U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

TRABALHO 3 - ANÁLISE NUMÉRICA

João Lopes - MIERSI - up201805078

João Santos - MIERSI - up201206800

Manuel Sá - MIERSI - up201805273

Roberto Leite - MIERSI – up201805421

Maio 2021

INTRODUÇÃO

Este é o terceiro trabalho de analise numérica. O objetivo é resolver exercícios acerca da matéria de interpolação e de splines. Para a resolução de exercícios, usamos a linguagem de programação C. A análise dos resultados encontra-se em baixo, bem como os mesmos obtidos.

Exercício 1

A) No primeiro exercício, é-nos apresentada uma tabela. Esta tabela é composta com 12 pontos e 2 colunas, mês, evaporação. É-nos pedido para construir o Polinômio interpolador e o spline cubico natural de acordo com o conjunto de pontos.

```c
#include<stdio.h>

int main(){

  float evap[12] ={8.6, 7.0, 6.4, 4.0, 2.8, 1.8, 1.8, 2.3, 3.2, 4.7, 6.2, 7.9};

    printf("polinomio:\n");


    printf("((x-%d)",2);
        for(int j=3;j<=12;j++){
            printf("*(x-%d)",j);
        }

        printf(")/");

        printf("(");

        printf("(1-2)");
        for(int j=3;j<=12;j++){
            printf("*(1-%d)",j);
        }

        printf(")");
        printf("*%.1f+", evap[0]);
```

```
for(int i=2;i<=12;i++){

    printf("(");
    if(i!=1) printf("(x-%d)",1);
    for(int j=2;j<=12;j++){
        if(i!=j) printf("*(x-%d)",j);
    }

    printf(")/");

    printf("(");
    if(i!=1) printf("(%d-%d)",i,1);
    for(int j=2;j<=12;j++){
        if(i!=j) printf("*(%d-%d)",i,j);
    }

    printf(")");


   if(i!=12)  printf("*%.1f+", evap[i-1]);


   else printf("*%.1f", evap[i-1]);
}


return 0;
```

Este programa dá-nos como resultado o Polinômio interpolador dos 12 pontos acima constados. Para o cálculo, utilizamos o método de Lagrange. A função gerada é a seguinte: P11(x)=

$$\frac{(x-2)*(x-3)*(x-4)*(x-5)*(x-6)*(x-7)*(x-8)*(x-9)*(x-10)*(x-11)*(x-12)}{(1-2)*(1-3)*(1-4)*(1-5)*(1-6)*(1-7)*(1-8)*(1-9)*(1-10)*(1-11)*(1-12)} *8.6+$$

$$\frac{(x-1)*(x-3)*(x-4)*(x-5)*(x-6)*(x-7)*(x-8)*(x-9)*(x-10)*(x-11)*(x-12))}{(2-1)*(2-3)*(2-4)*(2-5)*(2-6)*(2-7)*(2-8)*(2-9)*(2-10)*(2-11)*(2-12)} *7.0+$$

$$\frac{((x-1)*(x-2)*(x-4)*(x-5)*(x-6)*(x-7)*(x-8)*(x-9)*(x-10)*(x-11)*(x-12))}{((3-1)*(3-2)*(3-4)*(3-5)*(3-6)*(3-7)*(3-8)*(3-9)*(3-10)*(3-11)*(3-12))} *6.4+$$

$$\frac{(x-1)*(x-2)*(x-3)*(x-5)*(x-6)*(x-7)*(x-8)*(x-9)*(x-10)*(x-11)*(x-12)}{((4-1)*(4-2)*(4-3)*(4-5)*(4-6)*(4-7)*(4-8)*(4-9)*(4-10)*(4-11)*(4-12))} *4.0+$$

$$\frac{((x-1)*(x-2)*(x-3)*(x-4)*(x-6)*(x-7)*(x-8)*(x-9)*(x-10)*(x-11)*(x-12))}{((5-1)*(5-2)*(5-3)*(5-4)*(5-6)*(5-7)*(5-8)*(5-9)*(5-10)*(5-11)*(5-12))} *2.8+$$

$$\frac{((x-1)*(x-2)*(x-3)*(x-4)*(x-5)*(x-7)*(x-8)*(x-9)*(x-10)*(x-11)*(x-12))}{((6-1)*(6-2)*(6-3)*(6-4)*(6-5)*(6-7)*(6-8)*(6-9)*(6-10)*(6-11)*(6-12))} *1.8+$$

$$\frac{((x-1)*(x-2)*(x-3)*(x-4)*(x-5)*(x-6)*(x-8)*(x-9)*(x-10)*(x-11)*(x-12))}{((7-1)*(7-2)*(7-3)*(7-4)*(7-5)*(7-6)*(7-8)*(7-9)*(7-10)*(7-11)*(7-12))} *1.8+$$

$$\frac{((x-1)*(x-2)*(x-3)*(x-4)*(x-5)*(x-6)*(x-7)*(x-9)*(x-10)*(x-11)*(x-12))}{((8-1)*(8-2)*(8-3)*(8-4)*(8-5)*(8-6)*(8-7)*(8-9)*(8-10)*(8-11)*(8-12))} *2.3+$$

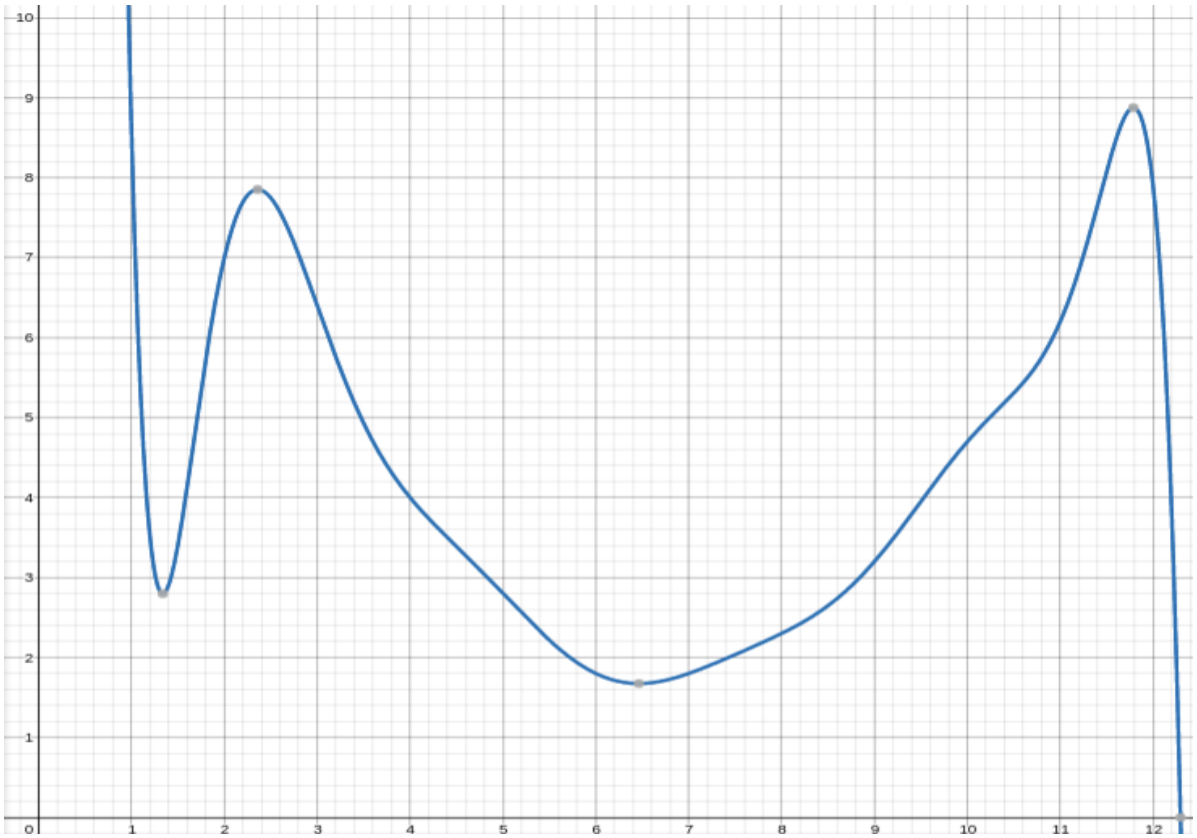
$$\frac{((x-1)*(x-2)*(x-3)*(x-4)*(x-5)*(x-6)*(x-7)*(x-8)*(x-10)*(x-11)*(x-12))}{((9-1)*(9-2)*(9-3)*(9-4)*(9-5)*(9-6)*(9-7)*(9-8)*(9-10)*(9-11)*(9-12))} *3.2+$$

