ASSIGNMENT 05 SOLUTIONS (ALL IN ONE)

Cubic Spline Interpolation

Given a function f defined on $[a, b]$ and a set of nodes $a = x_0 < x_1 < \cdots < x_n = b$, a cubic spline interpolant S for f is a function that satisfies the following conditions:

1. $S(x)$ is a cubic polynomial, denoted $S_j(x)$, on the subinterval $[x_j, x_{j+1}]$ for each $j = 0, 1, \cdots, n-1$
2. $S_j(x) = f(x_j)$ and $S_j(x_{j+1}) = f(x_{j+1})$ for each $j = 0, 1, \cdots, n-1$
3. $S_{j+1}(x_{j+1}) = S_j(x_{j+1})$ for each $j = 0, 1, \cdots, n-2$ (Implied by 2)
4. $S'_{j+1}(x_{j+1}) = S'_j(x_{j+1})$ for each $j = 0, 1, \cdots, n-2$
5. $S''_{j+1}(x_{j+1}) = S''_j(x_{j+1})$ for each $j = 0, 1, \cdots, n-2$
6. One of the following sets of boundary conditions is satisfied:
   - $S''(x_0) = S''(x_n) = 0$ (**natural or free boundary**)
   - $S'(x_0) = f'(x_0)$ and $S'(x_n) = f'(x_n)$ (**clamped boundary**).

Source: From the book Numerical Analysis by Richard L. Burden and J. Douglas Faires chapter 3.

Like everytime I'm going to write a c file `csplines.c` which will contain all the necessary functions such as natural and clamped cubic spline, lagrange interpolation and an integration function for simpson's 1/3 rule.

In [ ]:
```
//function to calculate Lagrange interpolated value
double lagrange(int n,double X[],double Y[],double x)
{
    // n=no of points (n-1=order of interpolation)
    double sum=0;
    int i,j;
    for(i=0;i<n;i++)
    {
        // initiating product part
        double prod=1;
        for(j=0;j<n;j++)
        {
            if(j!=i)
            prod=prod*(x-X[j])/(X[i]-X[j]);
        }
        sum=sum+prod*Y[i];
    }
    return sum;
}
//function to calculate natural cubic spline interpolated value
double csplines(int n,double x[],double a[],double X)
{
    n--;    // no of splines
    double h[n],alpha[n],l[n+1],u[n+1],z[n+1],c[n+1],b[n],d[n];
    double sum;
    int i,j;
    // Step 1
    for (i=0;i<=n-1;++i)
        h[i]=x[i+1]-x[i];
```

```c
    // Step 2
    for (i=1;i<=n-1;++i)
        alpha[i]=3*(a[i+1]-a[i])/h[i]-3*(a[i]-a[i-1])/h[i-1];
    // Step 3
    l[0] = 1;
    u[0] = 0;
    z[0] = 0;
    // Step 4
    for (i=1;i<=n-1;++i)
    {
        l[i]=2*(x[i+1]-x[i-1])-h[i-1]*u[i-1];
        u[i]=h[i]/l[i];
        z[i]=(alpha[i]-h[i-1]*z[i-1])/l[i];
    }
    // Step 5
    l[n] = 1;
    z[n] = 0;
    c[n] = 0;
    // Step 6
    for (j=n-1;j>=0;--j) {
        c[j]=z[j]-u[j]*c[j+1];
        b[j]=(a[j+1]-a[j])/h[j]-h[j]*(c[j+1]+2*c[j])/3;
        d[j]=(c[j+1]-c[j])/(3*h[j]);
    }
    // Step 7
    for (i=0;i<n+1;++i) {
        if (X>=x[i] && X<x[i+1])
        {
                sum=a[i]+b[i]*(X-x[i])+c[i]*pow((X-x[i]),2)
                +d[i]*pow((X-x[i]),3);
        }
    }
    return sum;
}
//function to calculate clamped cubic spline interpolated value
double clamped(int n,double x[],double a[],double F0,double F1, double X)
{
    n--;    // no of splines
    double h[n],alpha[n],l[n+1],u[n+1],z[n+1],c[n+1],b[n],d[n];
    double sum;
    int i,j;
    // Step 1
    for (i=0;i<=n-1;++i)
        h[i]=x[i+1]-x[i];
    // Step 2
    alpha[0]=3*(a[1]-a[0])/h[0]-3*F0;
    alpha[n]=3*F1-3*(a[n]-a[n-1])/h[n-1];
    // Step 3
    for (i=1;i<=n-1;++i)
        alpha[i]=(3/h[i])*(a[i+1]-a[i])-(3/h[i-1])*(a[i]-a[i-1]);
    // Step 4
    l[0] = 2*h[0];
    u[0] = 0.5;
    z[0] = alpha[0]/l[0];
    // Step 5
    for (i=1;i<=n-1;++i)
    {
        l[i]=2*(x[i+1]-x[i-1])-h[i-1]*u[i-1];
        u[i]=h[i]/l[i];
        z[i]=(alpha[i]-h[i-1]*z[i-1])/l[i];
    }
    // Step 6
    l[n] = h[n-1]*(2-u[n-1]);
    z[n] = (alpha[n]-h[n-1]*z[n-1])/l[n];
    c[n] = z[n];
    // Step 7
    for (j=n-1;j>=0;--j) {
        c[j]=z[j]-u[j]*c[j+1];
        b[j]=(a[j+1]-a[j])/h[j]-h[j]*(c[j+1]+2*c[j])/3;
        d[j]=(c[j+1]-c[j])/(3*h[j]);
    }
    // Step 8
    for (i=0;i<n+1;++i) {
        if (X>=x[i] && X<x[i+1])
        {
            sum=a[i]+b[i]*(X-x[i])+c[i]*pow((X-x[i]),2)
                +d[i]*pow((X-x[i]),3);
        }
    }
    return sum;
}
// simpson's 1/3 method
double simpson(double f(double x),double a,double b)
{
    int i,n=2;  // starting with two interval
    double integral,answer,x,h,sum,accuracy=0.00001;
    do{
        integral=answer;
        h=fabs(b-a)/n;
        sum=0;
        for(i=1;i<n;++i)
        {
            x=a+i*h;
            if(i%2==0){
                sum=sum+2*f(x);
            }
            else{
                sum=sum+4*f(x);
            }
        }
        answer=(h/3)*(f(a)+f(b)+sum);
        n=n+2;
    }while(fabs(answer-integral)>=accuracy);
  return answer;
}
```

