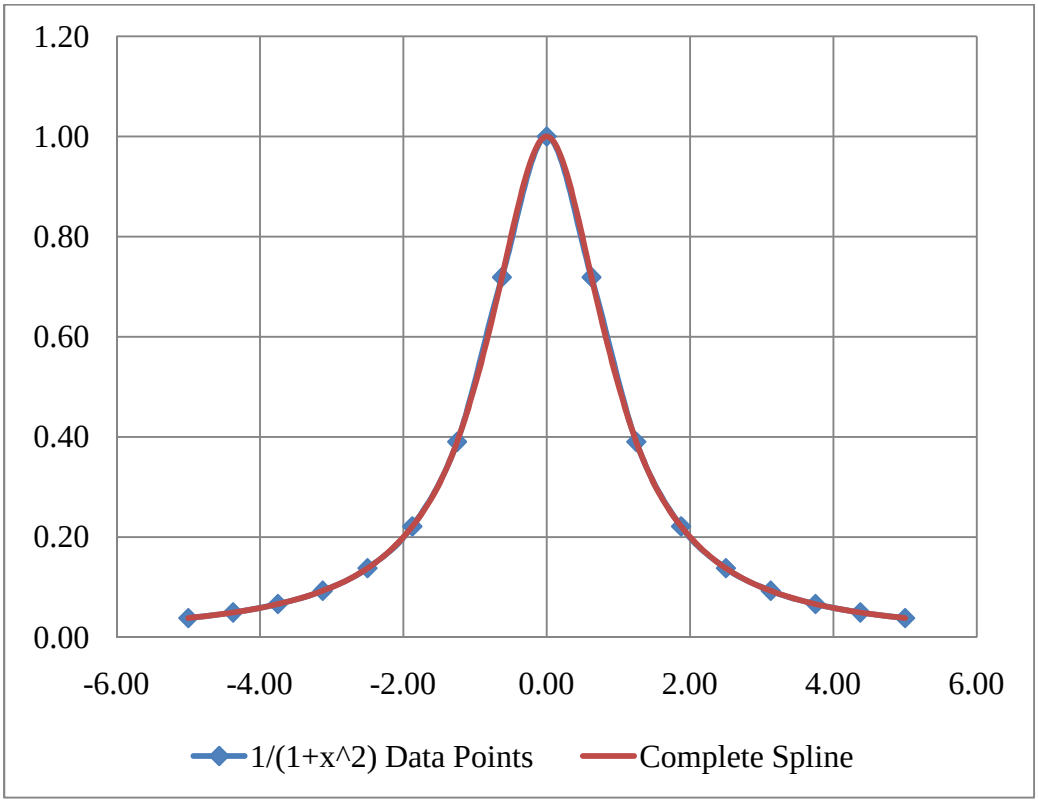

Figure 4.27: Complete cubic spline for 1/(1+x^2) with 17 nodes between [-5, 5]

