Final Project

Group: 1

Numerical Methods

Team members:

● Hugo David Franco Ávila	ISC	A01654856
● Roberto Carlos Guzmán Cortés	ISC	A01702388
● Manuel Flores Ramírez	ISC	A01703912

Teacher: Elizabeth Chávez

16/11/2018

Index

Introduction

Splines are used as a technique to simplify polynomial approximations. When approximating a curve to a set of data, there are many ways to generate the polynomial that will run through all the points, like Newton or Lagrange interpolation, however, these methods get a polynomial with a high degree, which can be messy to work with, and sometimes presents oscillations (even though the curve runs through the data points, the value of the polynomial for values in between can differ greatly from the actual or desirable values). Splines produce multiple low degree polynomials which can be easier to work with, and to perform operations like differentiation.

The term “spline” comes from a rubber ruler that would be bent to create the best mechanical drawings of boats, later on mathematicians realized that by using these bending material things they were actually getting the best way to connect the dots with the least amount of physical bending which was described in a scientific paper in the late 1960’s. In the modern day, splines are used as a mathematical method to find the best approximation of a polynomial with multiple data points as a base, and it is a method widely used in different real world applications.

The most popular splines methods are these:

- Linear
- **Cubic**
- Monotone
- Hermite
- Akima
- Catmull-Rom

In the cubic method the number of polynomials used to approximate the curve are n-1, n being the number of data points that we have. It is called cubic spline since we only use polynomials that are of 3rd degree.

In this report, we are going to use a set of data points “[0.1 10;0.2 5;0.5 2;1 1; 2 0.5; 5 0.2; 10 0.1]” for our demonstration of how splines work. This points are from the real function $f(x) = 1/x$, however most of the time as engineers we do not have the equation and we can only approximate a curve with numerical methods. As a point of comparison, we are going to analyze the same set of data points but with Lagrange interpolation, which is noticeably more computationally expensive and harder to work with, to see if there is any tangible benefit to using splines.

Developing

The three main ways of spline approximation are linear, quadratic and cubic. Each of them has their pros and cons, and situations where they fit better than the other.

Linear splines are the easiest type, and has the least restrictions. It is basically "connecting the dots" with the data in the graph, this method isn't too useful because even though the function is continuous, it is not differentiable at the points where the splines form, and this severely impacts what we can do with the data points.

Quadratic splines tries to fit a curve or a parabola between two points on a graph, and the results are better than with linear splines, however, there are some considerations. The first is that it is objectively more complex because the curve has to pass through the points and needs to have the same first derivative (or slope) for both polynomials in the same point.

Cubic splines are the ones we'll be using to solve the problem described in the introduction, they are also the most widely used spline technique because of the flexibility they provide, the simplicity to calculate them, and their accuracy.
Cubic splines don't require the points to have the same distance between them, although they need to be in ascendant order for the algorithm to correctly calculate the polynomials. The conditions that need to be fulfilled for the interpolation to work are that the first derivative in the points at the extreme of every subinterval is the same when evaluated with the left and right spline, and also, that the second derivative has the same value for both sides. What these equalities do, is that the function at the extreme of the intervals grows or decreases at the same rate (first derivative), and that the concavity or the way the curve is drawn (upwards or downwards) is the same (second derivative). By doing this over the whole interval, the result is a graph that will look smooth in every point of the plane, and gives a better estimation for an interpolation. Another advantage of cubic splines over Newton or Lagrange interpolations is that it can also be used to extrapolate, that means, predicting where a point would be outside of the main interval.

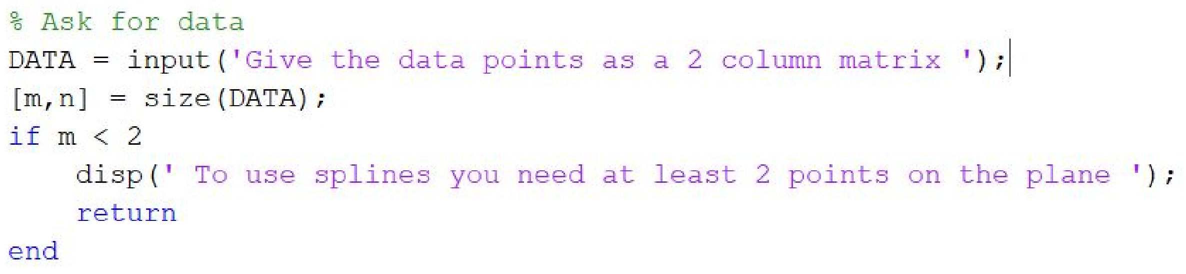
In this section we are going to describe our algorithm, and how it is applied to the set of data defined in the introduction of this report.