Now since I have written all the required functions in the `"csplines.c"`, we'll use `#include"csplines.c"` as a library.

How to use the functions?

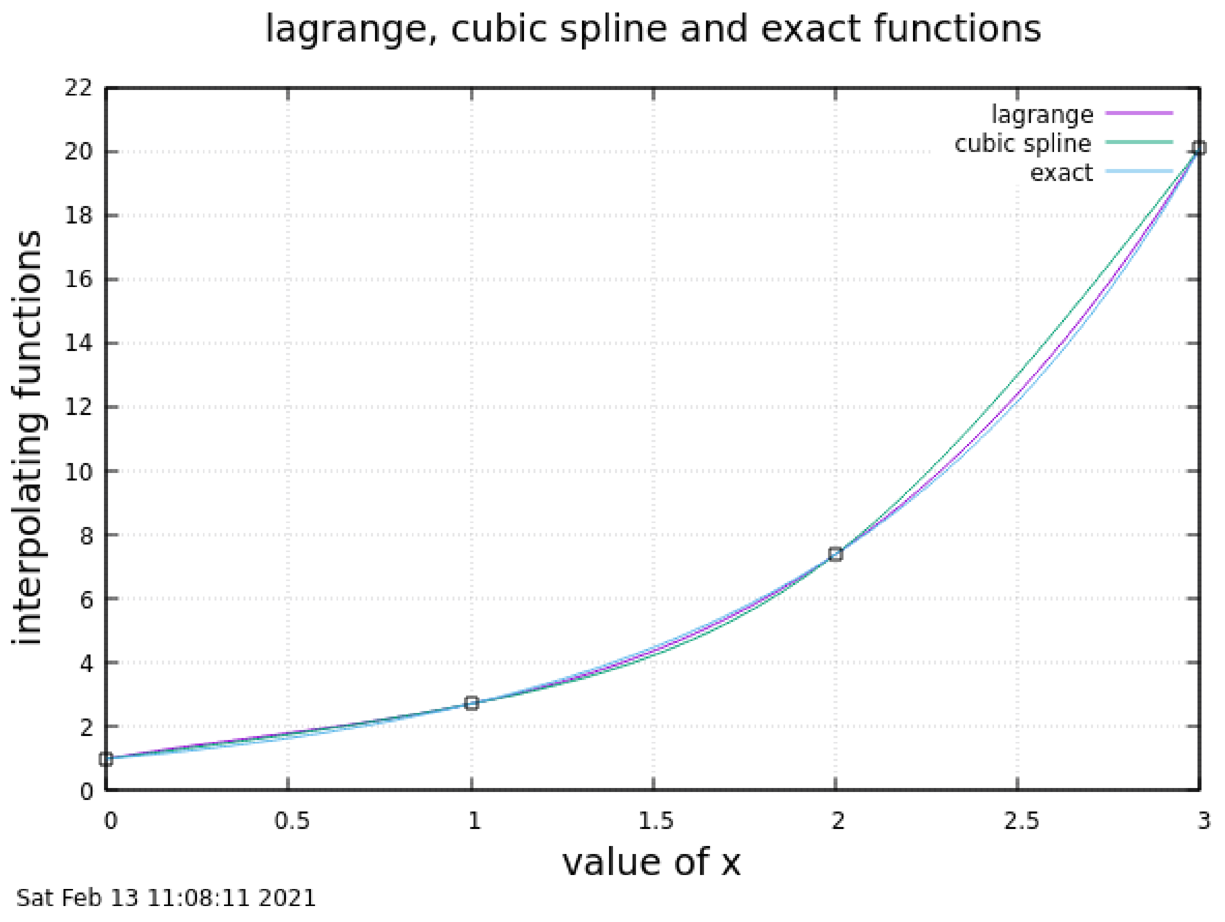
- Natural Cubic Spline: `csplines(n,x,a,X)`
- Clamped Cubic Spline: `clamped(n,x,a,X)`
- Lagrange Method: `lagrange(n,x,a,X)`
- Simpson's 1/3 Rule: `simpson(f,a,b)`

Where **x** and **a** are the input arrays defined as double variable, **n** is the number of points to be used *(n-1= number of splines)* defined as integer variable and **X** is the value where we want interpolated value. For simpson **f(x)** is the function to be integrated while **a** is the lower limit and **b** is the upper limit of integration.