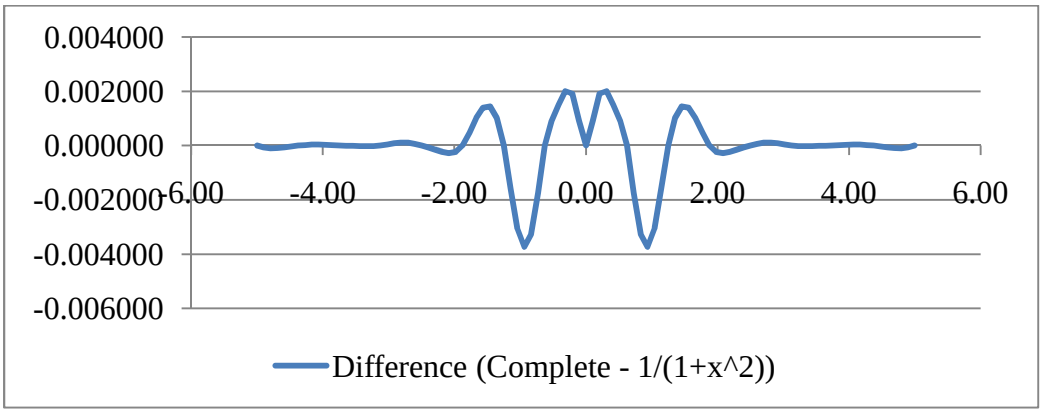

Figure 4.28: Difference between complete spline and 1/(1+x^2) for Figure 4.27

4.6.3 Clamped cubic spline for 1/(1+x^2) function

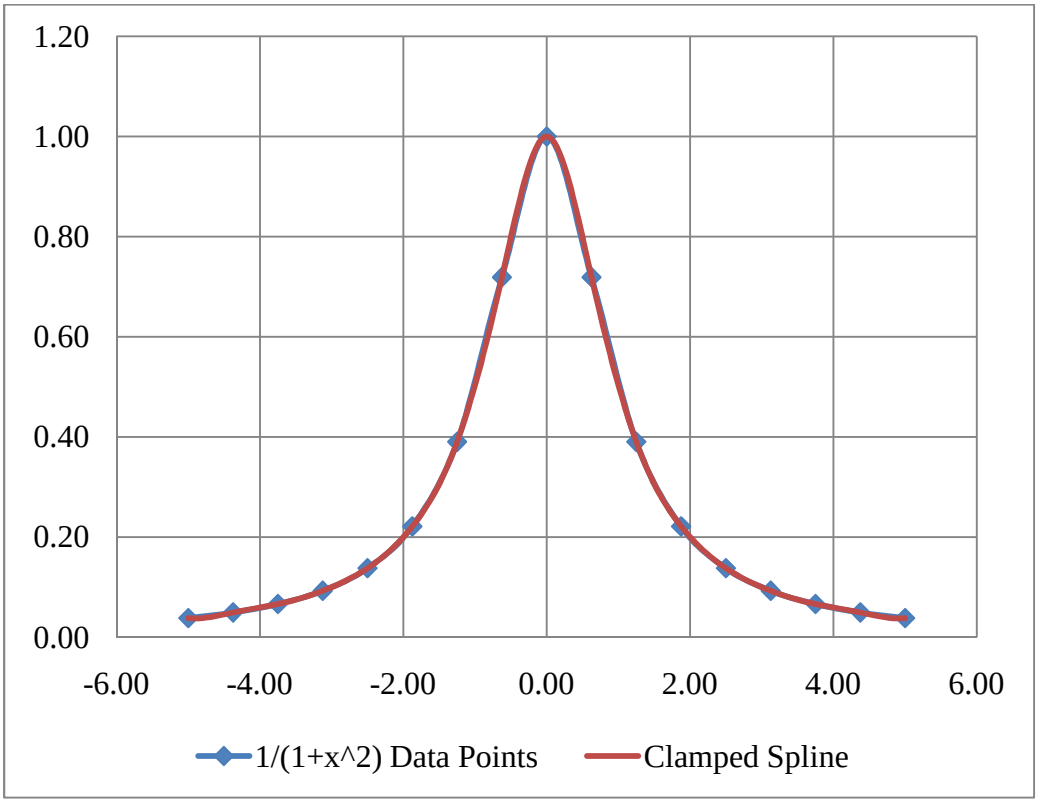

Figure 4.29: Clamped cubic spline for 1/(1+x^2) with 17 nodes between [-5, 5]

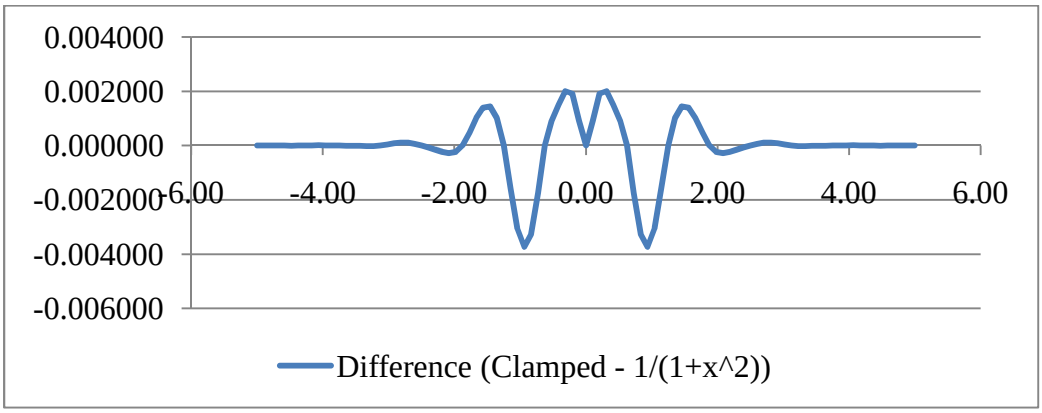

Figure 4.30: Difference between clamped spline and 1/(1+x^2) for Figure 4.29

4.6.4 Not-a-knot cubic spline for 1/(1+x^2) function

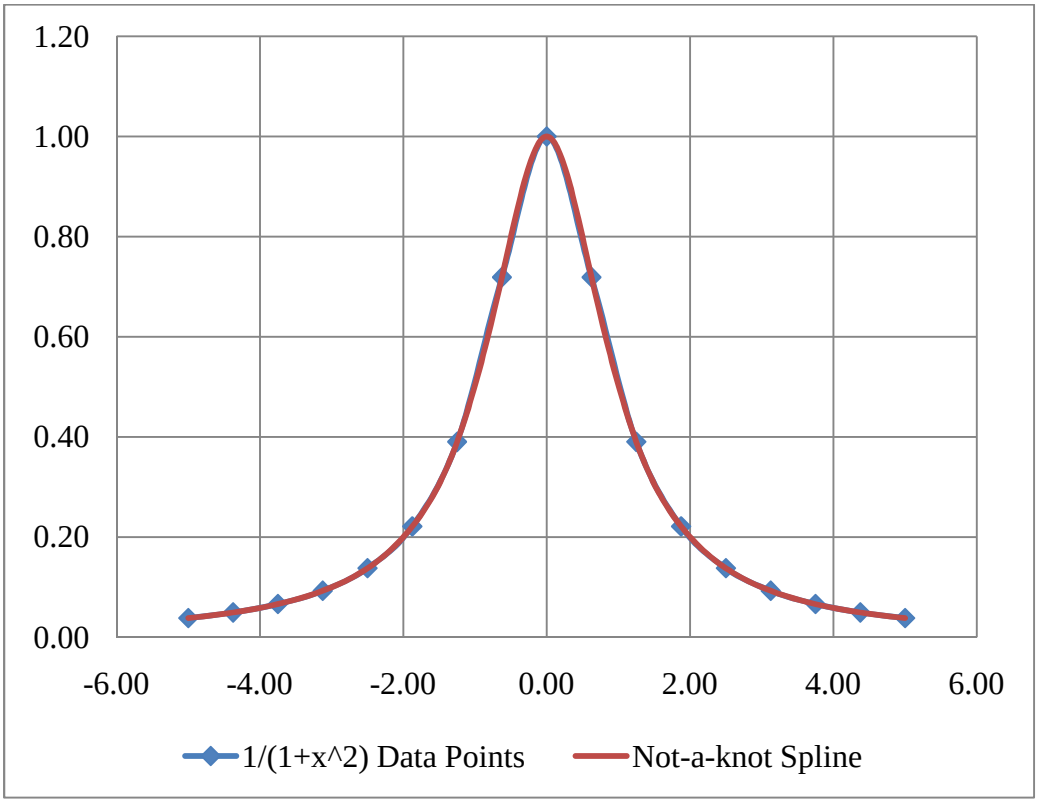

Figure 4.31: Not-a-knot cubic spline for 1/(1+x^2) with 17 nodes between [-5, 5]

