INF1608 - Análise Numérica
Cubic Spline

André Mazal - 1410386
Marcelo Paulon - 1411029
Renan da Fonte - 1412122

Solução

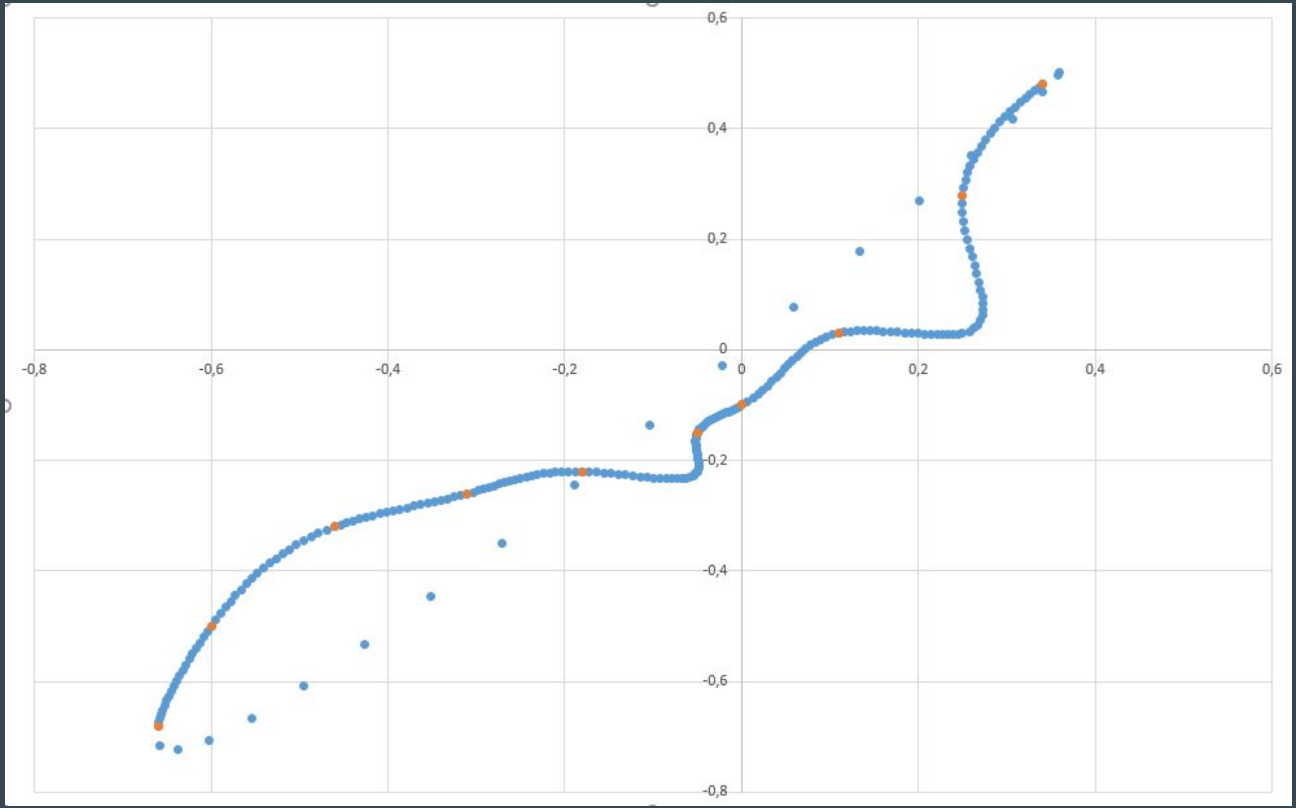