$$\frac{((x-1)*(x-2)*(x-3)*(x-4)*(x-5)*(x-6)*(x-7)*(x-8)*(x-9)*(x-11)*(x-12))}{((10-1)*(10-2)*(10-3)*(10-4)*(10-5)*(10-6)*(10-7)*(10-8)*(10-9)*(10-11)*(10-12))} *4.7+$$

$$\frac{((x-1)*(x-2)*(x-3)*(x-4)*(x-5)*(x-6)*(x-7)*(x-8)*(x-9)*(x-10)*(x-12))}{((11-1)*(11-2)*(11-3)*(11-4)*(11-5)*(11-6)*(11-7)*(11-8)*(11-9)*(11-10)*(11-12))} *6.2+$$

$$\frac{((x-1)*(x-2)*(x-3)*(x-4)*(x-5)*(x-6)*(x-7)*(x-8)*(x-9)*(x-10)*(x-11))}{((12-1)*(12-2)*(12-3)*(12-4)*(12-5)*(12-6)*(12-7)*(12-8)*(12-9)*(12-10)*(12-11))} *7.9$$

Com base neste resultado, obtivemos o seguinte gráfico de aproximação:



Relativamente ao spline cubico natural, utilizamos as seguintes linhas de código para imprimir um sistema de equações que representa uma parte da construção deste spline:

```c
#include <stdio.h>

void scn(){

    double ord[12] = {8.6,
           7.0,
           6.4,
           4.0,
           2.8,
           1.8,
           1.8,
           2.3,
           3.2,
           4.7,
           6.2,
           7.9};

    printf("M0 = 0\n"); //Para ser calculado o spline cubico natural
    for(int i=1 ; i<11 ; i++){
        printf("S%d(x) = (1/6)*M%d + (2/3)*M%d + (1/6)*M%d = %f\n",i ,i-1 ,i ,i+1 ,ord[i+1]-ord[i]-(ord[i]-ord[i-1]));
    }
    printf("M11 = 0\n");//Para ser calculado o spline cubico natural
}

int main(){
    scn();
}
```

Este programa printa como output o sistema de equações referido acima.

```
M0 = 0
S1(x) = (1/6)*M0 + (2/3)*M1 + (1/6)*M2 = 1.000000
S2(x) = (1/6)*M1 + (2/3)*M2 + (1/6)*M3 = -1.800000
S3(x) = (1/6)*M2 + (2/3)*M3 + (1/6)*M4 = 1.200000
S4(x) = (1/6)*M3 + (2/3)*M4 + (1/6)*M5 = 0.200000
S5(x) = (1/6)*M4 + (2/3)*M5 + (1/6)*M6 = 1.000000
S6(x) = (1/6)*M5 + (2/3)*M6 + (1/6)*M7 = 0.500000
S7(x) = (1/6)*M6 + (2/3)*M7 + (1/6)*M8 = 0.400000
S8(x) = (1/6)*M7 + (2/3)*M8 + (1/6)*M9 = 0.600000
S9(x) = (1/6)*M8 + (2/3)*M9 + (1/6)*M10 = 0.000000
S10(x) = (1/6)*M9 + (2/3)*M10 + (1/6)*M11 = 0.200000
M11 = 0
```

Nota:

$$\frac{h_i}{6}M_{i-1} + \frac{h_i + h_{i+1}}{3}M_i + \frac{h_{i+1}}{6}M_{i+1} = \frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i}, \quad i = 1, \ldots, n-1$$

Cada uma destas linhas é uma tradução da fórmula que está nos slides da cadeira, tendo em conta que a distancia das abcissas dos pontos é 1, então o valor de h assume-me sempre 1. As ordenadas dos pontos são representadas por *f* . Os valores de M0 e M11 são definidos como 0 devido ao facto de este ser um spline cubico **natural**.

Se não houver outra informação é habitual juntar ao sistema (1) as seguintes condições (anulamento da segunda derivada nos nós extremos)

$$S_1''(x_0) = M_0 = 0$$
$$S_n''(x_n) = M_n = 0$$

e o spline que se obtem nestas condições chama-se spline cúbico natural.

Usando uma calculadora online conseguimos obter os valores de M do sistema de equações já representado, sendo estes os seguintes:

M0 = 0

M1 = 2.52278

M2 = -4.09112

M3 = 3.04171

M4 = -0.87572

M5 = 1.653618

M6 = 0.25922

M7 = 0.30949

M8 = 0.902814

M9 = 0.320751

M10 = 0.380187

M11 = 0

Para a construção do spline, com base nos dados adquiridos acima, utilizamos a seguinte formula:

- Chega-se à seguinte expressão para $S_i(x)$:

$$S_i(x) = M_{i-1}\frac{(x_i - x)^3}{6h_i} + M_i\frac{(x - x_{i-1})^3}{6h_i} + \left(f_{i-1} - M_{i-1}\frac{h_i^2}{6}\right)\frac{x_i - x}{h_i} + \left(f_i - M_i\frac{h_i^2}{6}\right)\frac{x - x_{i-1}}{h_i}$$

Para então chegar ao spline cubico natural modificamos o código inicial para retornar o resultado obtido:

```c
#include <stdio.h>

void scn(){

    double ord[12] = {
        8.6,
        7.0,
        6.4,
        4.0,
        2.8,
        1.8,
        1.8,
        2.3,
        3.2,
        4.7,
        6.2,
        7.9
    };
    double m[12] = {
        0.0,
        2.52278,
        -4.09112,
        3.04171,
        -0.87572,
        1.653618,
        0.25922,
        0.30949,
        0.902814,
        0.320751,
        0.380187,
        0.0
    };

    for(int i=1 ; i<12 ; i++){
        printf("S%d(x) = %f*((%d-x)^3)/6 + %f*((x-%d)^3)/6 + %f*(%d-x) + %f*(x-%d)",i ,m[i-1] ,i+1 ,m[i] ,i, ord[i-1]-m[i-1]/6, i+1, ord[i]-m[i]/6, i);
        printf(", %d < x < %d\n", i, i+1);
    }
}

int main(){
    scn();
}
```