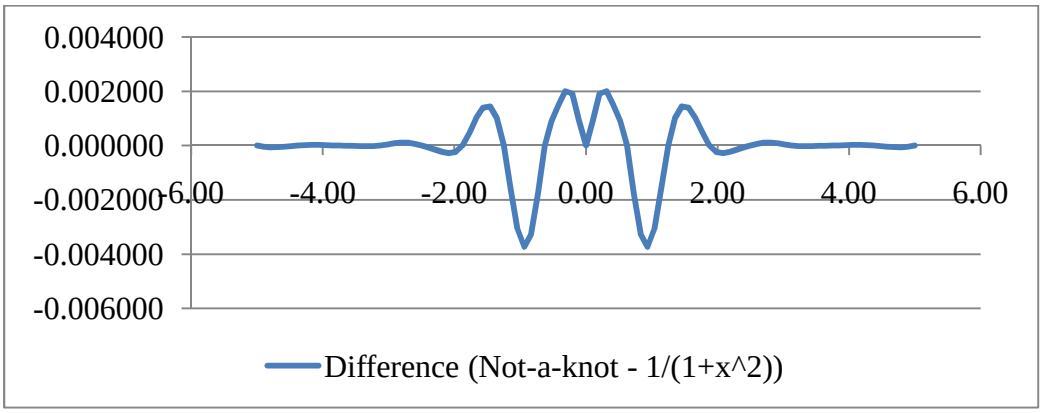

Figure 4.32: Difference between not-a-knot spline and 1/(1+x^2) for Figure 4.31

Out of the 4 splines, the clamped cubic spline is the one which is different than the other types of splines, as mentioned before. But, strangely enough, all the four splines differ from the original function alike. The maximum difference all these 4 splines achieve is relatively higher when compared to the previous trigonometric functions and the exponential function. It is nearly 4.0×10^{-3}, as we can see from the Figures 4.26, 4.28, 4.30, and 4.32 above. This can be attributed to the complexity of the function $\frac{1}{(1+x^2)}$.

CHAPTER V

CONCLUSION

5.1 Summary

The unified software package puts away the trouble of interpolating the cubic splines independently for a given set of data. With its help, we can interpolate all the various types of splines quickly and later compare them with ease.

After interpolating the various splines for various functions, some interesting results are obtained. In all the cases, the clamped cubic spline turned out to be the one interpolating the data closer to the function. This can be true as the user provides the first derivatives of the function at the end points. However, possessing this data is not so typical of the many cases. One other conclusion that is derived from the graphs is that, in many cases the complete cubic spline is similar to the not-a-knot cubic spline.

5.2 Future Work

The current software package consists of 4 types of cubic splines. Other cubic splines can be incorporated into this software without much difficulty. With the addition of more splines, this software package becomes more unified and its reach will be further extended. One immediate suggestion can be the shape-preserving cubic spline, mentioned in the book *Fundamentals of Numerical Computing* [4]. There doesn't seem to be any published algorithm worked out for this shape-preserving cubic spline. If the specifics of that cubic spline can be figured out, it acts a good addition to the package.

REFERENCES

[1] Schumaker, Larry L., *Spline Functions: Basic Theory.* New York : Cambridge University Press, 2007.

[2] Shampine, Lawrence F. and Allen, Richard C. Jr., *Numerical Computing: an introduction.* Philadelphia : W. B. Saunders Company, 1973.

[3] Forsythe, George E., Malcolm, Michael A. and Moler, Cleve B., *Computer Methods for Mathematical Computations.* Englewood Cliffs, NJ : Prentice-Hall, Inc., 1977.

[4] Shampine, Lawrence F., Allen, Richard C. Jr. and Pruess, Steven., *Fundamentals of Numerical Computing.* New York, NY : John Wiley & Sons, Inc., 1997.

[5] Blossey, Peter., Applied Linear Algebra and Numerical Analysis. *University of Washington.* [Online] March 4, 2002. [Cited: June 17, 2010.] http://www.amath.washington.edu/courses/352-winter-2002/spline_note.pdf.

[6] Behforooz, G., "The Not-a-knot Piecewise Interpolatory Cubic Polynomial." Applied Mathematics and Computation, New York, NY : Elsevier Science Publishing Co., Inc., 1992, Vol. 52.

[7] Bradie, Brian., An Introduction to Numerical Analysis with Applications to the Physical, Natural and Social Sciences. *Christopher Newport University.* [Online] January 13, 2006. [Cited: July 23, 2010.] http://www.pcs.cnu.edu/~bbradie/cinterpolation.html.

[8] Wikipedia contributors., Runge's phenomenon. *Wikipedia, The Free Encyclopedia.* [Online] July 12, 2010. [Cited: December 2, 2010.] http://en.wikipedia.org/w/index.php?title=Runge%27s_phenomenon&oldid=400459076.