Implementação em C++

Uso dos laboratórios de Matriz e Fatoração LU

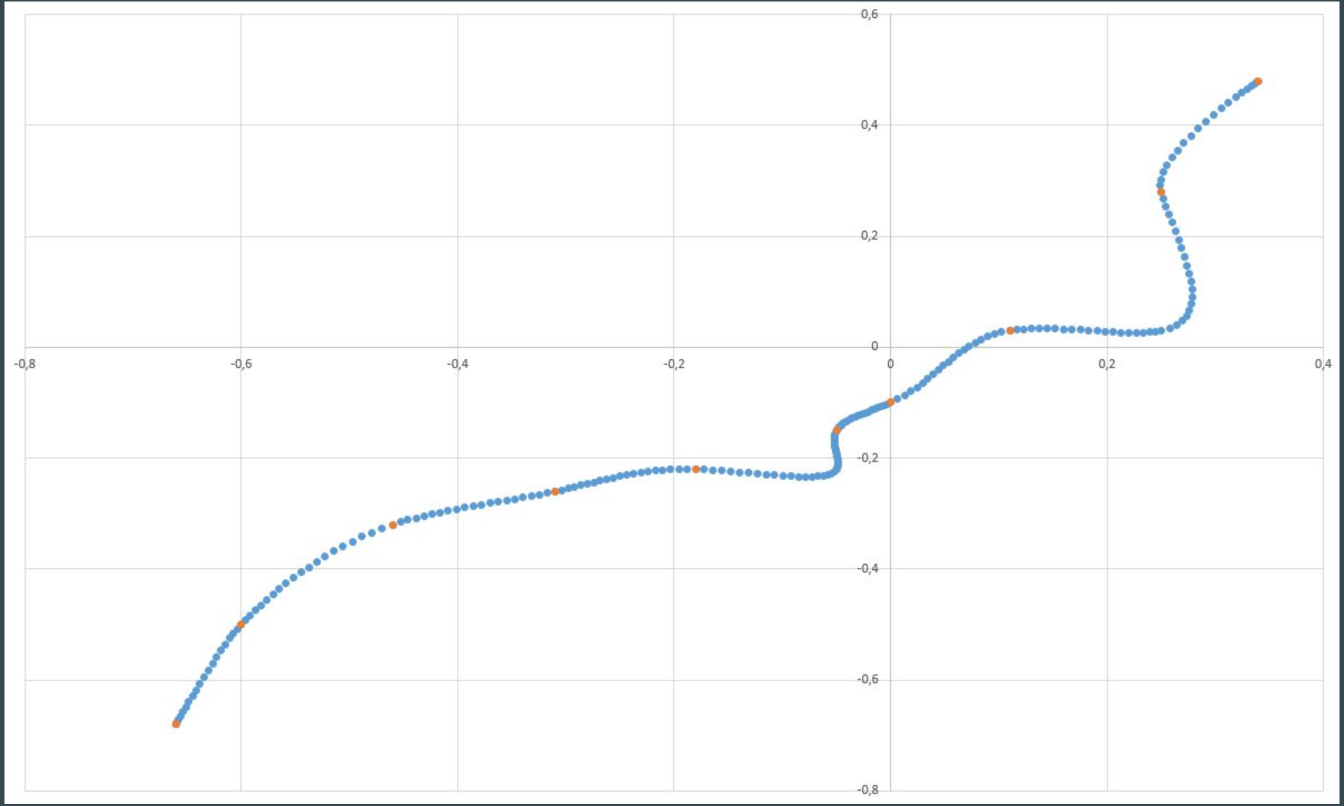
Método de Thomas para resolução de sistemas tridiagonais (O(n))