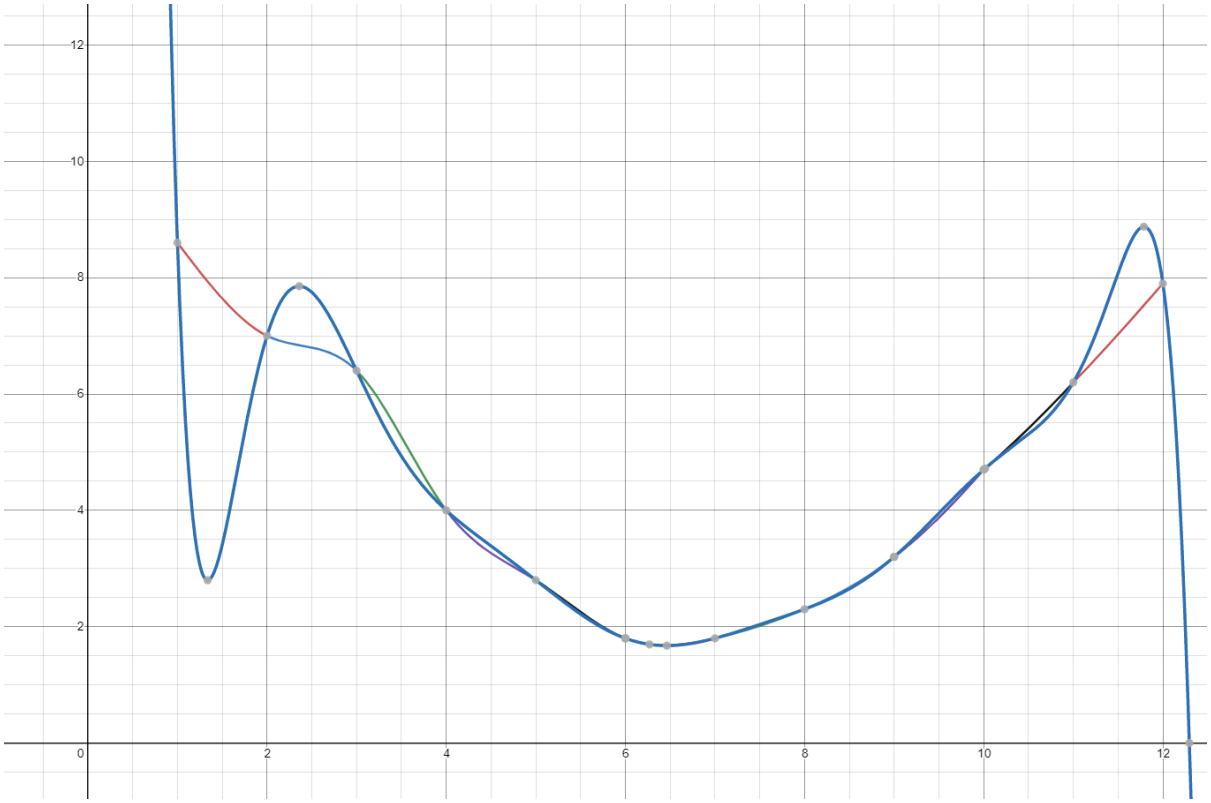
```
S1(x) = 0.000000*((2-x)^3)/6 + 2.522780*((x-1)^3)/6 + 8.600000*(2-x) + 6.579537*(x-1), 1 < x < 2
S2(x) = 2.522780*((3-x)^3)/6 + -4.091120*((x-2)^3)/6 + 6.579537*(3-x) + 7.081853*(x-2), 2 < x < 3
S3(x) = -4.091120*((4-x)^3)/6 + 3.041710*((x-3)^3)/6 + 7.081853*(4-x) + 3.493048*(x-3), 3 < x < 4
S4(x) = 3.041710*((5-x)^3)/6 + -0.875720*((x-4)^3)/6 + 3.493048*(5-x) + 2.945953*(x-4), 4 < x < 5
S5(x) = -0.875720*((6-x)^3)/6 + 1.653618*((x-5)^3)/6 + 2.945953*(6-x) + 1.524397*(x-5), 5 < x < 6
S6(x) = 1.653618*((7-x)^3)/6 + 0.259220*((x-6)^3)/6 + 1.524397*(7-x) + 1.756797*(x-6), 6 < x < 7
S7(x) = 0.259220*((8-x)^3)/6 + 0.309490*((x-7)^3)/6 + 1.756797*(8-x) + 2.248418*(x-7), 7 < x < 8
S8(x) = 0.309490*((9-x)^3)/6 + 0.902814*((x-8)^3)/6 + 2.248418*(9-x) + 3.049531*(x-8), 8 < x < 9
S9(x) = 0.902814*((10-x)^3)/6 + 0.320751*((x-9)^3)/6 + 3.049531*(10-x) + 4.646542*(x-9), 9 < x < 10
S10(x) = 0.320751*((11-x)^3)/6 + 0.380187*((x-10)^3)/6 + 4.646542*(11-x) + 6.136636*(x-10), 10 < x < 11
S11(x) = 0.380187*((12-x)^3)/6 + 0.000000*((x-11)^3)/6 + 6.136636*(12-x) + 7.900000*(x-11), 11 < x < 12
```

Gráfico do spline:

B) Para obter uma comparação melhor entre o polinómio interpolador e o spline cubico natural resolvemos criar um gráfico com uma reta que representa cada um deles sendo a reta azul o polinómio interpolador e a que começa em laranja e vai alternando as cores o spline cubico natural.

Observando então as duas retas, podemos concluir que ambas se intercetam quando o x é 1, 2, …,12 e se tomarmos atenção, é possível concluir que os pontos de interceção destas nesses pontos de x são os pontos do enunciado. Sendo estas duas retas aproximações, era bastante previsível para nós que o resultado fosse algo deste género. Outra observação notável é que desde o início nós sabemos que o spline cubico natural é mais preciso que o polinómio interpolador e podemos também comprová-lo através do gráfico, mais precisamente entre os intervalos de [1,3] e de [11, 12] onde o polinómio “despista-se” muito, já o spline é mais “direto” quando vai de ponto para ponto, fazendo com que a sua aproximação seja mais precisa.