APPPENDICES

APPENDIX A: THE SOURCE CODE FOR THE SOFTWARE PACKAGE WRITTEN IN JAVA LANGUAGE

A.1 A simple main driver program to interpolate sin *x* using natural cubic spline for an input of 17 nodes

```
public static void main(String[] args)
   {
      final int MAX_NODES = 20;
      final int SPL_PTS = 6;

      final double LOWEST_VALUE = 0.00000001;


      // The starting and ending points of X for the Sine function

      // Sin(X) Range
      final double SIN_BEG = 0.0;
      final double SIN_END = Math.PI;

      double[]
         xn = new double[MAX_NODES],   // X values at the nodes
         yn = new double[MAX_NODES],   // F(X) values at the nodes
         b = new double[MAX_NODES],    // Linear term Coefficients
         c = new double[MAX_NODES],    // Quadratic term Coefficients
         d = new double[MAX_NODES],    // Cubic term Coefficients
         sprpr = new double[MAX_NODES], // Second derivatives
         splineValues = new double[3]; // Spline values, first derivatives
                                       // and the second derivatives of the
                                       // spline at one point

      int
         n,   // Number of nodes
         k,   // 2 power k + 1 nodes
         i,
         j,
         iFunction, // The choice of function
         iMethod;   // The choice of Cubic Spline
```

```
double
   x,
   t,
   dVar1,
   dVar2,
   width;       // Width between nodes


System.out.println("Cubic Spline Interpolation\n");


// Initializations
width = 0.0;
dVar1 = 0.0;
dVar2 = 0.0;


// Assigning the number of nodes as 17
k = 4;
n = (int)Math.pow(2, k) + 1;
System.out.println("Number of nodes = " + n);


// Assigning the interpolating function as Sine
iFunction = 1;
System.out.println("Function being interpolated = Sine\n");

width = (SIN_END - SIN_BEG) / (n - 1);
for(i = 0; i < n; i++)
{
   xn[i] = SIN_BEG + (width * i);
   yn[i] = Math.sin(xn[i]);
   if(yn[i] < LOWEST_VALUE)   yn[i] = 0.0;
}


// Displaying the actual function nodes
for(i = 0; i < n; i++)
   System.out.println("X = " + xn[i] + " Y = " + yn[i]);
System.out.println();


// Selecting the type of cubic spline as Natural cubic spline
iMethod = 1;



// Cubic Spline Coefficients' Calculation
cubicSplineCoeffs(n, xn, yn, iMethod, dVar1, dVar2, b, c, d, sprpr);



// Cubic Spline Values' Calculation
for (i = 0; i < n - 1; i++)
{
   for(j = 0; j < SPL_PTS; j++)
   {
      t = xn[i] + j / (double)SPL_PTS * (xn[i+1] - xn[i]);
      cubicSplineValue(n, xn, yn, iMethod, b, c, d, sprpr, t, i,
         splineValues);
      System.out.println("T = " + t + " Sp = " + splineValues[0]);
   }
}
```

```
      t = xn[i];
      cubicSplineValue(n, xn, yn, iMethod, b, c, d, sprpr, t, i,
         splineValues);
      System.out.println("T = " + t + " Sp = " + splineValues[0]);

   }
   // End public static void main
```