Plot - Microsoft Excel

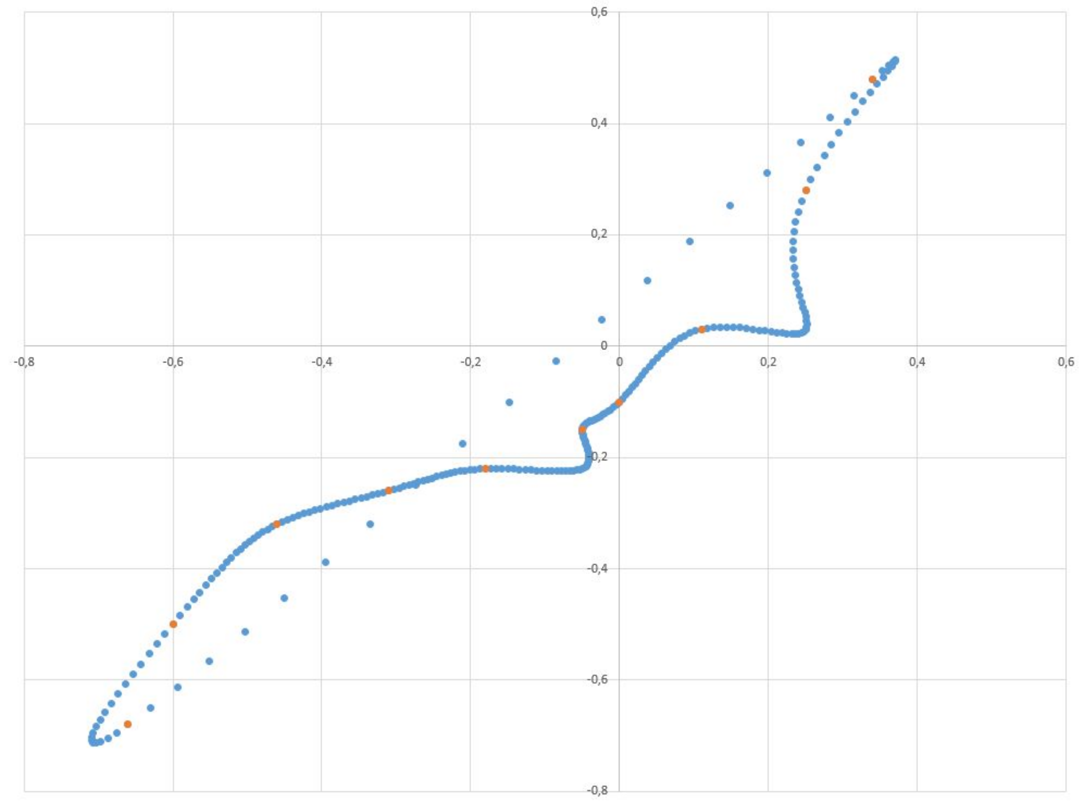
Polinômio - 10 pontos - fechado e não-uniforme



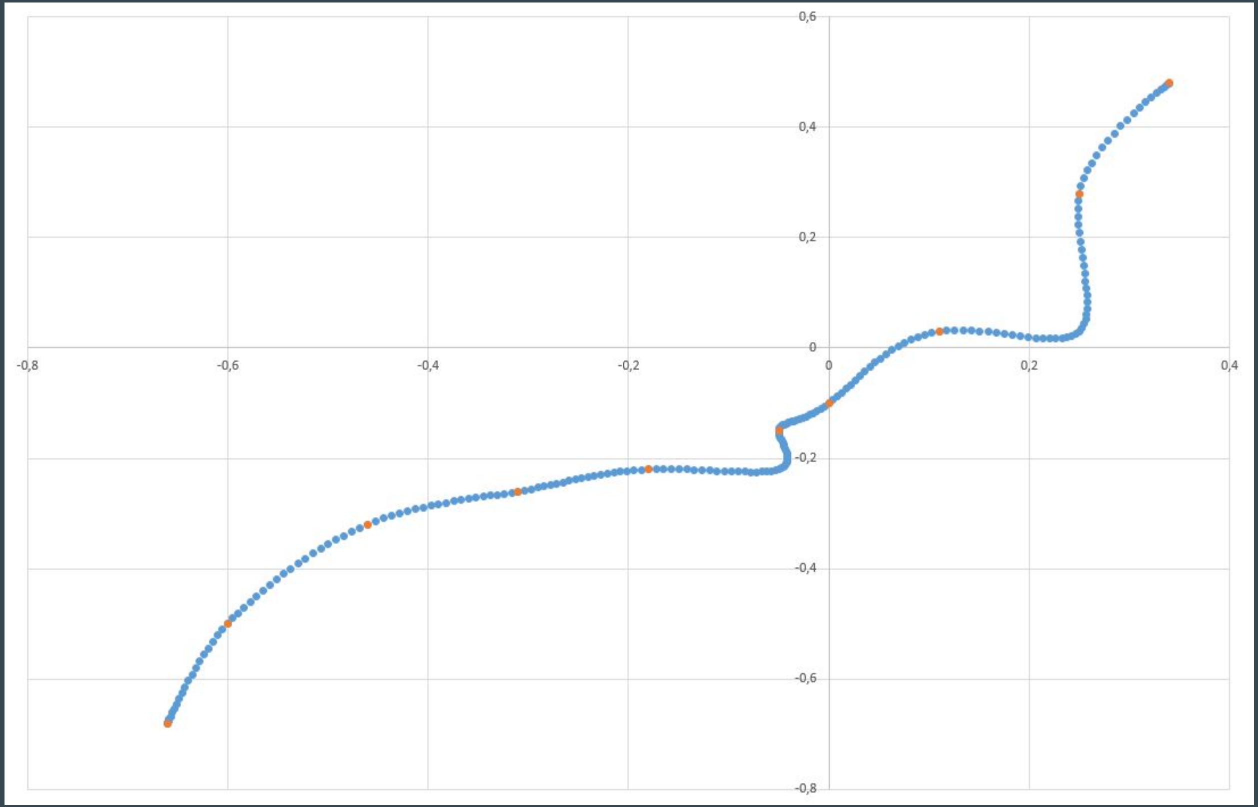
Polinômio - 10 pontos - aberto e não-uniforme