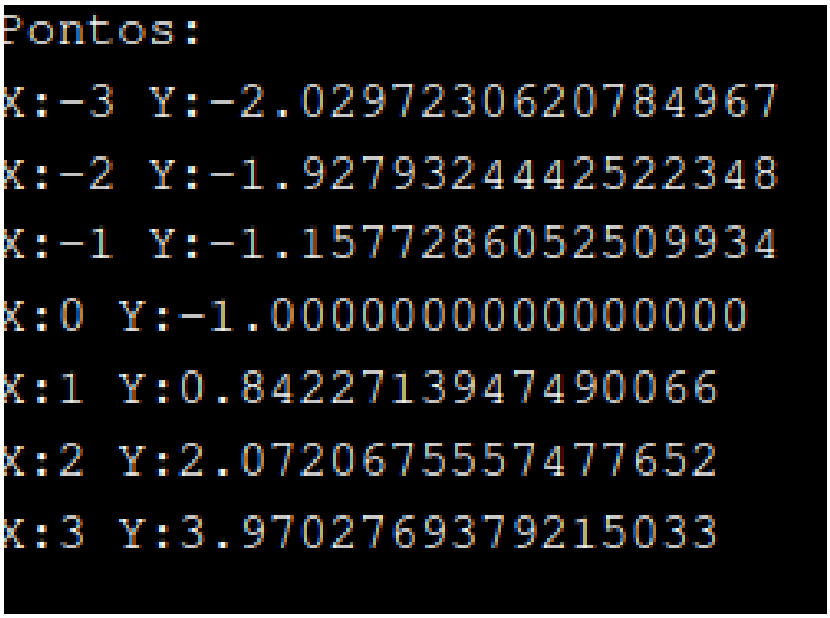
Gráfico de comparação de aproximações:



Exercício 2

Considerando então a função $f(x) = x - \cos(x)^3$, para $-3 \leq x \leq 3$ seguimos com a resolução das seguintes alíneas.

A) Na alínea a, é-nos pedido para criar um conjunto de 7 pontos onde as abcissas. Vão de zero até n igualmente espaçadas no intervalo de menos 3 a 3. Sendo assim, criámos o conjunto:



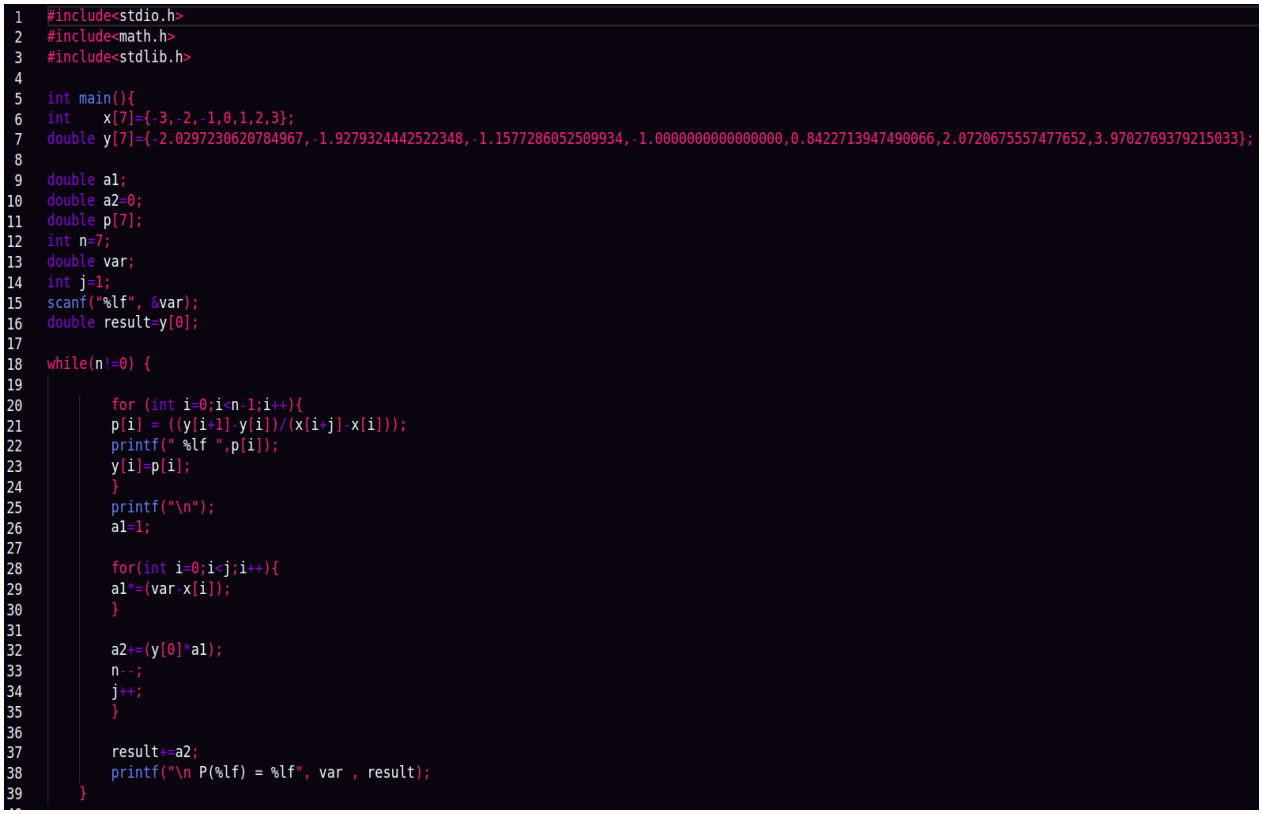
Com o seguinte Código:

```c
#include<stdio.h>
#include<math.h>
#include<stdlib.h>


double fun(int x){
        double cosseno=0;
    double resultado;
    cosseno=cos(x);
    resultado=pow(cosseno,3);
    return x-resultado;
}

void imprimir(int i){
    printf("X:%d ",i);
    printf("Y:%.16lf\n",fun(i));
    }


int main(){
    printf("Pontos: \n");
    for(int i=-3;i<=0;i++){
    imprimir(i);
    if(i==0){
    for(int i=1;i<=3;i++){
    imprimir(i);}}}

}
```

O código calcula todo o f(x) para cada x dado. Como x=[-3,3], feito o calcuo 7 vezes.

B) A partir daqui calculamos o polinomio interpolador com dois metodos diferentes: newton e lagranje. Ambos os resultados foram iguais graficamente, portanto temos certeza da expressao.

Pelo método de Newton o código implementado foi o seguinte:



```c
#include<stdio.h>
#include<math.h>
#include<stdlib.h>

int main(){
int    x[7]={-3,-2,-1,0,1,2,3};
double y[7]={-2.0297230620784967,-1.9279324442522348,-1.1577286052509934,-1.0000000000000000,0.8422713947490066,2.0720675557477652,3.9702769379215033};

double a1;
double a2=0;
double p[7];
int n=7;
double var;
int j=1;
scanf("%lf", &var);
double result=y[0];

while(n!=0) {

      for (int i=0;i<n-1;i++){
      p[i] = ((y[i+1]-y[i])/(x[i+j]-x[i]));
      printf(" %lf ",p[i]);
      y[i]=p[i];
      }
      printf("\n");
      a1=1;

      for(int i=0;i<j;i++){
      a1*=(var-x[i]);
      }

      a2+=(y[0]*a1);
      n--;
      j++;
      }

      result+=a2;
      printf("\n P(%lf) = %lf", var , result);
   }
```

Dados x0,x1,...,xn, n+1 valores distintos podemos calcular as sucessivas diferenças divididas de uma função(x) naquelas abcissas.