A.2 Interactive main driver program along with the routines comprising the total software package for 17 input nodes

```
// SplineInterp.java   November 2010

// Praveen Motapotu, MS in Computer Science
// Oklahoma State University

import java.util.*;   // Contains Random, StringTokenizer, Scanner, Locale
import java.text.*;   // Contains DecimalFormat, NumberFormat
import java.lang.*;   // Contains Math functions, constants

public class SplineInterp
{
   public static void main(String[] args)
   {
      final int MAX_NODES = 101;
      final int K_MIN = 2;
      final int K_MAX = 6;
      final int SPL_PTS = 6;

      final double LOWEST_VALUE = 0.00000001;


      // The starting and ending points of X for the 4 functions

      // Sin(X) Range
      final double SIN_BEG = 0.0;
      final double SIN_END = Math.PI;
      // Cos(X) Range
      final double COS_BEG = 0.0;
      final double COS_END = Math.PI;
      // e^X Range
      final double EPX_BEG = 0.0;
      final double EPX_END = 1.0;
      // 1/(1+X^2) Range
      final double XSQ_BEG = -5.0;
      final double XSQ_END = 5.0;

      double[]
         xn = new double[MAX_NODES],   // X values at the nodes
         yn = new double[MAX_NODES],   // F(X) values at the nodes
         b = new double[MAX_NODES],    // Linear term Coefficients
         c = new double[MAX_NODES],    // Quadratic term Coefficients
         d = new double[MAX_NODES],    // Cubic term Coefficients
         sprpr = new double[MAX_NODES],// Second Derivatives
         splineValues = new double[3]; // Spline values, first derivatives
                                       // and the second derivatives of the
                                       // spline at one point
```

```
int
   n,   // Number of nodes
   k,   // 2 power k + 1 nodes
   i,
   j,
   iFunction, // The choice of function
   iMethod;   // The choice of Cubic Spline

double
   x,
   t,
   dVar1,
   dVar2,
   width;       // Width between nodes

Scanner sc = new Scanner(System.in);

System.out.println("Cubic Spline Interpolation\n");



// Requesting number of nodes

System.out.print("Enter k, where the number of nodes is 2^k+1: ");
k = sc.nextInt();
if(k < K_MIN || k > K_MAX)
{
   System.out.println("Accepted k values are between 2 & 6, included.");
   System.exit(0);
}

n = (int)Math.pow(2, k) + 1;
System.out.println("Number of nodes = " + n);

// Initializations
width = 0.0;
dVar1 = 0.0;
dVar2 = 0.0;



// Requesting the interpolating function

System.out.println("Enter the function to be interpolated: ");
System.out.println("1. Sin 2. Cos 3. e^x 4. 1/(1+x^2)");
iFunction = sc.nextInt();

if(iFunction == 1)   // Sine function
{
   width = (SIN_END - SIN_BEG) / (n - 1);
   for(i = 0; i < n; i++)
   {
      xn[i] = SIN_BEG + (width * i);
      yn[i] = Math.sin(xn[i]);
      if(yn[i] < LOWEST_VALUE && yn[i] > -LOWEST_VALUE)
         yn[i] = 0.0;
   }
}
else if(iFunction == 2)   // Cosine function
{
   width = (COS_END - COS_BEG) / (n - 1);
   for(i = 0; i < n; i++)
```

```
         {
            xn[i] = COS_BEG + (width * i);
            yn[i] = Math.cos(xn[i]);
            if(yn[i] < LOWEST_VALUE && yn[i] > -LOWEST_VALUE)
               yn[i] = 0.0;
         }
      }
      else if(iFunction == 3)   // e^x function
      {
         width = (EPX_END - EPX_BEG) / (n - 1);
         for(i = 0; i < n; i++)
         {
            xn[i] = EPX_BEG + (width * i);
            yn[i] = Math.pow(Math.E, xn[i]);
            if(yn[i] < LOWEST_VALUE && yn[i] > -LOWEST_VALUE)
               yn[i] = 0.0;
         }
      }
      else if(iFunction == 4)   // 1/1+x^2 function
      {
         width = (XSQ_END - XSQ_BEG) / (n - 1);
         for(i = 0; i < n; i++)
         {
            xn[i] = XSQ_BEG + (width * i);
            yn[i] = 1 / (1 + Math.pow(xn[i], 2));
            if(yn[i] < LOWEST_VALUE && yn[i] > -LOWEST_VALUE)
               yn[i] = 0.0;
         }
      }
      else
      {
         System.out.println("Only 4 choices of functions available!");
         System.exit(0);
      }

      // Displaying the actual function nodes
      for(i = 0; i < n; i++)
         System.out.println("X = " + xn[i] + " Y = " + yn[i]);



      // Requesting the type of cubic spline

      System.out.println("Enter the type of cubic spline: ");
      System.out.println("1. Natural 2. Complete 3. Clamped 4. Not-a-knot");
      iMethod = sc.nextInt();

      // Requesting the first derivatives for Clamped Spline
      if(iMethod == 3)
      {
         System.out.println("Please enter the two end conditions for clamped
spline:");
         dVar1 = sc.nextDouble();
         dVar2 = sc.nextDouble();
      }

      if(iMethod < 1 || iMethod > 4)
      {
         System.out.println("Only 4 types of cubic splines available!");
         System.exit(0);
      }
```

```
      // Cubic Spline Coefficients' Calculation
      cubicSplineCoeffs(n, xn, yn, iMethod, dVar1, dVar2, b, c, d, sprpr);



      // Cubic Spline Values' Calculation
      for (i = 0; i < n - 1; i++)
      {
         for(j = 0; j < SPL_PTS; j++)
         {
            t = xn[i] + j / (double)SPL_PTS * (xn[i+1] - xn[i]);
            cubicSplineValue(n, xn, yn, iMethod, b, c, d, sprpr, t, i,
               splineValues);
            System.out.println("T = " + t + " Sp = " + splineValues[0]);
         }
      }
      t = xn[i];
      cubicSplineValue(n, xn, yn, iMethod, b, c, d, sprpr, t, i,
         splineValues);
      System.out.println("T = " + t + " Sp = " + splineValues[0]);

   }
   // End public static void main



// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *



   public static void cubicSplineCoeffs(int n, double[] xn, double[] yn,
      int iSpline, double dVar1, double dVar2, double[] b, double[] c,
      double[] d, double[] sprpr)
   {
   /*
        PURPOSE:
             Calculate the coefficients for any type of the cubic spline
             for a given set of data


        CALLING SEQUENCE:
             cubicSplineCoeffs(n, xn, yn, iSpline, dVar1, dVar2);


        INPUTS:
             n   number of interpolating points or nodes
             xn  array containing interpolating points
             yn  array containing function values to be interpolated;
                 yn[i] is the function value corresponding to xn[i]
             iSpline
                 the number representing the type of cubic spline
             dVar1
                 the first derivative at x=a used by clamped spline
             dVar2
                 the first derivative at x=b used by clamped spline
             b   array of size at least n;
             c   array of size at least n;
             d   array of size at least n;
             sprpr
                 array of size at least n;
```

```
        OUTPUTS:
             b   coefficients of linear terms in cubic spline
             c   coefficients of quadratic terms in cubic spline
             d   coefficients of cubic terms in cubic spline
             sprpr
                 second derivatives of the cubic spline
   */

      if(iSpline == 1)
         natSplCoeffs(n, xn, yn, sprpr);

      else if(iSpline == 2)
         cmplSplCoeffs(n, xn, yn, b, c, d);

      else if(iSpline == 3)
         clmpSplCoeffs(n, xn, yn, b, c, d, dVar1, dVar2);

      else if(iSpline == 4)
         nakSplCoeffs(n, xn, yn, b, c, d);

      else
      {
         System.out.println("Only 4 types of cubic splines available!");
         System.exit(0);
      }
   }
   // End public static void cubicSplineCoeffs



// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *



   public static void cubicSplineValue(int n, double[] xn, double[] yn,