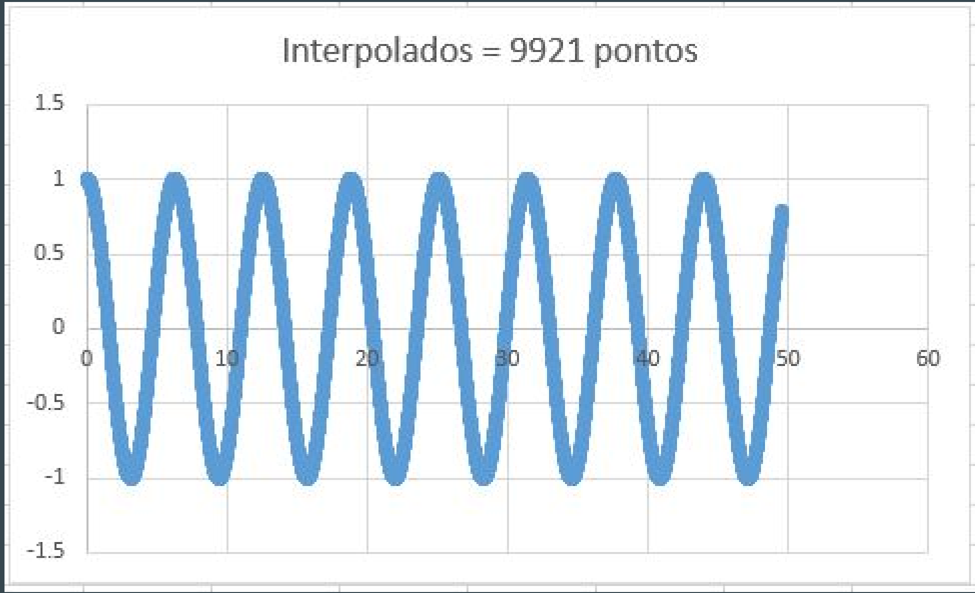
Polinômio - 10 pontos - fechado e uniforme