Algorithm
To program cubic splines interpolation in MatLab, we first needed to settle what type of cubic spline we were going to do. The algorithm needs a set of values called σ (sigma) which are essentially the second derivative of each equation or polynomial,

and depending on how you calculate the first and the last of these values is the type of spline you are going to use.
There are four different alternatives: setting the value of σ0 and σn to 0, in which case it would be called a natural cubic spline; setting σ0 and σn with the same values as σ1 and σn-1 respectively; establishing the second derivative as a linear equation and defining σ0 and σn as a linear equation; and directly giving a value to the first derivative on both points to solve a linear equation.
We chose to do natural cubic splines, because they are the easiest to do, provide a good approximation and are the least computationally expensive.

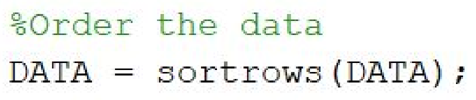
The algorithm is as follows:

1. We ask the user for a matrix with the data points we are going to use for the approximation, the 2 conditions for the program to continue is that the number of columns is exactly 2, and that the number of rows is bigger than 1, because we need at least 2 points to give an Spline approximation



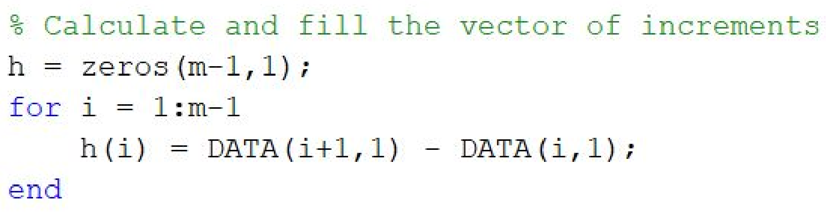
```
% Ask for data
DATA = input('Give the data points as a 2 column matrix ');
[m,n] = size(DATA);
if m < 2
    disp(' To use splines you need at least 2 points on the plane ');
    return
end
```

2. We need to sort the matrix so it is in ascending order with respect to the x values, this step is crucial because if we don't do it, the results will vary greatly.



```
%Order the data
DATA = sortrows(DATA);
```

3. The next step is to calculate the "vector of increments", this is a vector that holds the difference of two consecutive x values, $h(i) = x(i+1) - x(i)$. This is one of the reasons the x values have to be sorted, so each of these differences is a positive value.



```
% Calculate and fill the vector of increments
h = zeros(m-1,1);
for i = 1:m-1
    h(i) = DATA(i+1,1) - DATA(i,1);
end
```

4. Next comes what is probably the "hardest" part of the method, solving a system a of m-2 equations to get the values of the sigma vector. It's m-2 because 2 of these values are already known (σ0 and σn). The matrix of coefficients is given by a set of equations which are a little complicated to get

to, but describe the conditions of the equality of the first and second derivative between subintervals. The matrix is squared, and also called a tridiagonal matrix, because every element in the matrix is 0 except for the main diagonal and the 2 closest diagonals to it.

```
%Fill matrix of coefficients
for i = 1:m-2
    for j = 1:m-2
        if i == j
            coeff(i,j) = 2*(h(i)+h(i+1));
        end
        if (i - j) == 1
            coeff(i,j) = h(j+1);
        end
        if (j - i) == 1
            coeff(i,j) = h(i+1);
        end
    end
end
```

```
coeff =

    0.8000    0.3000         0         0         0
    0.3000    1.6000    0.5000         0         0
         0    0.5000    3.0000    1.0000         0
         0         0    1.0000    8.0000    3.0000
         0         0         0    3.0000   16.0000
```