Define-se diferença dividida de ordem **n**

No algoritmo é determinado através:

While(n!=0){

for (int i=0;i<n-1;i++){

p[i] = ((y[i+1]-y[i])/(x[i+j]-x[i]));C

printf(" %lf ",p[i]);

y[i]=p[i];

}

…...

}

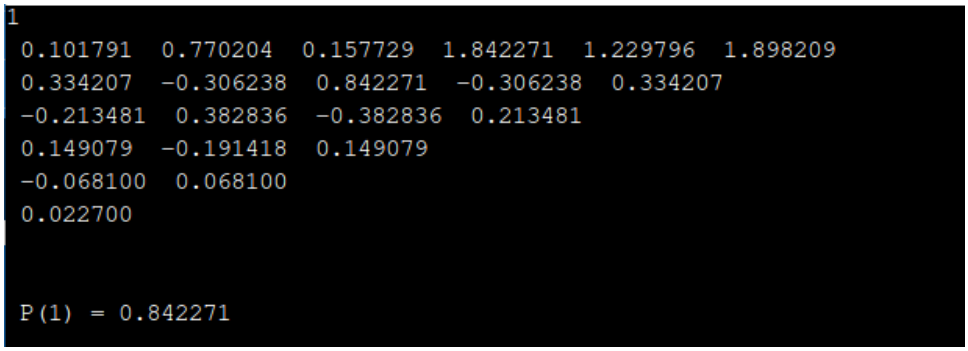
Depois precisamos de determinar a expressão

```
for(int i=0;i<j;i++){

a1*=(var-x[i]);

}

a2+=(y[0]*a1);

n--;

j++;

}

……
```

Nota: var corresponde ao valor pedido pelo scanf

Por fim o valor result=y[0] é adicionado ao a2 e obtemos Pn(x):

Executando o Programa vamos ter que inserir um valor X para saber o valor do polinómio interpolador P(X). Vai retornar os valores de todas as diferenças divididas calculadas e o valor do P(x). No exemplo da imagem o valor inserido foi 1.



A segunda implementação é semelhante à do primeiro exercicio. Pelo método de Lagranje o programa é o seguinte:

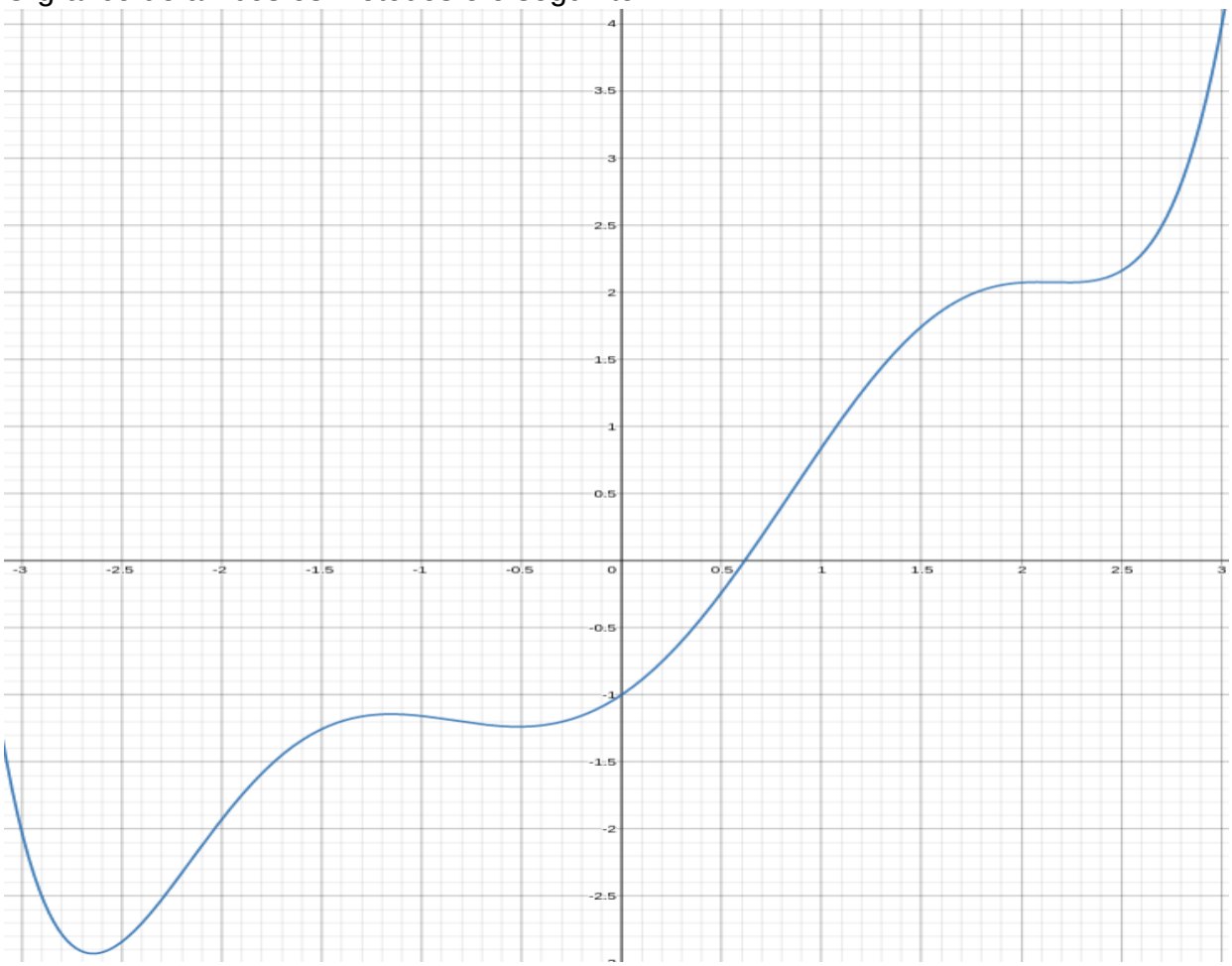
```c
#include<stdio.h>

int main(){

  float evap[7] ={-2.0297230620784967,
                  -1.9279324442522348,
                  -1.1577286052509934,
                  -1.0000000000000000,
                  0.8422713947490066,
                  2.0720675557477652,
                  3.9702769379215033};

    printf("polinomio:\n");


    printf("((x+%d)",2);
    printf("*(x+%d)",1);

        for(int j=0;j<=3;j++){
            printf("*(x-%d)",j);
        }

        printf(")/");
        printf("(");

        printf("(-3+2)");
        printf("*(-3+1)");

        for(int j=0;j<=3;j++){
            printf("*(-3-%d)",j);
        }

        printf(")");
        printf("*(%f)+", evap[0]);
```

```c
    for(int i=-2;i<=3;i++){

        printf("(");
        printf("(x+%d)",3);
        if(i!=-2) printf("*(x+%d)",2);
        if(i!=-1) printf("*(x+%d)",1);


        for(int j=0;j<=3;j++){
            if(i!=j) printf("*(x-%d)",j);
        }

        printf(")/");

        printf("(");
        printf("(%d+%d)",i,3);
         if(i!=-2) printf("*(%d+%d)",i,2);
         if(i!=-1) printf("*(%d+%d)",i,1);
        for(int j=0;j<=3;j++){
            if(i!=j) printf("*(%d-%d)",i,j);
        }

        printf(")");


       if(i!=3)  printf("*(%f)+", evap[i+3]);


       else printf("*(%f)", evap[i+3]);
    }


    return 0;
}
```