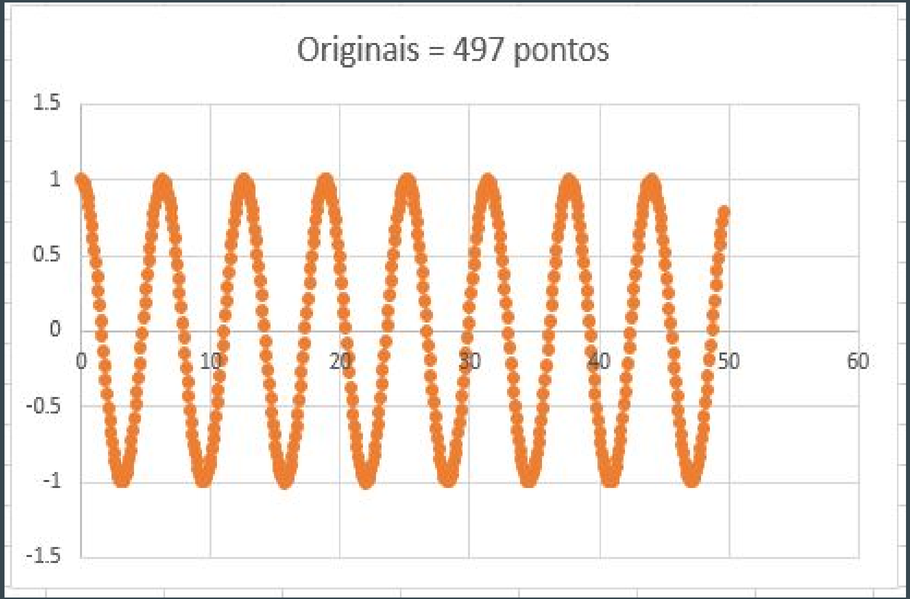


Polinômio - 10 pontos - aberto e uniforme



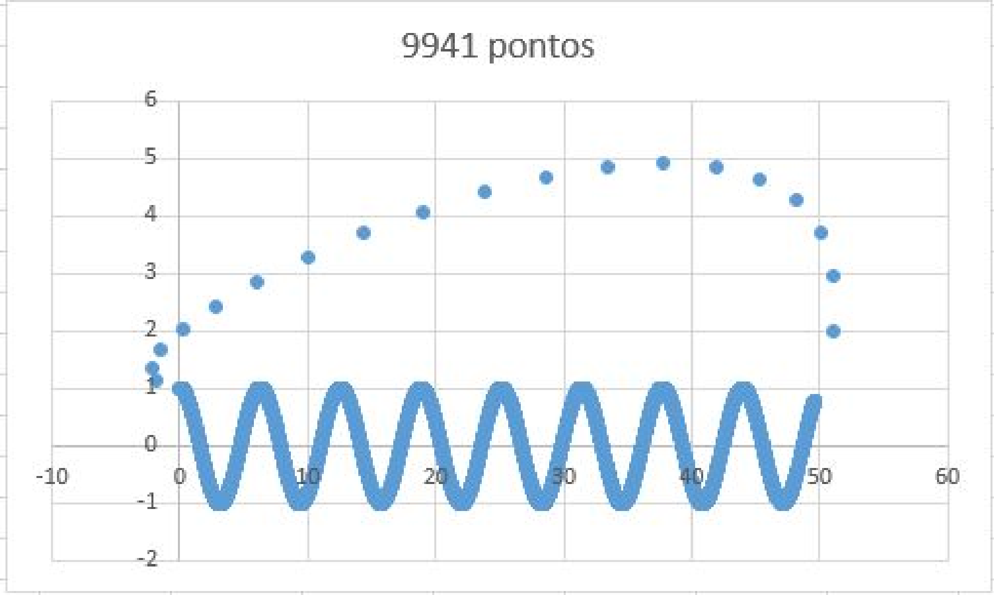
Cos(x): interpolação aberta e não uniforme