PROBLEM 1 & 2 :

In [ ]:
```
// problem 1 and 2
// make sure "csplines.c" file is in the same directory
#include<stdio.h>
#include<math.h>
#include"csplines.c"

// main program to do our job
int main()
{
        double X;
        double x[]={0,1,2,3};
        double y[]={exp(0),exp(1),exp(2),exp(3)};
        FILE*fp=NULL;
        fp=fopen("csp.txt","w");
        for (X=0;X<=3;X+=0.01)
        {
                fprintf(fp,"%lf\t%lf\t%lf\t%lf\n",X,lagrange(4,x,y,X),
                csplines(4,x,y,X),exp(X));
        }
}
```

OUTPUT:



PROBLEM 3 & 5 :

In [ ]:
```
// problem 3 and 5
// make sure "csplines.c" file is in the same directory
#include<stdio.h>
#include<math.h>
#include"csplines.c"
// natural cubic spline function
double f1(double X) {
        double x[4]={0,1,2,3};
        double y[4]={exp(0),exp(1),exp(2),exp(3)};
        return csplines(4,x,y,X);
}
// lagrange interpolated polynomial
double f2(double X) {
        double x[4]={0,1,2,3};
        double y[4]={exp(0),exp(1),exp(2),exp(3)};
        return lagrange(4,x,y,X);
}
// exact function to be integrated
double f3(double X) {
        return exp(X);
}
// clamped cubic spline function
double f4(double X) {
        double F0=1,F1=exp(3); // clamped conditions
    double x[4]={0,1,2,3};
    double y[4]={exp(0),exp(1),exp(2),exp(3)};
        return clamped(4,x,y,F0,F1,X);
}
// main program to do our job
int main()
{
        printf("Integrating the interpolant via simpsons 1/3 rule\n");

        printf("\nusing natural cubic spline: %lf\n",simpson(f1,0,3));
        printf("using lagrange polynomial: %lf\n",simpson(f2,0,3));
        printf("using exact function: %lf\n",simpson(f3,0,3));
        printf("using clamped cubic spline: %lf\n",simpson(f4,0,3));
}
```

OUTPUT:

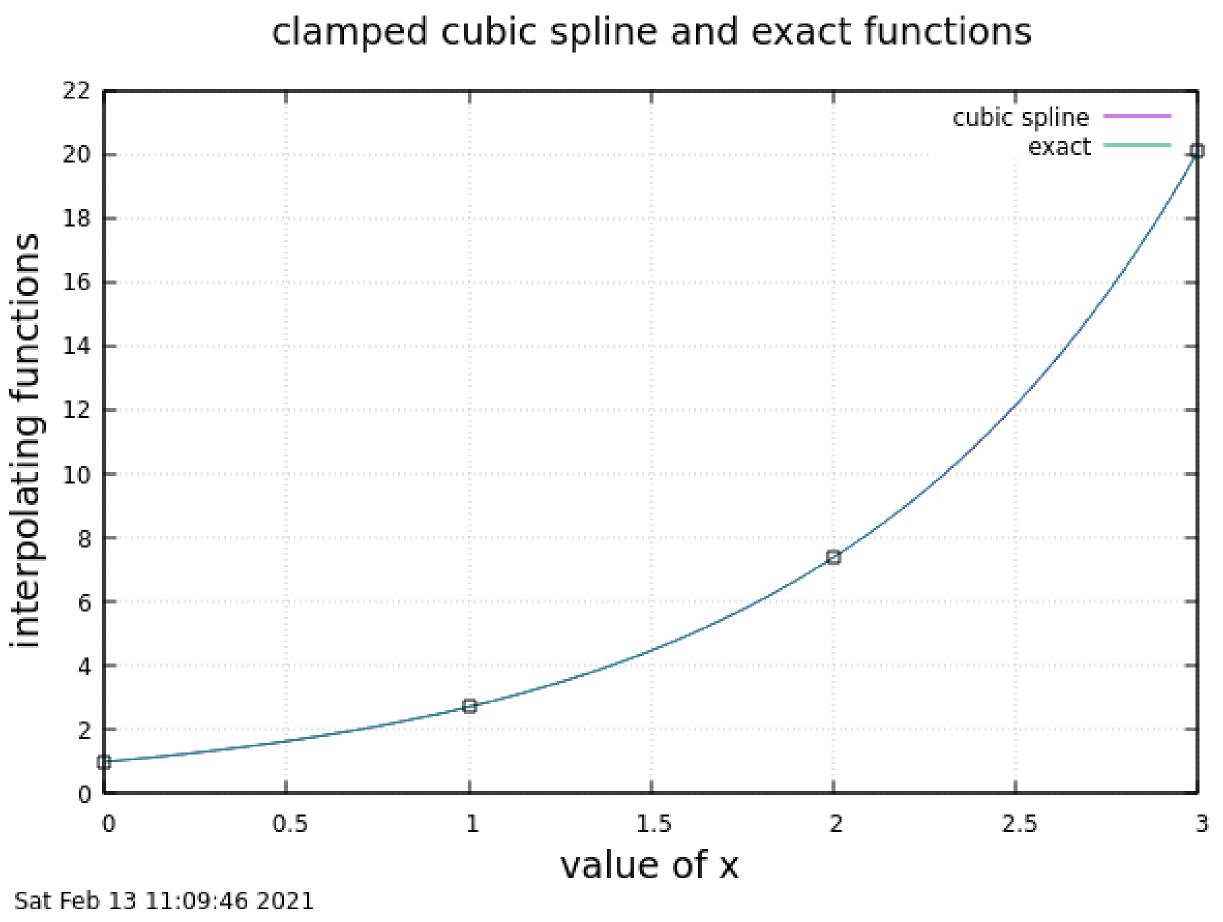
```
Integrating the interpolant via simpsons 1/3 rule

using natural cubic spline: 19.542529
using lagrange polynomial: 19.277832
using exact function: 19.085556
using clamped cubic spline: 19.059656
```