O Output deste programa é o seguinte:

```
polinomio:
((x+2)*(x+1)*(x-0)*(x-1)*(x-2)*(x-3))/((-3+2)*(-3+1)*(-3-0)*(-3-1)*(-3-2)*(-3-3))*(
-2.029723)+((x+3)*(x+1)*(x-0)*(x-1)*(x-2)*(x-3))/((-2+3)*(-2+1)*(-2-0)*(-2-1)*(-2-2
)*(-2-3))*(-1.927933)+((x+3)*(x+2)*(x-0)*(x-1)*(x-2)*(x-3))/((-1+3)*(-1+2)*(-1-0)*(
-1-1)*(-1-2)*(-1-3))*(-1.157729)+((x+3)*(x+2)*(x+1)*(x-1)*(x-2)*(x-3))/((0+3)*(0+2)
*(0+1)*(0-1)*(0-2)*(0-3))*(-1.000000)+((x+3)*(x+2)*(x+1)*(x-0)*(x-2)*(x-3))/((1+3)*
(1+2)*(1+1)*(1-0)*(1-2)*(1-3))*(0.842271)+((x+3)*(x+2)*(x+1)*(x-0)*(x-1)*(x-3))/((2
+3)*(2+2)*(2+1)*(2-0)*(2-1)*(2-3))*(2.072067)+((x+3)*(x+2)*(x+1)*(x-0)*(x-1)*(x-2))
/((3+3)*(3+2)*(3+1)*(3-0)*(3-1)*(3-2))*(3.970277)manel@derpro:~/Desktop/anexo/A$ cd
```

O gráfico de ambos os métodos é o seguinte:



O próximo programa corresponde ao spline cúbico natural s :

O código foi adaptado pelo pseudocódigo do livro (pag 149) **Numerical Analysis 9th** Richard L. Burden and J. Douglas Faires

Natural Cubic Spline

To construct the cubic spline interpolant S for the function f, defined at the numbers $x_0 < x_1 < \cdots < x_n$, satisfying $S''(x_0) = S''(x_n) = 0$:

INPUT $n; x_0, x_1, \ldots, x_n; a_0 = f(x_0), a_1 = f(x_1), \ldots, a_n = f(x_n)$.

OUTPUT a_j, b_j, c_j, d_j for $j = 0, 1, \ldots, n-1$.

(*Note:* $S(x) = S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$ *for* $x_j \le x \le x_{j+1}$.)

Step 1 For $i = 0, 1, \ldots, n-1$ set $h_i = x_{i+1} - x_i$.

Step 2 For $i = 1, 2, \ldots, n-1$ set

$$\alpha_i = \frac{3}{h_i}(a_{i+1} - a_i) - \frac{3}{h_{i-1}}(a_i - a_{i-1}).$$

Step 3 Set $l_0 = 1$; (*Steps 3, 4, 5, and part of Step 6 solve a tridiagonal linear system using a method described in Algorithm 6.7.*)
$\mu_0 = 0$;
$z_0 = 0$.

Step 4 For $i = 1, 2, \ldots, n-1$
set $l_i = 2(x_{i+1} - x_{i-1}) - h_{i-1}\mu_{i-1}$;
$\mu_i = h_i / l_i$;
$z_i = (\alpha_i - h_{i-1}z_{i-1})/l_i$.

Step 5 Set $l_n = 1$;
$z_n = 0$;
$c_n = 0$.

Step 6 For $j = n-1, n-2, \ldots, 0$
set $c_j = z_j - \mu_j c_{j+1}$;
$b_j = (a_{j+1} - a_j)/h_j - h_j(c_{j+1} + 2c_j)/3$;
$d_j = (c_{j+1} - c_j)/(3h_j)$.

Step 7 OUTPUT $(a_j, b_j, c_j, d_j$ for $j = 0, 1, \ldots, n-1)$;
STOP. ■

O que origina o seguinte código:

```c
#include <stdio.h>

int main() {

    int n=6;
    float  h[n], A[n], l[n + 1],u[n + 1], z[n + 1],c[n + 1],b[n], d[n];

   float x[7]={-3,-2,-1,0,1,2,3};
   float
y[7]={-2.0297230620784967,-1.9279324442522348,-1.1577286052509934,-1.0000000000000000,0.8422713947490066,

    // Passo 1
    for (int i = 0; i <= n - 1; i++)
    h[i] = x[i + 1] - x[i];

    // Passo 2
    for (int i = 1; i <= n - 1; i++)
    A[i] = 3 * (y[i + 1] - y[i]) / h[i] - 3 * (y[i] - y[i - 1]) / h[i - 1];

    // Passo 3
    l[0] = 1;
    u[0] = 0;
    z[0] = 0;

    // Passo 4
    for (int 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] = (A[i] - h[i - 1] * z[i - 1]) / l[i];
    }


    // Passo 5
    l[n] = 1;
    z[n] = 0;
    c[n] = 0;

    // Passo 6
    for (int j = n - 1; j >= 0; j--) {
        c[j] = z[j] - u[j] * c[j + 1];
        b[j] = (y[j + 1] - y[j]) / h[j] - h[j] * (c[j + 1] + 2 * c[j]) / 3;
        d[j] = (c[j + 1] - c[j]) / (3 * h[j]);
    }

     for (int i = 0; i < n; i++)
     printf("%2d %8.2f %8.2f %8.2f %8.2f\n", i, y[i], b[i], c[i], d[i]);

     printf("\n");

    for (int i = 0; i < n; i++)
    printf("S(%d) = %.2f + %.2f*(x - %.2f) + %.2f*(x - %.2f)^2 + %.2f*(x - %.2f)^3 \n", i, y[i],b[i],x[i], c[i],x[i], d[i],x[i]);
    return 0;
}
```

Executando o programa, vamos ter no output do terminal as seguintes retas:

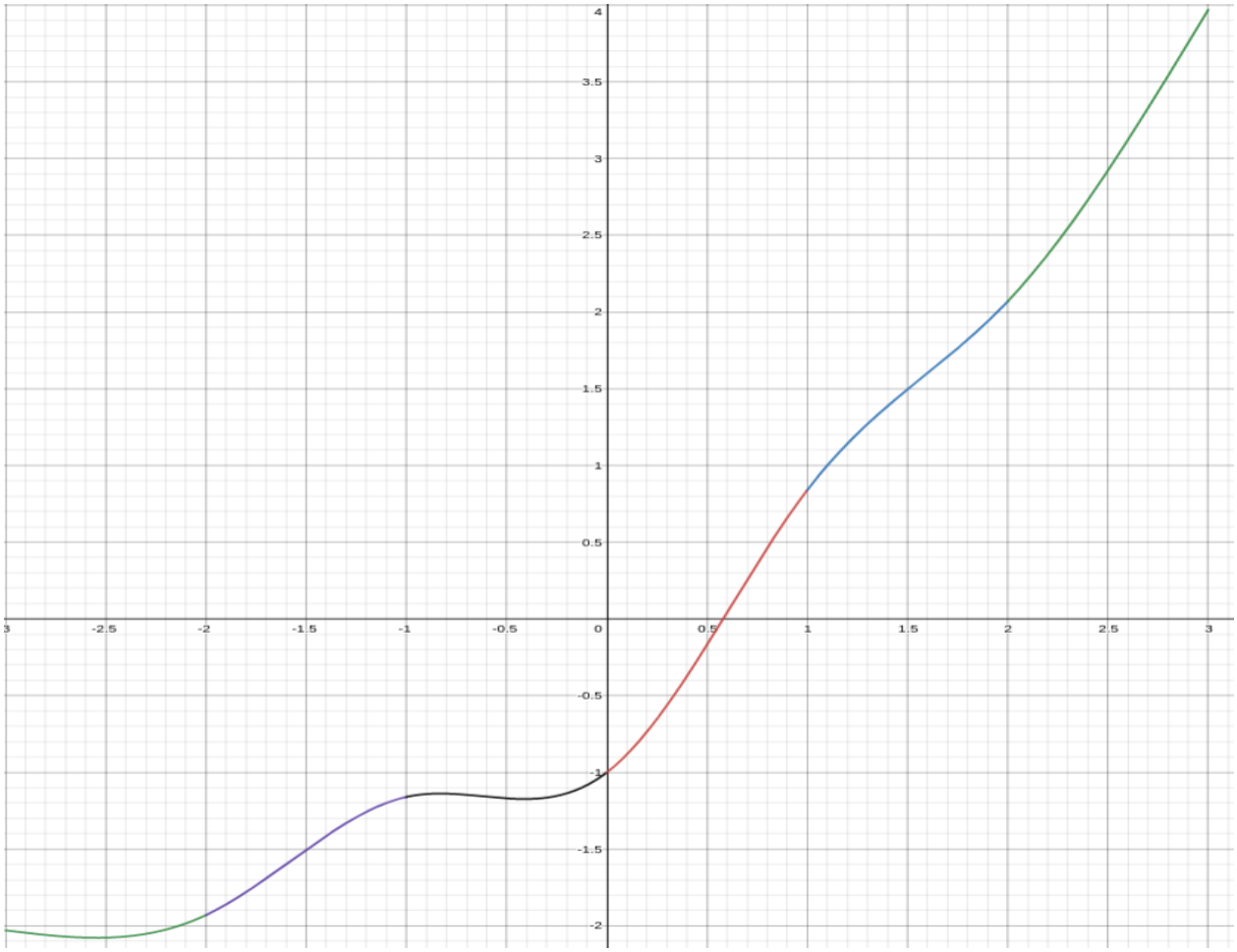
```
0    -2.03    -0.16     0.00     0.26
1    -1.93     0.62     0.78    -0.63
2    -1.16     0.29    -1.11     0.98
3    -1.00     1.00     1.82    -0.98
4     0.84     1.71    -1.11     0.63
5     2.07     1.38     0.78    -0.26

S(0) = -2.03 + -0.16*(x - -3.00) + 0.00*(x - -3.00)^2 + 0.26*(x - -3.00)^3
S(1) = -1.93 + 0.62*(x - -2.00) + 0.78*(x - -2.00)^2 + -0.63*(x - -2.00)^3
S(2) = -1.16 + 0.29*(x - -1.00) + -1.11*(x - -1.00)^2 + 0.98*(x - -1.00)^3
S(3) = -1.00 + 1.00*(x - 0.00) + 1.82*(x - 0.00)^2 + -0.98*(x - 0.00)^3
S(4) = 0.84 + 1.71*(x - 1.00) + -1.11*(x - 1.00)^2 + 0.63*(x - 1.00)^3
S(5) = 2.07 + 1.38*(x - 2.00) + 0.78*(x - 2.00)^2 + -0.26*(x - 2.00)^3
```

Nota :A ordem das retas é do Ponto de abcissa x=-1 até ao Ponto x=3.

É de notar tambem que em cada S, x está restrito dentro de intervalos. Sendo h=1,

S(0)=>-3<x<-2,S(1)=>-2<x<-1,(…),S(5)=>2<x<3

O gráfico do spline cúbico natural é o seguinte:



C) Tendo já calculado e representado graficamente as funções acima mencionadas, estes são os seguintes gráficos que comparam os 2 métodos diferentes.

O gráfico a roxo representa o cálculo do polinômio interpolado acima. A outra função a várias cores, simboliza o spline dividido em intervalos.

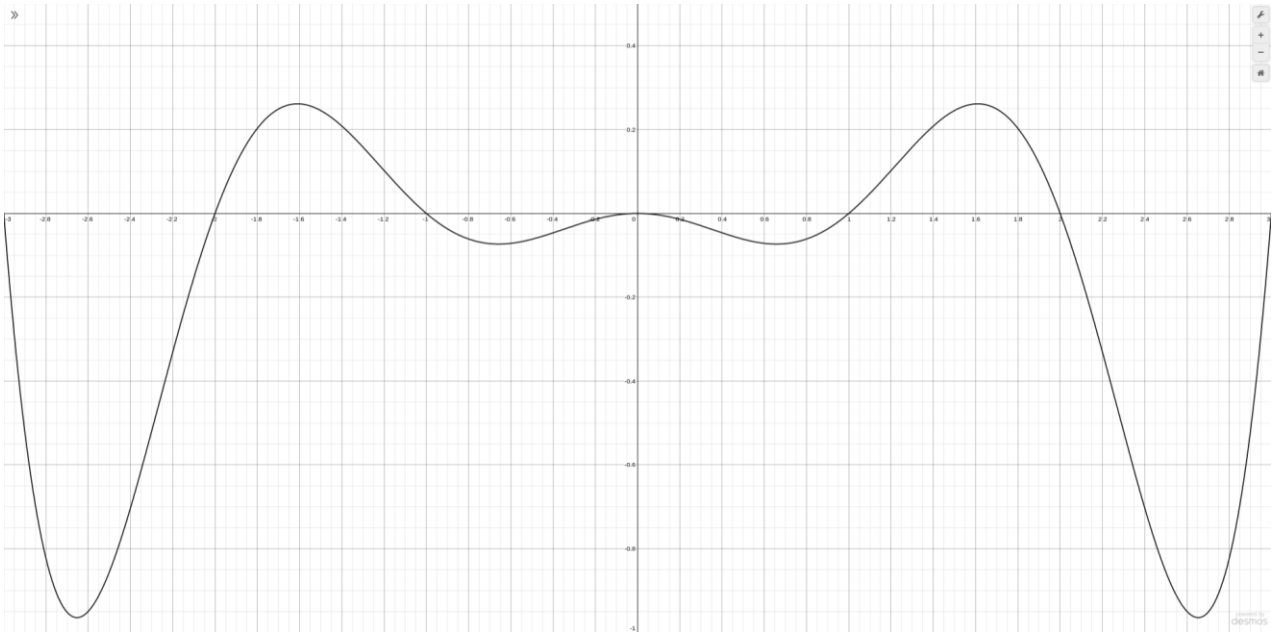
É de notar que ambas as aproximações são bastante positivas e muitos semelhantes uma à outra. No entanto é necessária uma resposta exata e para isso precisamos de comparar as funções de erro de cada uma ou seja:

|p-f| sendo função de erro do polinómio interpolador;

|p-s| sendo a função de erro do spline.