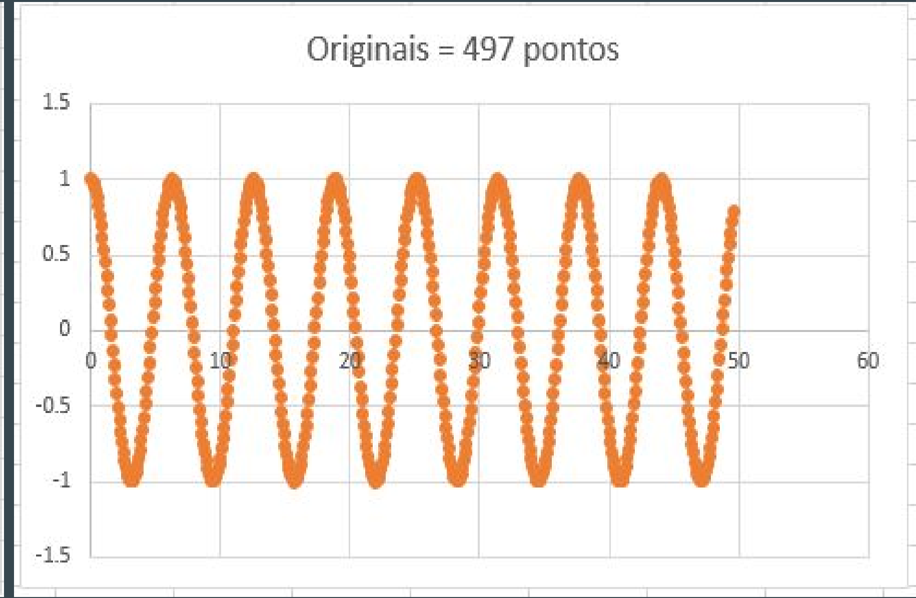




0.003 segundos

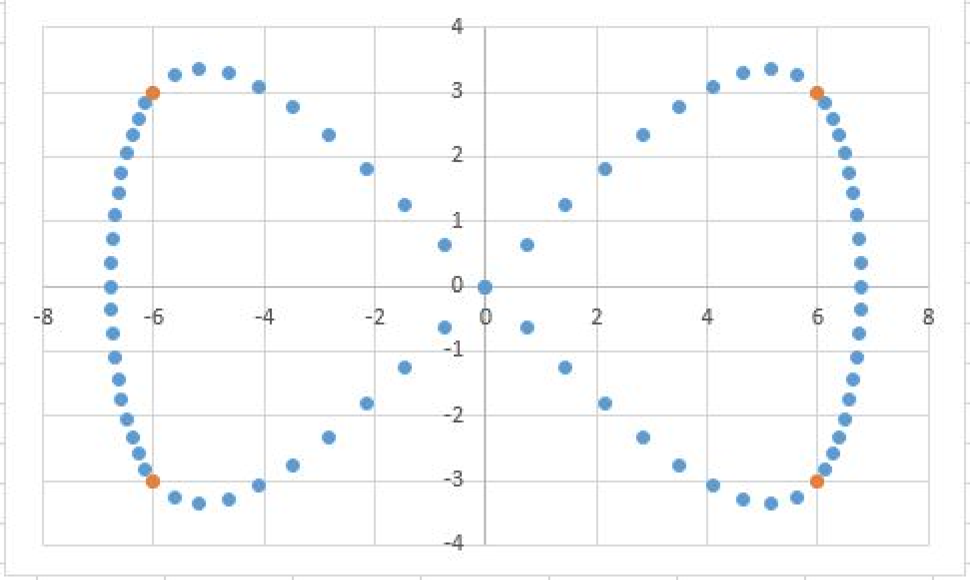