      int iSpline, double[] b, double[] c, double[] d, double[] sprpr,
      double t, int index, double[] splVals)
   {
   /*
        PURPOSE:
             Calculate the actual value for any type of the cubic spline
             by providing it's coefficients at a single point within the
             nodes
             Also calculate the values of the first and second
             derivatives of the spline at that point


        CALLING SEQUENCE:
             cubicSplineValue(n, xn, yn, iSpline, b, c, d, sprpr, t, index,
             fds, sds);


        INPUTS:
             n   number of interpolating points or nodes
             xn  array containing interpolating points
             yn  array containing function values to be interpolated;
                 yn[i] is the function value corresponding to xn[i]
             iSpline
                 the number representing the type of cubic spline
             b   coefficients of linear terms in cubic spline
             c   coefficients of quadratic terms in cubic spline
             d   coefficients of cubic terms in cubic spline
```

```
            sprpr
                second derivatives of the cubic spline
            t   point where the cubic spline is to be evaluated
            index
                index of the closest node less than or equal to 't'
            splVals
                array of size 3;


        OUTPUTS:
            splVals
                array containing the value of the cubic spline at [0]
                the value of the first derivative of the spline at [1]
                the value of the second derivative of the spline at [2]
   */

      if(iSpline == 1)
         natSplValue(n, xn, yn, sprpr, t, index+1, splVals);

      else if(iSpline == 2 || iSpline == 3 || iSpline == 4)
         normSplValue(n, xn, yn, b, c, d, t, splVals);

      else
      {
         System.out.println("Only 4 types of cubic splines available!");
         System.exit(0);
      }

   }
   // End public static void cubicSplineValue



// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *



   public static void natSplCoeffs(int n, double[] xn, double[] yn, double[] s)
   {

//  Natural cubic spline interpolation

//  "Numerical Computing: An Introduction"
//      L. F. Shampine and R. C. Allen, Jr.,
//  W. B. Saunders Co., 1973

//  natSplCoeffs calculates the array s[] of second derivatives needed to
define
//     the spline.

      double[]
         rho = new double[n],
         tau = new double[n];

      double him1, hi, temp, d;

//  Compute the elements of the arrays rho[] and tau[].

      rho[1] = 0.0;
      tau[1] = 0.0;

      for(int i = 1; i < n - 1; i++)
```

```
      {
         him1 = xn[i] - xn[i-1];
         hi = xn[i+1] - xn[i];


         temp = (him1 / hi) * (rho[i] + 2.0) + 2.0;
         rho[i+1] = -1.0 / temp;
         d = 6.0 * ((yn[i+1] - yn[i]) / hi - (yn[i] - yn[i-1]) / him1) / hi;
         tau[i+1] = (d - him1 * tau[i] / hi) / temp;
      }

//  Compute the array of second derivatives s[] for the natural spline.

      s[0] = 0.0;
      s[n-1] = 0.0;
      for(int i = (n - 2); i > 0; i--)
         s[i] = rho[i+1] * s[i+1] + tau[i+1];

   }
   //  End public static void natSplCoeffs



// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *



   public static void cmplSplCoeffs(int n, double[] x, double[] f, double[] b,
      double[] c, double[] d)
   {
   /*
   FUNCTION:     Functions for setting up and evaluating a cubic
                 interpolatory spline.
   AUTHORS:      Lawrence Shampine, Richard Allen, Steven Pruess  for
                 the text  Fundamentals of Numerical Computing
   DATE:         August 25, 1995
   LAST CHANGE:  April 3, 1998

   PURPOSE:
        Calculate coefficients defining a complete cubic interpolatory spline.

   INPUTS:
        n   number of data points
        x   vector of values of the independent variable ordered
            so that  x[i] < x[i+1]  for all i
        f   vector of values of the dependent variable

   OUTPUTS:
        b   vector of S'(x[i]) values
        c   vector of S"(x[i])/2 values
        d   vector of S'''(x[i]+)/6 values
   */

      int
         i,
         k;

      double
         fp1,
         fpn,
         h = 0.0,
         p;
```

```
//  Calculate coefficients for the tri-diagonal system:
//     store sub-diagonal in b, diagonal in d, difference quotient in c.

      b[0] = x[1]-x[0];

      c[0] = (f[1] - f[0]) / b[0];

      if(n == 2)
      {
         b[0] = c[0];
         c[0] = 0.0;
         d[0] = 0.0;
         b[1] = b[0];
         c[1] = 0.0;

         return;
      }

      d[0] = 2.0 * b[0];

      for(i = 1; i < n-1; i++)
      {
         b[i] = x[i+1]-x[i];

         c[i] = (f[i+1] - f[i]) / b[i];
         d[i] = 2.0 * (b[i] + b[i-1]);
      }

      d[n-1] = 2.0 * b[n-2];


//  Calculate estimates for the end slopes.
//  Use polynomials interpolating data nearest the end.

      fp1 = c[0] - b[0] * (c[1] - c[0]) / (b[0] + b[1]);

      if (n > 3)
         fp1 = fp1 + b[0] * ((b[0] + b[1]) * (c[2] - c[1]) /
                     (b[1] + b[2]) - c[1] + c[0]) / (x[3] - x[0]);

      fpn = c[n-2] + b[n-2] * (c[n-2] - c[n-3]) / (b[n-3] + b[n-2]);

      if (n > 3)
         fpn = fpn + b[n-2] * (c[n-2] - c[n-3] - (b[n-3] + b[n-2]) *
                  (c[n-3] - c[n-4]) / (b[n-3] + b[n-4])) /
                  (x[n-1] - x[n-4]);


//  Calculate the right-hand-side and store it in c.

      c[n-1] = 3.0 * (fpn - c[n-2]);

      for (i = n-2; i > 0; i--)
         c[i] = 3.0 * (c[i] - c[i-1]);

      c[0] = 3.0 * (c[0] - fp1);


//  Solve the tridiagonal system.

      for (k = 1; k < n; k++)
```

```
      {
         p = b[k-1] / d[k-1];
         d[k] = d[k] - p * b[k-1];
         c[k] = c[k] - p * c[k-1];
      }

      c[n-1] = c[n-1] / d[n-1];

      for (k = n-2; k >= 0; k--)
         c[k] = (c[k] - b[k] * c[k+1]) / d[k];


//  Calculate the coefficients defining the spline.

      for (i = 0; i < n-1; i++)
      {
         h = x[i+1] - x[i];
         d[i] = (c[i+1] - c[i]) / (3.0 * h);
         b[i] = (f[i+1] - f[i]) / h - h * (c[i] + h*d[i]);
      }

      b[n-1] = b[n-2] + h * (2.0 * c[n-2] + h * 3.0 * d[n-2]);

      return;

   }
   //  End public static int cmplSplCoeffs



// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *



   public static void clmpSplCoeffs(int n, double[] x, double[] f, double[] b,
      double[] c, double[] d, double fpa, double fpb)
   {
/*
     PURPOSE:
          determine the coefficients for the clamped
          cubic spline for a given set of data


     CALLING SEQUENCE:
          clmpSplCoeffs(n, x, f, b, c, d, fpa, fpb);


     INPUTS:
          n    number of interpolating points
          x    array containing interpolating points
          f    array containing function values to
               be interpolated;  f[i] is the function
               value corresponding to x[i]
          b    array of size at least n;
          c    array of size at least n;
          d    array of size at least n;
          fpa  derivative of function at x=a
          fpb  derivative of function at x=b


     OUTPUTS:
          b    coefficients of linear terms in cubic spline
```

```
          c    coefficients of quadratic terms in cubic spline
          d    coefficients of cubic terms in cubic spline
*/


      double[]
         h  = new double[n],
         dl = new double[n],
         dd = new double[n],
         du = new double[n];

      int i;

      for(i = 0; i < n-1; i++)
      {
         h[i] = x[i+1] - x[i];
         dl[i] = du[i] = h[i];
      }

      dd[0] = 2.0 * h[0];
      dd[n-1] = 2.0 * h[n-2];