De seguida estão representadas as funções de erro de cada uma.

Função de erro polinómio interpolador



Função de erro do spline:

Como é possível observar, tendo em conta que ambas as imagens partilham os mesmos intervalos, é fácil concluir que a função de spline é superior à do Polinômio Interpolador. Os valores das ordenadas são muito menores, portanto, aproxima-se mais do valor real. Logo concluímos com certeza que o spline é superior a nível de precisão em comparação ao Polinômio Interpolador.

Alínea d)

Comecemos por majorar o erro usando o polinómio interpolador. Para tal, teremos que nos sustentar no seguinte teorema:

Teorema

Dada uma função $f(x) \in C^{n+1}[a,b]$, seja $p_n(x) \in \mathbb{P}_n$ o polinómio interpolador de $f(x)$ nos pontos de abcissas distintas $x_0, x_1, \ldots, x_n \in [a,b]$ então
$\forall x \in [a,b] \ \exists c_x \in]a,b[$:

$$f(x) - p_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(c_x)\pi_{n+1}(x) \qquad (1)$$

onde $\pi_{n+1}(x) = (x-x_0)(x-x_1)\ldots(x-x_n)$

Se observarmos com atenção o teorema, verificamos que necessitamos do valor exato de f(0.1), f(2.6), p(0.1) e p(2.6).

f(0.1) = $0.1 - \cos^3(0.1) = -0.88508\ldots$

f(2.6) = $2.6 - \cos^3(2.6) = 3.22917\ldots$

Porém, para obtermos o valor exato de p(0.1) e p(2.6) teremos que executar o programa implementado que calcula o polinómio interpolador segundo o método de Lagrange num determinado ponto.

```c
#include<stdio.h>

int main(){

  float evap[7] ={-2.0297230620784967,
                  -1.9279324442522348,
                  -1.1577286052509934,
                  -1.0000000000000000,
                  0.8422713947490066,
                  2.0720675557477652,
                  3.9702769379215033};

    printf("polinomio:\n");


    printf("((x+%d)",2);
    printf("*(x+%d)",1);

        for(int j=0;j<=3;j++){
            printf("*(x-%d)",j);
        }

        printf(")/");
        printf("(");

        printf("(-3+2)");
        printf("*(-3+1)");

        for(int j=0;j<=3;j++){
            printf("*(-3-%d)",j);
        }

        printf(")");
        printf("*(%f)+", evap[0]);
```

```c
    for(int i=-2;i<=3;i++){

        printf("(");
        printf("(x+%d)",3);
        if(i!=-2) printf("*(x+%d)",2);
        if(i!=-1) printf("*(x+%d)",1);


        for(int j=0;j<=3;j++){
            if(i!=j) printf("*(x-%d)",j);
        }

        printf(")/");

        printf("(");
        printf("(%d+%d)",i,3);
         if(i!=-2) printf("*(%d+%d)",i,2);
         if(i!=-1) printf("*(%d+%d)",i,1);
        for(int j=0;j<=3;j++){
            if(i!=j) printf("*(%d-%d)",i,j);
        }

        printf(")");


       if(i!=3)  printf("*(%f)+", evap[i+3]);


       else printf("*(%f)", evap[i+3]);
    }


    return 0;
}
```

Assim, obtemos: p(0.1) = -0.888785585

p(2.6) = 2.2798918

Comecemos então o cálculo com base no teorema referido inicialmente.

$$|f(x) - p_6(x)| <= \frac{M}{7!}|(x-x0)(x-x1)(x-x2)(x-x3)(x-x4)(x-x5)(x-x6)|$$

M = *max*|$f_7(x)$|no intervalo [-3,3], sendo que $f_7(x) = 3\,(\,182.\sin(x) - 729.\cos^2(x).\sin(x))$

(f_7 corresponde à sétima derivada de f)

Com base numa calculadora online verificamos que o máximo no respetivo intervalo é 547.125, correspondendo ao M.

Para 0.1:

$$|-0.88508 - (-0.888786)| <= \frac{547.125}{7!}|(0.1+3))(0.1+2))(0.1+1)(0.1-0)(0.1-1)(0.1-2)(0.1-3)| <= 3.3 * 101$$

Sendo o resultado: $0.0037 \pm 3.3 * 10^1$

Para 2.6:

$$|3.22917 - 3.279877| <= \frac{547.125}{7!}|(2.6-(-3))(2.6-(-2))(2.6-(-1))(2.6-0)(2.6-1)(2.6-2)(2.6-3)| <= 8.5 * 10^2$$

Sendo o resultado: $-0.051 \pm 8.5 * 10^2$

Já para majorar o erro usando o spline cúbico, sustentamo-nos noutro teorema:

- Se $f(x) \in C^4[a,b]$ então mostra-se que $\forall x \in [a,b]$

$$|f(x) - S(x)| \leq \frac{5}{384} M h^4$$

onde $M = \max_{x\in[a,b]} |f^{(4)}(x)|$ e $h = \max h_i,\ i = 1,2,\ldots,n.$

Sabendo que M = *max*|$f_4(x)$| e que $f_4(x) = 3 (20.\cos(x) - 27.\cos^3(x))$, com base novamente na calculadora online, verificamos que o máximo é 21, ou seja, M = 21.

Como o h corresponde ao intervalo entre as abcissas, e estas por sua vez estão igualmente espaçadas, então os intervalos são todos iguais e portanto h=1.

De seguida, segundo o sistema calculado na alínea b), obtemos os valores:

s(0.1) = −0.88278

s(2.6) = 3.12264

Para 0.1:
$|-\ 0.88508\ -\ \ (-0.88278)| <\ \ = \frac{5}{384} \cdot 21 \cdot 1^4 <\ \ = 2.7 \cdot 10^{-1}$
sendo o resultado: $-0.0023 \pm 2.7 * 10^{-1}$

Para 2.6: $|3.22917 - 3.12264| <= \frac{5}{384} * 21 * 1^4 <= 2.7 * 10^{-1}$

Sendo o resultado: $0.10653 \pm 2.7 * 10^{-1}$

E) Num modo geral, a análise dos resultados obtidos acima, permite-nos concluir que o método de spline apresenta-nos melhores resultados em todos os casos. A principal razão para isto acontecer tem a ver com o facto do método de spline dividir em intervalos mais pequenos o domínio e aplicar o método do polinómio interpolador lá. Isto faz com que a possibilidade de erro seja menor, viste que o intervalo de abissas também o é, tornando o resultado mais preciso.

Conclusão

Para concluir, ao longo do trabalho tivemos alguns problemas com a impressão dos gráficos devido a algumas calculadoras online apenas conseguirem usar um limite de 10 funções e, devido ao spline, precisamos de mais do que isso. As impressões de funções em si que não sejam gráficos foram feitas em código e imprimidas no terminal como se pode ver ao longo do trabalho.

A resolução dos exercícios em si não foi difícil e não enfrentamos grandes dificuldades nesse aspeto, onde surgiram alguns problemas era a maneira como os iriamos resolver, o que faríamos em código e o que faríamos de outras maneiras.