Cos(x): fechado e não uniforme

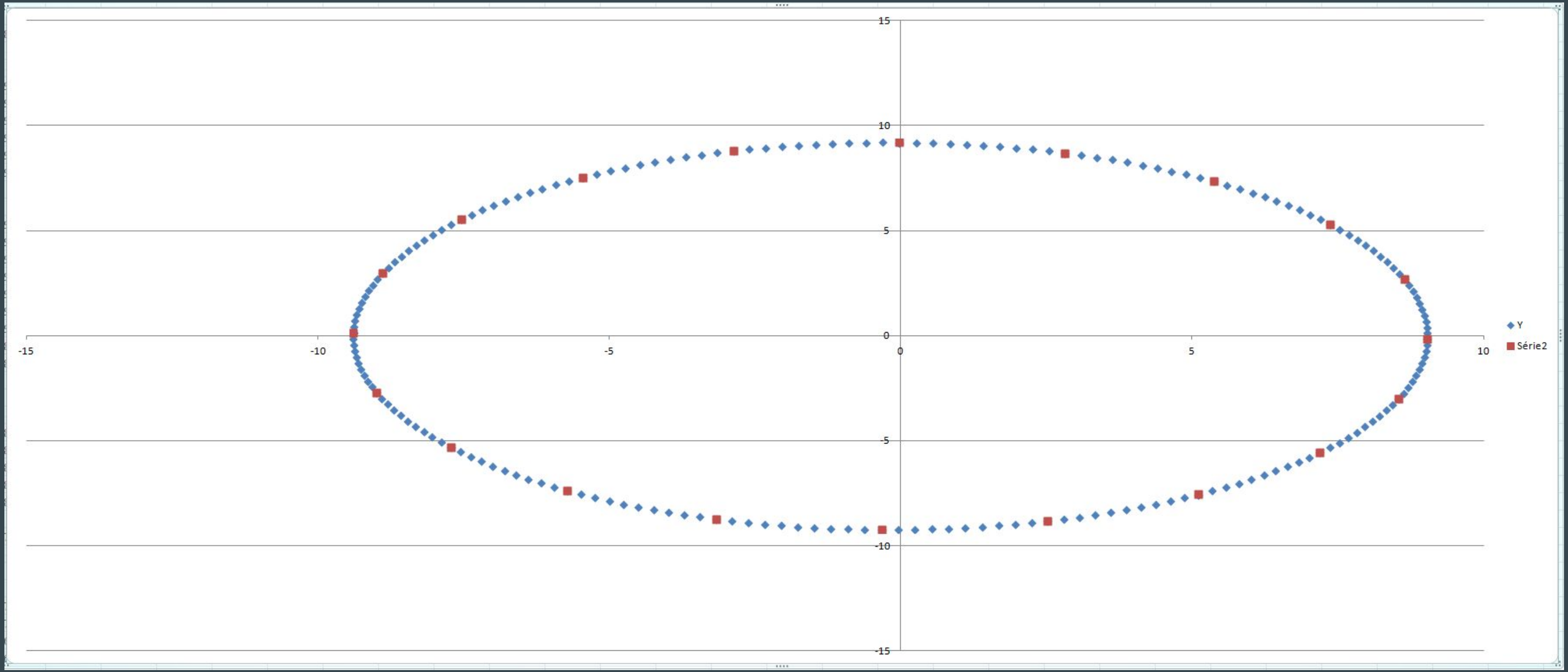




0.197 segundos

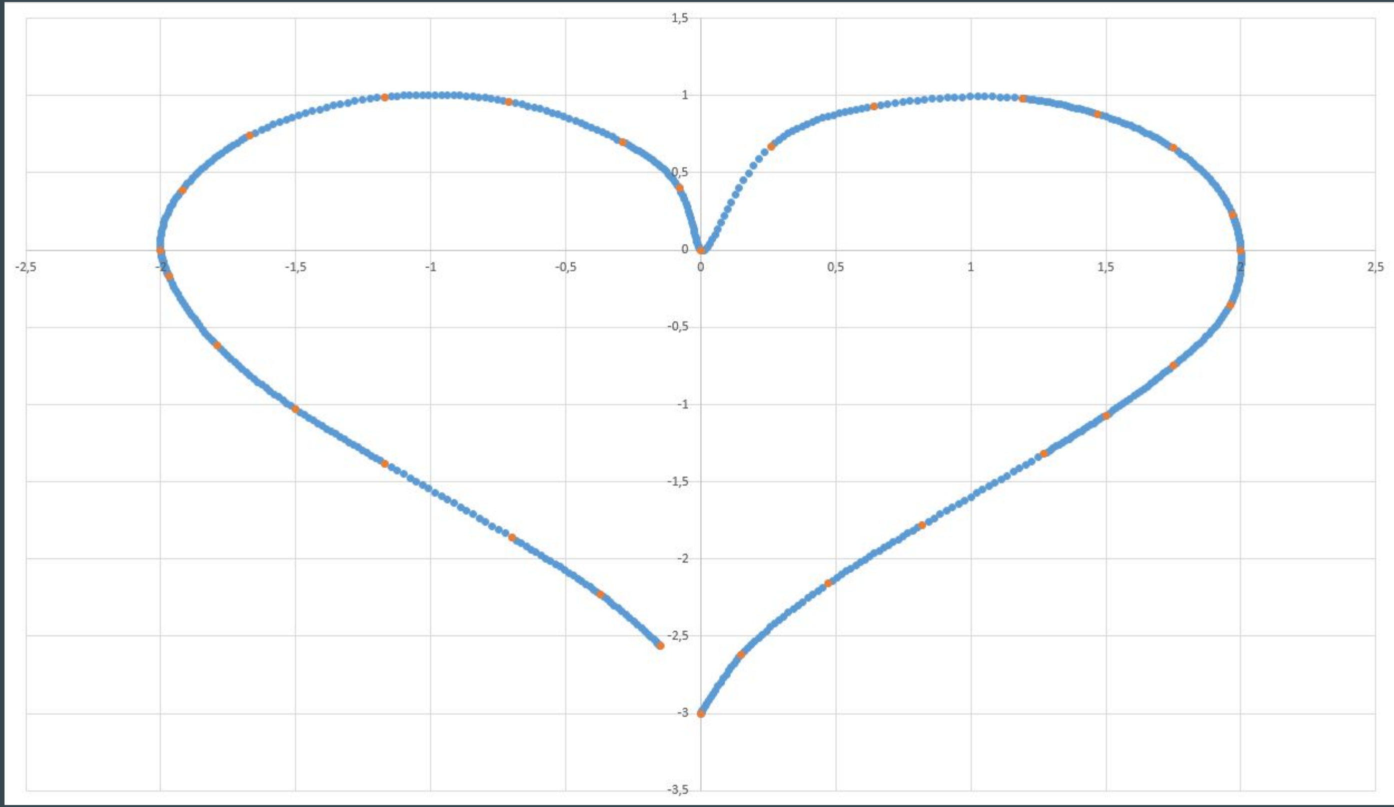