5. Given that we need to solve a system of linear equations, we need to get a vector of constants, and after that, we can calculate vector σ by using any of the methods seen in class to solve linear equations, for this application, we are going to use the inverse matrix method, which given the characteristics of the matrix is not to difficult too use. σ0 and σn will have a value of 0 for the reasons we mentioned earlier.

```
%Vector of constants
con = zeros(m-2,1);
%Fill vector of constants
for i = 1:m-2
    con(i) = 6*((DATA(i+2,2)-DATA(i+1,2))/(h(i+1)) - (DATA(i+1,2)-DATA(i,2))/(h(i)));
end
```

```
% Now we have to calculate the values of the sigmas
%Create and initialize with 0 vector sigma
sigma = 0;
sig = coeff\con;
sigma = [sigma;sig;sigma];
```

6. After this we can finally calculate the polynomials we are going to use for our approximations. To implement this we needed to find a way to store function handles in a vector or a similar data structure, and we were able to do it once we found a structure in MatLab called cell, that can hold anything, from

functions to vectors to integers, etc. And we used it to store every equation of our approximations.

```
%-----The block bellow creates a cell array that contains the polynomials
q = cell(m-1,1);
for i = 1:m-1
    q{i,1} = @(x)  (sigma(i)/6)*(((DATA(i+1,1)-x).^3)/h(i)-h(i)*((DATA(i+1,

end
%-----------------------------------------------------------------
```

```
q =

  6×1 cell array

    {function_handle}
    {function_handle}
    {function_handle}
    {function_handle}
    {function_handle}
    {function_handle}
```

The equation used to calculate the equation is really long and it's not going to fit, so we will use the equation given to us in the article we read.

$$q_k(x) = \frac{\sigma_k}{6}\left[\frac{(x_{k+1}-x)^3}{h_k} - h_k(x_{k+1}-x)\right] + \frac{\sigma_{k+1}}{6}\left[\frac{(x-x_k)^3}{h_k} - h_k(x-x_k)\right] + y_k\left[\frac{x_{k+1}-x}{h_k}\right] + y_{k+1}\left[\frac{x-x_k}{h_k}\right],$$

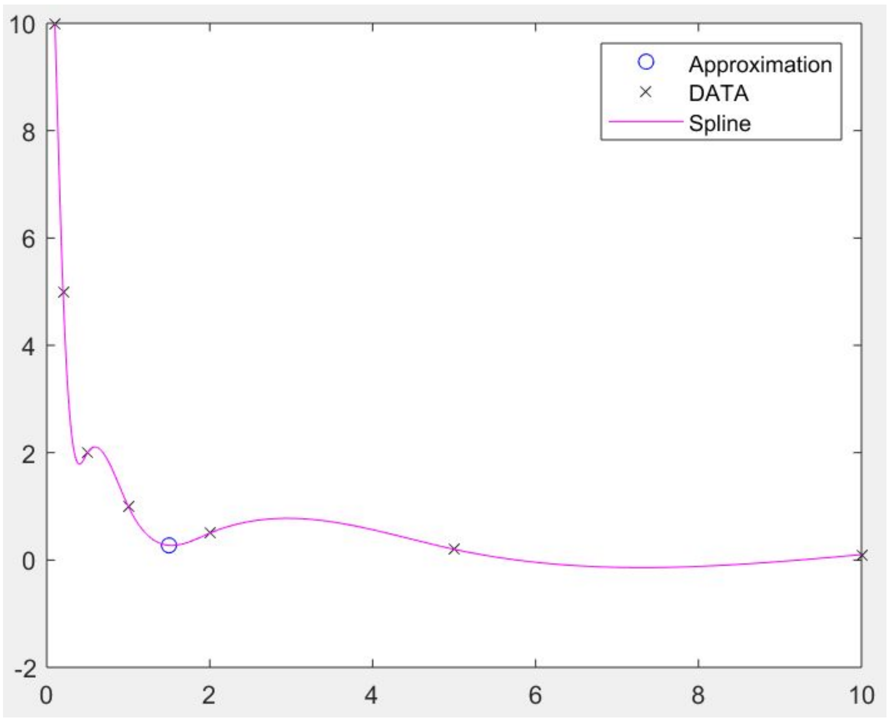
Each of the values represented have been described previously.