PROBLEM 4 :

In [ ]:
```
// problem 4
// make sure "csplines.c" file is in the same directory
#include<stdio.h>
#include<math.h>
#include"csplines.c"
// main program to do our job
int main()
```

```
{
    double X,F0=1,F1=exp(3); // clamped conditions
    double x[]={0,1,2,3};
    double y[]={exp(0),exp(1),exp(2),exp(3)};
    FILE*fp=NULL;
    fp=fopen("clamp.txt","w");
    for (X=0;X<3;X+=0.01)
    {
        fprintf(fp,"%lf\t%lf\t%lf\n",X,clamped(4,x,y,F0,F1,X),exp(X));
    }
}
```

OUTPUT:

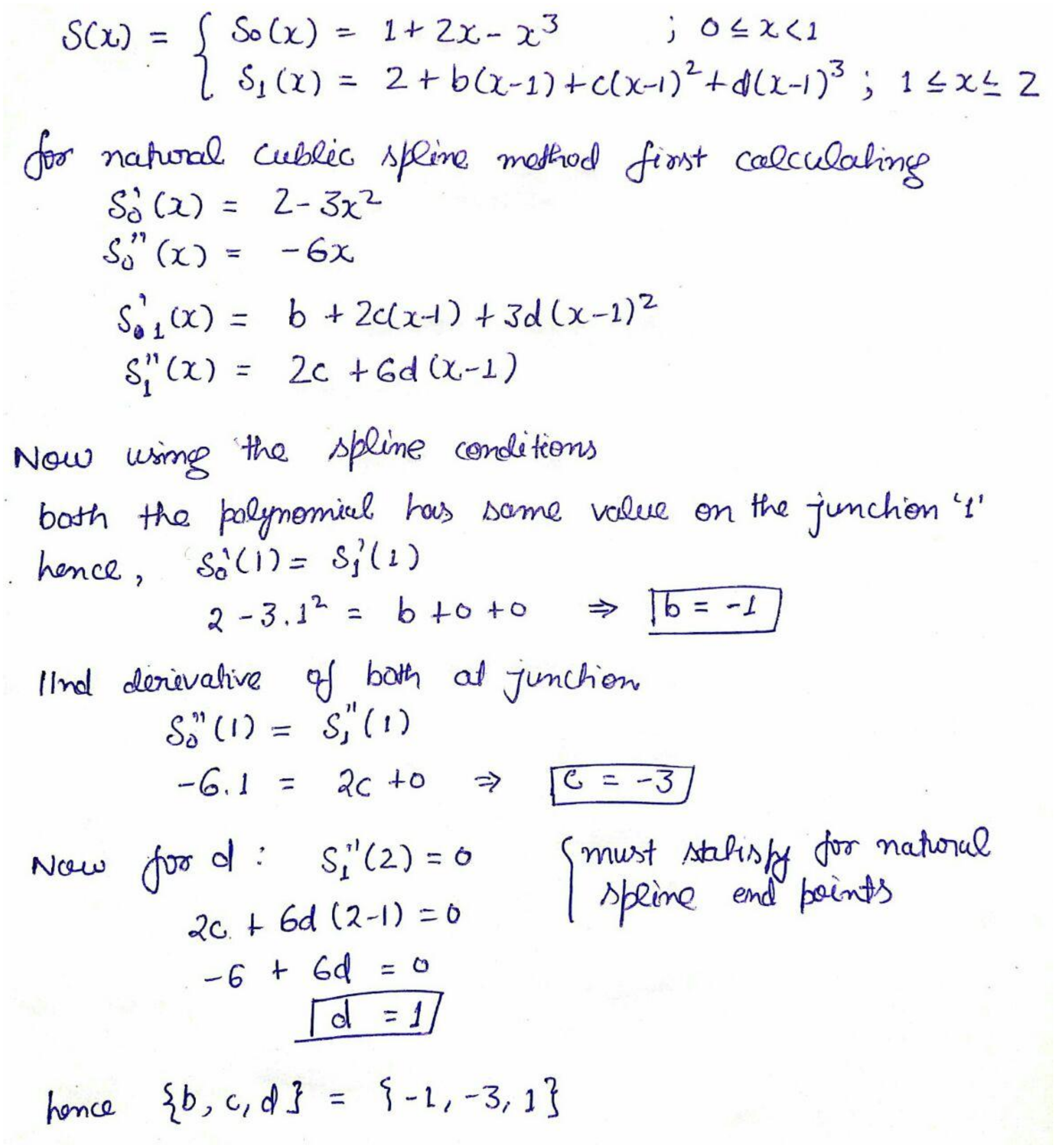


Comments or Discussions:

1. In these cases where we interpolated 4 datapoints using lagrange polynomial, natural cubic spline and clamped cubic spline the result precision order is natural cspline < lagrange < clamped cspline. Also when the integration performed using simpson's 1/3 rule similiar precision is noted.
2. Here we have used lagrange for order three polynomial which gave us satisfactory result but as we go for higher order polynomial for large datapoints it starts giving oscillatory behaviour which can be seen in the last problem of assignment while the natural cspline is giving promising result for large datapoints since it is generating order three polynomial for each spline.
3. **If we use lagrange for more than order three polynomial we'll most probably compromise the result.**

4. In the case of clamped cubic spline we are providing two additional true information which lead the result closer to the actual value. **If somehow we are able to extract such true conditions in our experiment we'll get much accurate result for our cubic spline problem.**

PROBLEM 6 :



$$S(x) = \begin{cases} S_0(x) = 1 + 2x - x^3 & ; \ 0 \le x < 1 \\ S_1(x) = 2 + b(x-1) + c(x-1)^2 + d(x-1)^3 & ; \ 1 \le x \le 2 \end{cases}$$

for natural cublic spline method first calculating

$$S_0'(x) = 2 - 3x^2$$
$$S_0''(x) = -6x$$

$$S_1'(x) = b + 2c(x-1) + 3d(x-1)^2$$
$$S_1''(x) = 2c + 6d(x-1)$$

Now using the spline conditions

both the polynomial has same value on the junction '1'

hence, $S_0'(1) = S_1'(1)$

$$2 - 3.1^2 = b + 0 + 0 \Rightarrow \boxed{b = -1}$$

IInd derivative of both at junction

$$S_0''(1) = S_1''(1)$$
$$-6.1 = 2c + 0 \Rightarrow \boxed{c = -3}$$

Now for d : $S_1''(2) = 0$ { must statisfy for natural spline end points

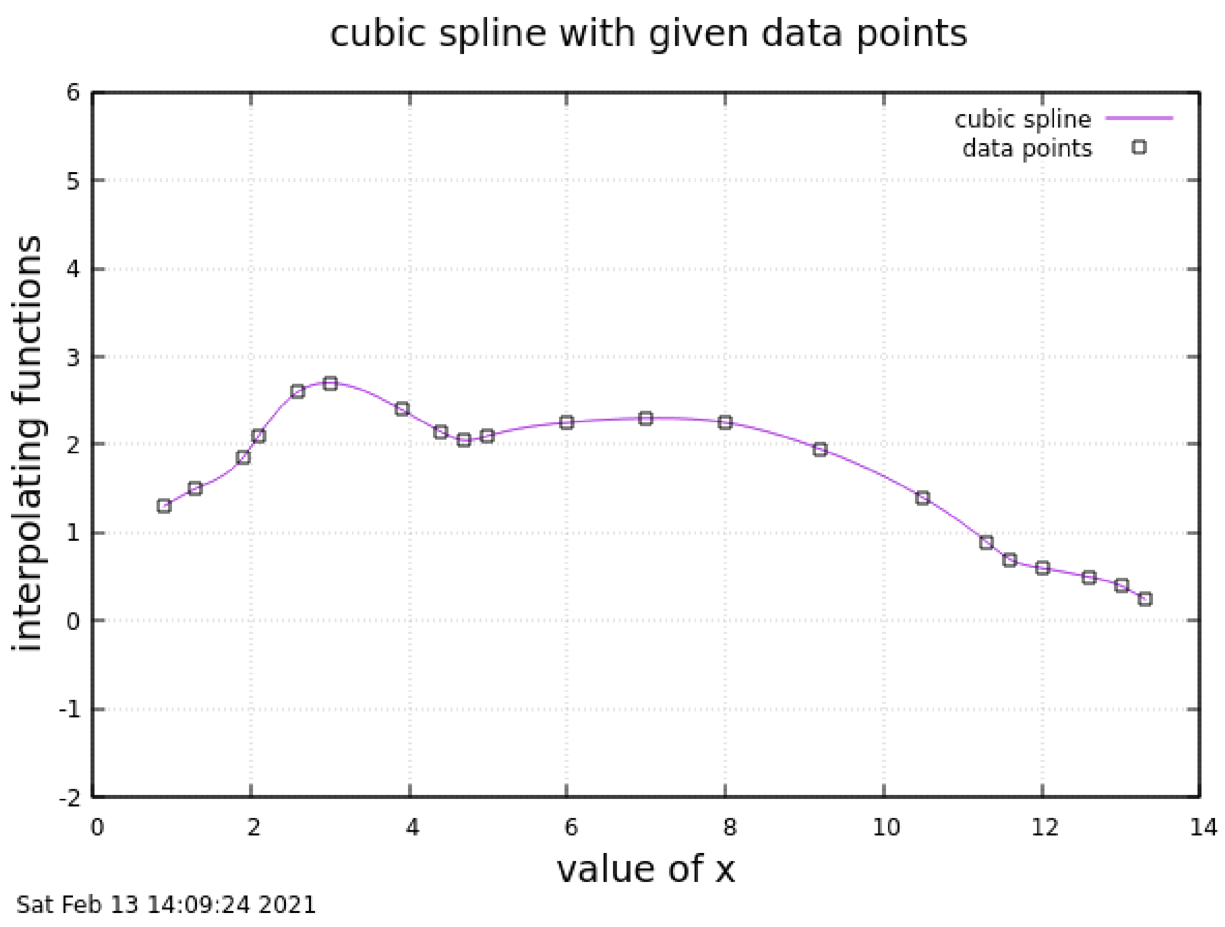
$$2c + 6d(2-1) = 0$$
$$-6 + 6d = 0$$
$$\boxed{d = 1}$$