      c[0] = (3.0 / h[0]) * (f[1] - f[0]) - 3.0 * fpa;
      c[n-1] = 3.0 * fpb - (3.0 / h[n-2]) * (f[n-1] - f[n-2]);
      for(i = 0; i < n-2; i++)
      {
         dd[i+1] = 2.0 * (h[i] + h[i+1]);
         c[i+1] = (3.0 / h[i+1]) * (f[i+2] - f[i+1]) - (3.0 / h[i]) * (f[i+1] -
f[i]);
      }

      tridiagonal(n, dl, dd, du, c);

      for(i = 0; i < n-1; i++)
      {
         d[i] = (c[i+1] - c[i]) / (3.0 * h[i]);
         b[i] = (f[i+1] - f[i]) / h[i] - h[i] * (c[i+1] + 2.0 * c[i]) / 3.0;
      }
   }
   //  End public static void clmpSplCoeffs



// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *



   public static void nakSplCoeffs(int n, double[] x, double[] f, double[] b,
      double[] c, double[] d )
   {
   /*
        PURPOSE:
             determine the coefficients for the 'not-a-knot'
             cubic spline for a given set of data


        CALLING SEQUENCE:
             nakSplCoeffs(n, x, f, b, c, d);


        INPUTS:
             n   number of interpolating points
             x   array containing interpolating points
```

```
            f   array containing function values to
                be interpolated;  f[i] is the function
                value corresponding to x[i]
            b   array of size at least n;
            c   array of size at least n;
            d   array of size at least n;


        OUTPUTS:
            b   coefficients of linear terms in cubic spline
            c   coefficients of quadratic terms in cubic spline
            d   coefficients of cubic terms in cubic spline
    */

        double[]
           h  = new double[n],
           dl = new double[n],
           dd = new double[n],
           du = new double[n];
        int i;

        for(i = 0; i < n-1; i++)
           h[i] = x[i+1] - x[i];
        for(i = 0; i < n-3; i++)
           dl[i] = du[i] = h[i+1];

        for(i = 0; i < n-2; i++)
        {
           dd[i] = 2.0 * (h[i] + h[i+1]);
           c[i] = (3.0 / h[i+1]) * (f[i+2] - f[i+1]) -
                     (3.0 / h[i]) * (f[i+1] - f[i]);
        }
        dd[0] += (h[0] + h[0]*h[0] / h[1]);
        dd[n-3] += (h[n-2] + h[n-2]*h[n-2] / h[n-3]);
        du[0] -= (h[0]*h[0] / h[1]);
        dl[n-4] -= (h[n-2]*h[n-2] / h[n-3]);


        tridiagonal(n-2, dl, dd, du, c);


        for(i = n-3; i >= 0; i--)
           c[i+1] = c[i];
        c[0] = (1.0 + h[0] / h[1]) * c[1] - h[0] / h[1] * c[2];
        c[n-1] = (1.0 + h[n-2] / h[n-3]) * c[n-2] - h[n-2] / h[n-3] * c[n-3];
        for(i = 0; i < n-1; i++)
        {
           d[i] = (c[i+1] - c[i]) / (3.0 * h[i]);
           b[i] = (f[i+1] - f[i]) / h[i] - h[i] * (c[i+1] + 2.0*c[i]) / 3.0;
        }
    }
    //  End public static void nakSplCoeffs



// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *



    public static void natSplValue(int n, double[] xn, double[] yn, double[] s,
       double x, int i, double[] splVals)
    {
```

```
/*
     PURPOSE:
          Evaluate a cubic spline at a single value of
          the independent variable given the coefficients of
          the cubic spline interpolant

     TYPE:
          It works for Natural cubic spline only

     CALLING SEQUENCE:
          y = natSplValue(n, xn, yn, s, x, i);

     INPUTS:
          n   number of interpolating points or nodes
          xn  array containing interpolating points
          yn  array containing the constant terms from
              the cubic spline
          s   array containing the second derivatives
              of the cubic spline
          x   value of independent variable at which
              the interpolating polynomial is to be
              evaluated
          i   index value of the nodes
          splVals
              array of size 3;


     OUTPUTS:
          splVals
              array containing the value of the cubic spline at [0]
              the value of the first derivative of the spline at [1]
              the value of the second derivative of the spline at [2]
*/
   double
      a,
      b,
      hL;

   if(i == n)
   {
      splVals[0] = yn[i-1];
      splVals[1] =
      splVals[2] = 0.0;
   }

   else
   {
      a = xn[i] - x;
      b = x - xn[i-1];
      hL = xn[i] - xn[i-1];

      // Spline value
      splVals[0] = a * s[i-1] * (a * a / hL - hL)/6.0 +
                     b * s[i] * (b * b / hL - hL) / 6.0 +
                     (a * yn[i-1] + b * yn[i]) / hL;

      // First derivative
      splVals[1] = (b * b * s[i] - a * a * s[i-1])/(2.0 * hL) +
                     hL * (s[i-1] - s[i])/6.0 + (yn[i] - yn[i-1])/hL;

      // Second derivative
      splVals[2] = (a * s[i-1] + b * s[i])/hL;
   }
```

```
   }
   //  End public static void natSplValue


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


   public static void normSplValue(int n, double[] x, double[] f, double[] b,
      double[] c, double[] d, double t, double[] splVals)
    {
   /*
        PURPOSE:
             Evaluate a cubic spline at a single value of
             the independent variable given the coefficients of
             the cubic spline interpolant

             Also evaluates the first and second derivatives of
             the spline at this single value

        TYPE:
             It works for Complete, Clamped & Not-a-knot cubic
             splines

        CALLING SEQUENCE:
             y = normSplValue(n, x, f, b, c, d, t, fds, sds);

        INPUTS:
             n   number of interpolating points
             x   array containing interpolating points
             f   array containing the constant terms from
                 the cubic spline
             b   array containing the coefficients of the
                 linear terms from the cubic spline
             c   array containing the coefficients of the
                 quadratic terms from the cubic spline
             d   array containing the coefficients of the
                 cubic terms from the cubic spline
             t   value of independent variable at which
                 the interpolating polynomial is to be
                 evaluated
             splVals
                 array of size 3;


        OUTPUTS:
             splVals
                 array containing the value of the cubic spline at [0]
                 the value of the first derivative of the spline at [1]
                 the value of the second derivative of the spline at [2]
   */

        int
           i,
           found;

        double
           dt;

        i = 1;
```

```
        found = 0;

        while ( (found == 0) && ( i < n-1 ) )
        {
           if ( t < x[i] )
              found = 1;
           else
              i++;
        }

        dt = t - x[i-1];

        // Calculating the spline value
        splVals[0] = f[i-1] + dt * (b[i-1] + dt * (c[i-1] + dt * d[i-1]));

        // Calculating the first derivative
        splVals[1] = b[i-1] + dt * (2.0 * c[i-1] + dt * 3.0 * d[i-1]);

        // Calculating the second derivative
        splVals[2] = 2.0 * c[i-1] + dt * 6.0 * d[i-1];

   }
   //  End public static void normSplValue



// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *



   public static void tridiagonal(int n, double[] c, double[] a, double[] b,
      double[] r)
   {
        int i;

        for(i = 0; i < n-1; i++ )
        {
            b[i] /= a[i];
            a[i+1] -= c[i]*b[i];
        }

        r[0] /= a[0];

        for ( i = 1; i < n; i++ )
            r[i] = ( r[i] - c[i-1] * r[i-1] ) / a[i];

        for ( i = n-2; i >= 0; i-- )
            r[i] -= r[i+1] * b[i];
   }
   // End public static void tridiagonal



// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *



}
// End public class SplineInterp
```