7. The algorithm is finished. We have the functions stored and we can give it a value and we will get the approximation using the spline for that interval.

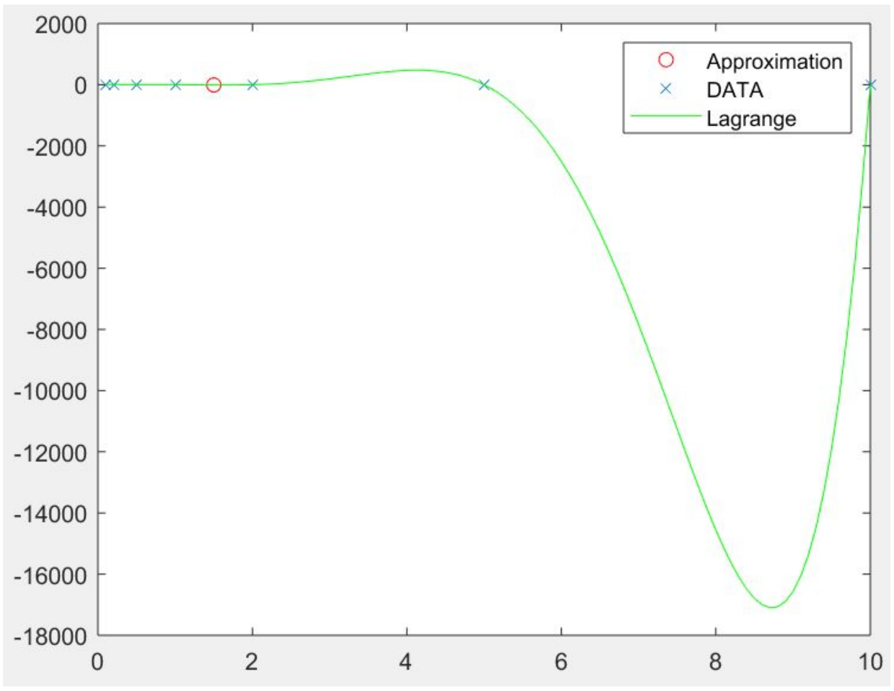
```
con = 1;
ind = 2;
xu = input(' Write x value in which you want to see the value of the approximation: ');
%Check the polynomial that xu has to use
while xu > DATA(ind,1)
    con = con + 1;
    ind = ind + 1;
end
yu = q{con}(xu)
```

Additionally, we chose to plot the graph of the splines to see if the approximation was similar to the original function $f(x) = 1/x$. The graph will include legends indicating what is what in the graph, and our approximation. We also graphed the results for the same set of points using Lagrange interpolation and the real graph.