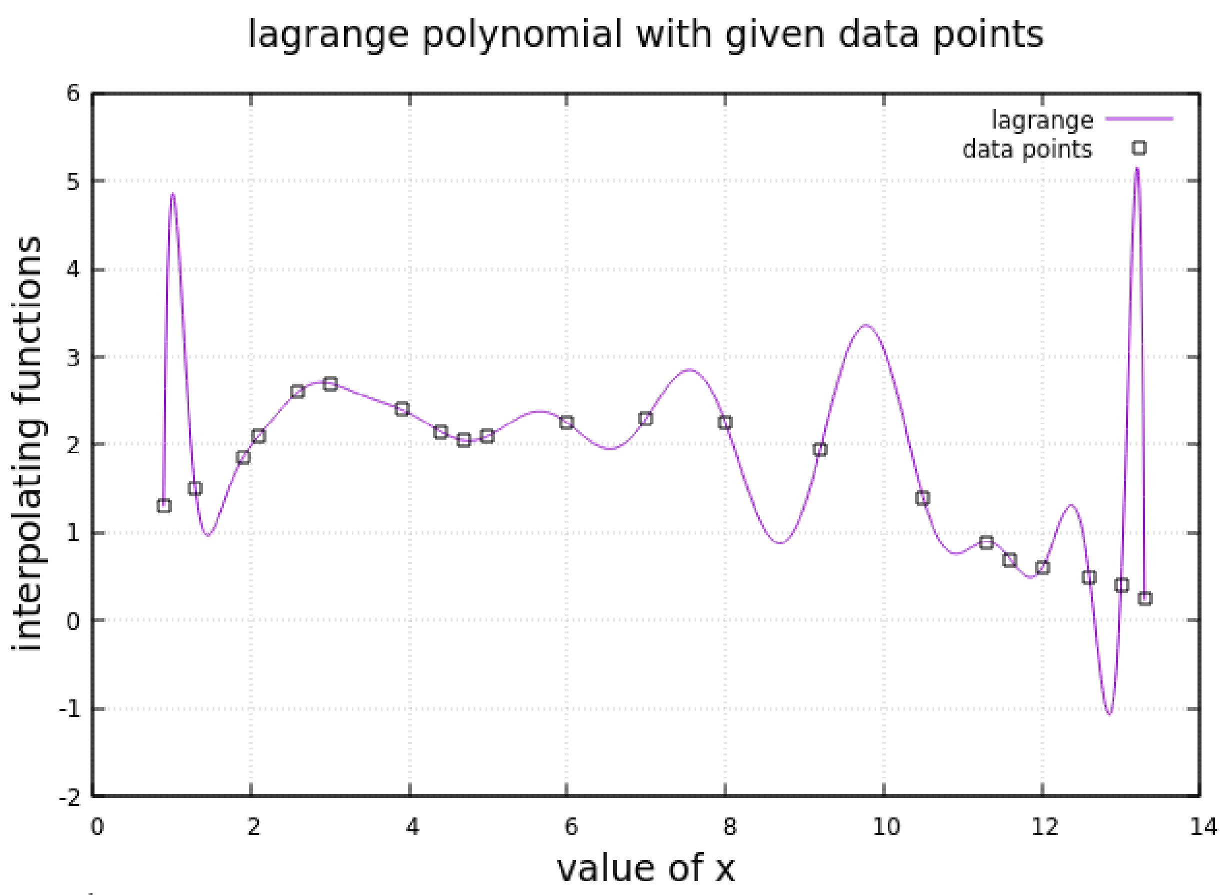
hence $\{b, c, d\} = \{-1, -3, 1\}$

PROBLEM 7 :