VITA

Praveen Motapotu

Candidate for the Degree of

Master of Science

Thesis: A UNIFIED SOFTWARE PACKAGE FOR CUBIC SPLINE INTERPOLATION

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Master of Science in Computer Science at Oklahoma State University, Stillwater, Oklahoma in December, 2010.

Completed the requirements for the Bachelor of Technology in Electronics & Communication Engineering at Jawaharlal Nehru Technological University, Hyderabad, Andhra Pradesh, India in 2003.

Name: Praveen Motapotu Date of Degree: December, 2010

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: A UNIFIED SOFTWARE PACKAGE FOR CUBIC SPLINE INTERPOLATION

Pages in Study: 53 Candidate for the Degree of Master of Science

Major Field: Computer Science

Scope and Method of Study:

Cubic spline interpolation is currently performed using various routines present in different computer programming languages for various types of splines. It is less tedious and more convenient for users to use a unified software package consisting of different types of splines. However, there is no known software that interpolates all the types of splines for a given set of data points. Hence, a unified software package using Java language is developed to interpolate four types of cubic splines: natural cubic spline, complete cubic spline, clamped cubic spline, and not-a-knot cubic spline.

Findings and Conclusions:

The unified software package is tested for four basic functions: $\sin x$, $\cos x$, e^x and $\frac{1}{(1+x^2)}$ within the ranges of $[0, \pi]$, $[0, \pi]$, $[0, 1]$, and $[-5, 5]$, respectively. The number of nodes is taken as 17 $(2^4 + 1)$, which represents a typical number in the interpolation of splines. Two graphs are drawn, for each function and spline pair, one showing the cubic spline curve with respect to the actual function curve and its data points, and another one displaying the difference between the spline values and the function values. Out of all the four types, the clamped cubic spline turned out to be the one interpolating the data closest to the function. This is not a surprise considering the fact that the user provides the end conditions calculated from the actual function. Also, the complete cubic spline is similar to the not-a-knot cubic spline in all the cases tested. To conclude, with the help of the unified software package, cubic spline interpolation can be done with much ease, as shown.

ADVISER'S APPROVAL: Dr. John Chandler