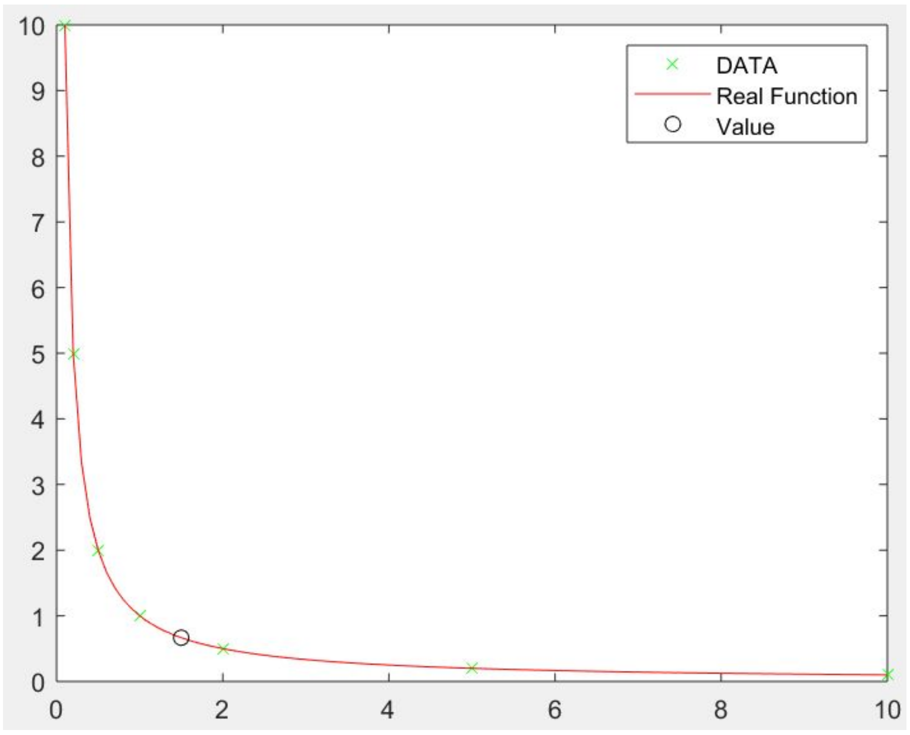
Splines



Lagrange



Real function

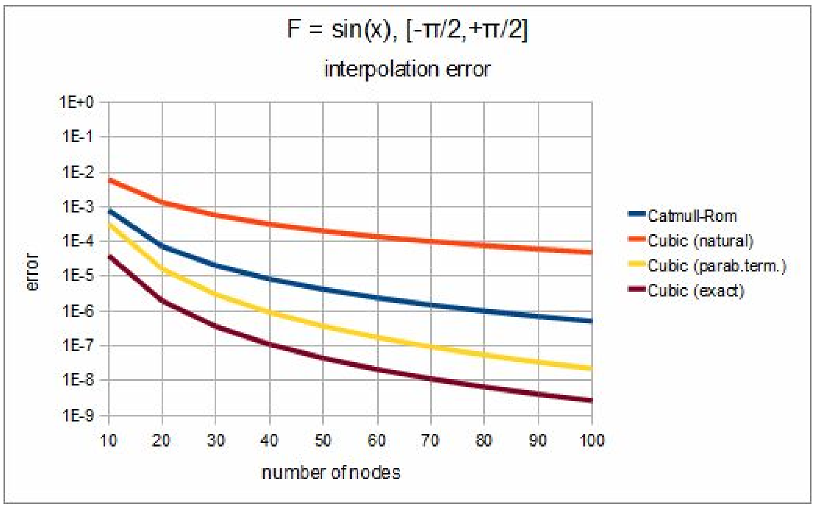


We can easily see Splines are much closer to the real graph than Lagrange interpolation, which is expected, because of the type of function. Lagrange struggles with anything other than a normal polynomial, whereas Splines with their simpler approach can be used for virtually any set of data.

Conclusions

According to **alglib**, the cross-platform of numerical analysis and data processing library, this method is fast, efficient and stable. In addition, due to the exact values of the first derivative in both boundaries are known, this spline method is called *clamped spline*, or *spline with exact boundary conditions*. This spline has interpolation error of $O(h^4)$.

The following graph shows us the advantages of using these method compared with the other “Cubic splines” methods.



Splines are much easier to use compared to the high degree polynomials given by other numerical methods like Lagrange and Newton, and their behaviour is more easily defined because they only work with two points at a time.

The biggest issue with our implementation of Splines is definitely the use of the Cell structure, it is probably very expensive computationally and we are not sure if there is any other way to solve our need to hold functions in a vector. Perhaps there is, or maybe MatLab isn’t the most efficient language for the programming of Splines. We found many Python implementations so that is probably the easiest/most efficient language to program them.

References

- Spline interpolation and fitting - ALGLIB, C++ and C# library", *Alglib.net*, 2018. [Online]. Available: http://www.alglib.net/interpolation/spline3.php. [Accessed: 15- Nov- 2018].
- García, J. (2000). *Tutorial de Analisis Numerico Interpolacion : Splines cubicos*. 1st ed. [ebook] Tafira, España, pp.2-22. Available at: https://www.u-cursos.cl/ingenieria/2007/2/MA33A/4/material_docente/bajar?id_material=145305 [Accessed 15 Nov. 2018].
- Diaz, W. (1998). Splines cúbicos. *UV.* Available at https://www.uv.es/~diaz/mn/node40.html