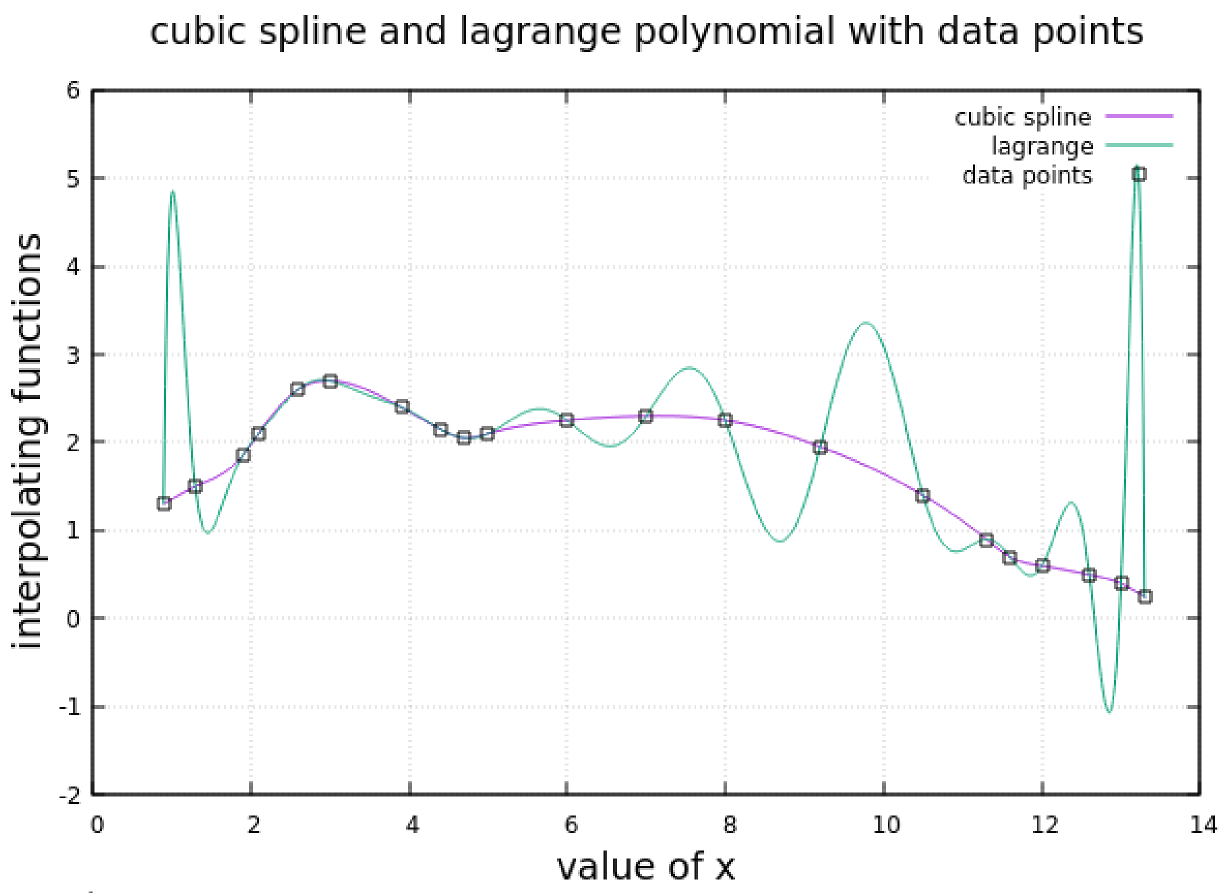
In [ ]:
```
// problem 7
// make sure "csplines.c" file is in the same directory
#include<stdio.h>
#include<math.h>
#include"csplines.c"
// main program to do our job
int main()
{
        double X;
        double x[21]= {0.9, 1.3, 1.9, 2.1, 2.6, 3.0, 3.9, 4.4, 4.7, 5.0, 6.0,
                   7.0, 8.0, 9.2, 10.5, 11.3, 11.6, 12.0, 12.6, 13.0, 13.3};
    double fx[21] = {1.3, 1.5, 1.85, 2.1, 2.6, 2.7, 2.4, 2.15, 2.05, 2.1,
                     2.25, 2.3, 2.25, 1.95, 1.4, 0.9, 0.7, 0.6, 0.5, 0.4, 0.25};
        FILE*fp=NULL;
        fp=fopen("7.txt","w");
        for (X=0.9;X<=13.3;X+=0.01)
        {
                fprintf(fp,"%lf\t%lf\t%lf\n",X,csplines(21,x,fx,X),
                lagrange(21,x,fx,X));
        }
}
```

OUTPUT:





Sat Feb 13 11:54:06 2021

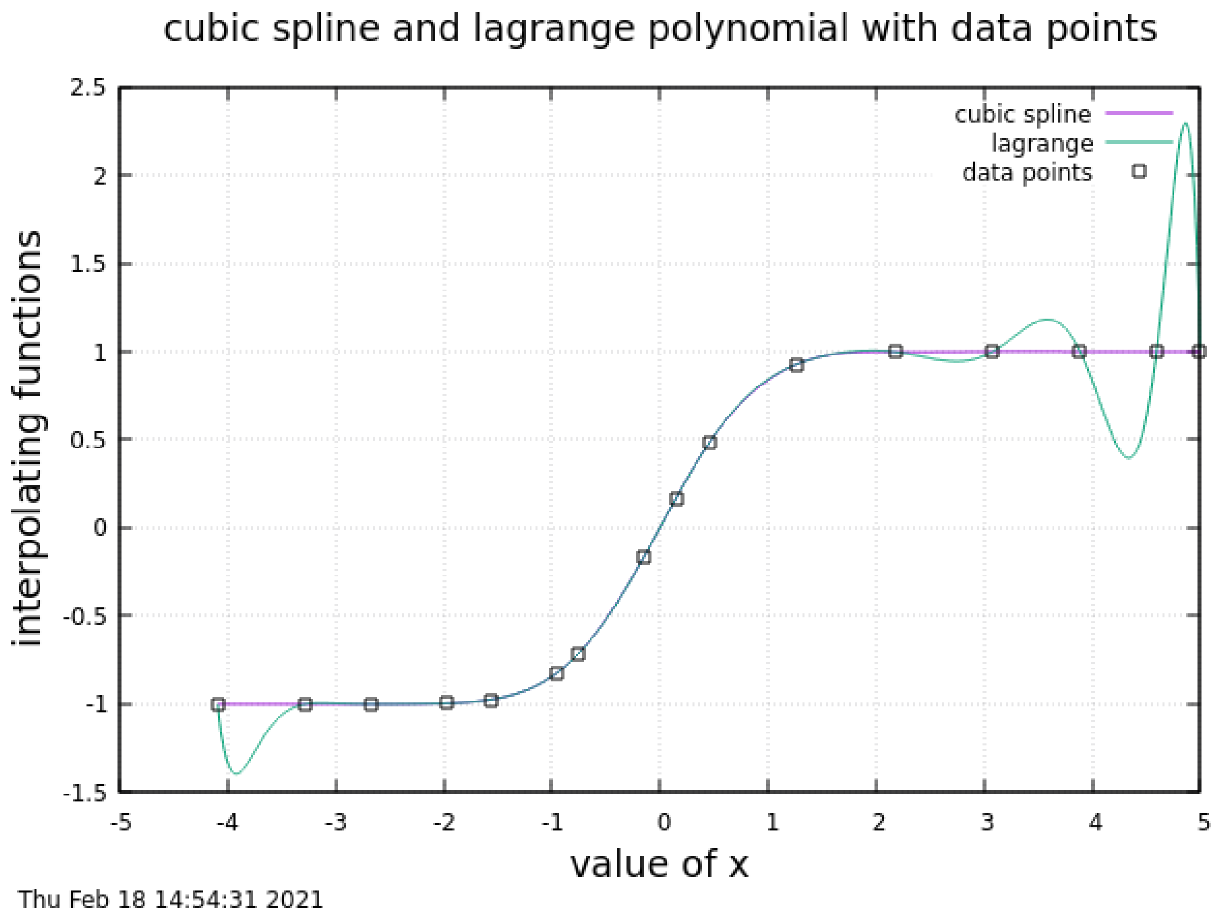


Sat Feb 13 11:52:49 2021

PROBLEM 8 :

In [ ]:
```
// problem 8
// make sure "csplines.c" file is in the same directory
#include<stdio.h>
#include<math.h>
#include"csplines.c"
// main program to do our job
int main()
{
        double X;
        double x[]={-4.09091,-3.28283,-2.67677,-1.9697,-1.56566,-0.959596,-0.757
        -0.151515,0.151515,0.454545,1.26263,2.17172,3.08081,3.88889,4.59596,5};
        double y[]={-1,-0.999997,-0.999847,-0.994657,-0.973183,-0.825242,-0.7159
        -0.169667,0.169667,0.479662,0.92584,0.997869,0.999987,1,1,1};
        FILE*fp=NULL;
        fp=fopen("8.txt","w");
        for (X=x[0];X<=x[15];X+=0.01)
        {
                fprintf(fp,"%lf\t%lf\t%lf\n",X,csplines(16,x,y,X),
        lagrange(16,x,y,X));
        }
}
```

OUTPUT:



Cubic spline interpolation and why it wins ?

Suppose we have empirical data $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ that was obtained from some experiment. We have strong reason to believe y is related to x by some smooth function, but we do not know what that function is. We would like to find a function that can be used to approximate values of y for given values of x between the given data points. There are many ways to approach this problem, but the three most common are **least squares, Lagrange interpolation, and cubic splines**. Each of these methods has advantages and limitations compared to the other two.

Least Squares Method: The biggest advantage is that it can use all of the data, even if some of the x values are repeated with different corresponding values of y. This is not an unusual situation if an experiment is repeated several times. The other two methods require that the x values be distinct. A disadvantage of the least squares method is that you must know the nature of the function you are trying to approximate; that is, is the function a line, an exponential, a quadratic, etc.?

Lagrange Interpolation: The main advantage is that it is based on a fairly straight forward formula that is relatively easy to program. Given n nodes, it produces a polynomial of degree $n-1$. It does, however, have several disadvantages. For one thing, it tends to be quite sensitive to the accuracy of the data. Even a small error in a few of the y values (not at all uncommon in empirical data) can cause large errors in an approximation. This problem becomes more pronounced as the degree of the polynomial increases. The rule of thumb is to not use Lagrange polynomials of degree greater than three. A common practice is to write programs for Lagrange polynomials that will use only two nodes to the left and two to the right of the point at which the function is to be approximated. In that way, only four nodes are being used so the degree of the polynomial will be three. But, of course, that means that the rest of the data is being ignored.

Cubic Spline Interpolation: Although it still has the restriction that the x values must be distinct, one of the advantages of the cubic spline method is that it uses all of the data. The other big advantage is that it is not as sensitive to minor errors in the data as Lagrange polynomials. Given n nodes, the cubic spline is a sequence of $n-1$ cubic polynomials, one between each pair of adjacent nodes. These polynomials are such that they fit together smoothly at the nodes; not only do the values agree at the nodes, but the first and second derivatives also agree at the nodes. Finding the coefficients for the polynomials involves solving a system of $n-2$ equations for n unknowns. That means there are two degrees of freedom, so two of the unknowns can be chosen as desired. *This leads to two different commonly used splines, the natural or free spline and the clamped spline. In the case of the free spline the two parameters that are chosen is that the second derivatives at x_1 and x_n are set to zero. In the case of the clamped spline we must know (or be able to estimate) the derivatives of the function we are approximating at x_1 and x_n. We then use those as the derivatives of the spline at those points.* There are, of course, a variety of ways those derivatives can be estimated from the given nodes.

Source: https://resources.thiel.edu/mathproject/index.htm