Testes



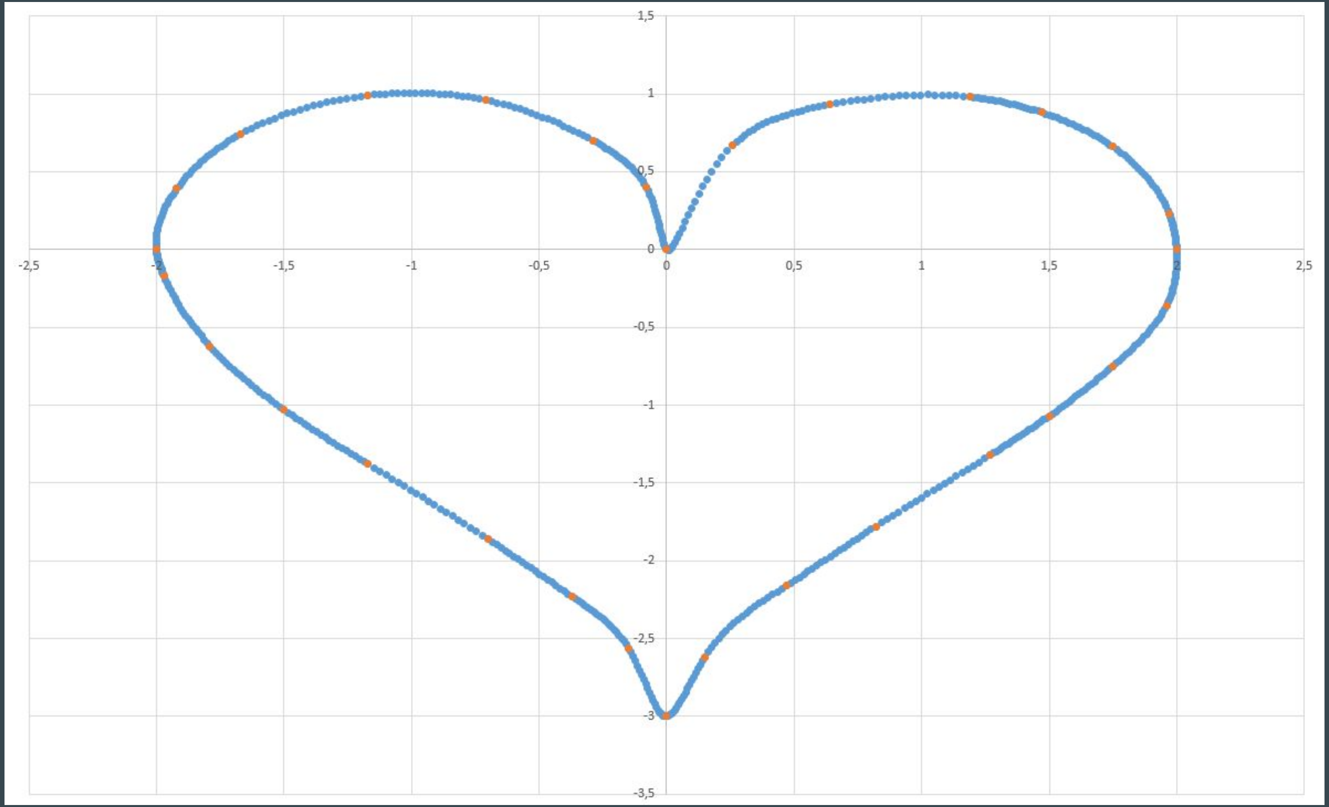


15
10
5
0
-5
-10
-15
-15
-10
-5
0
5
10
Y
Série2

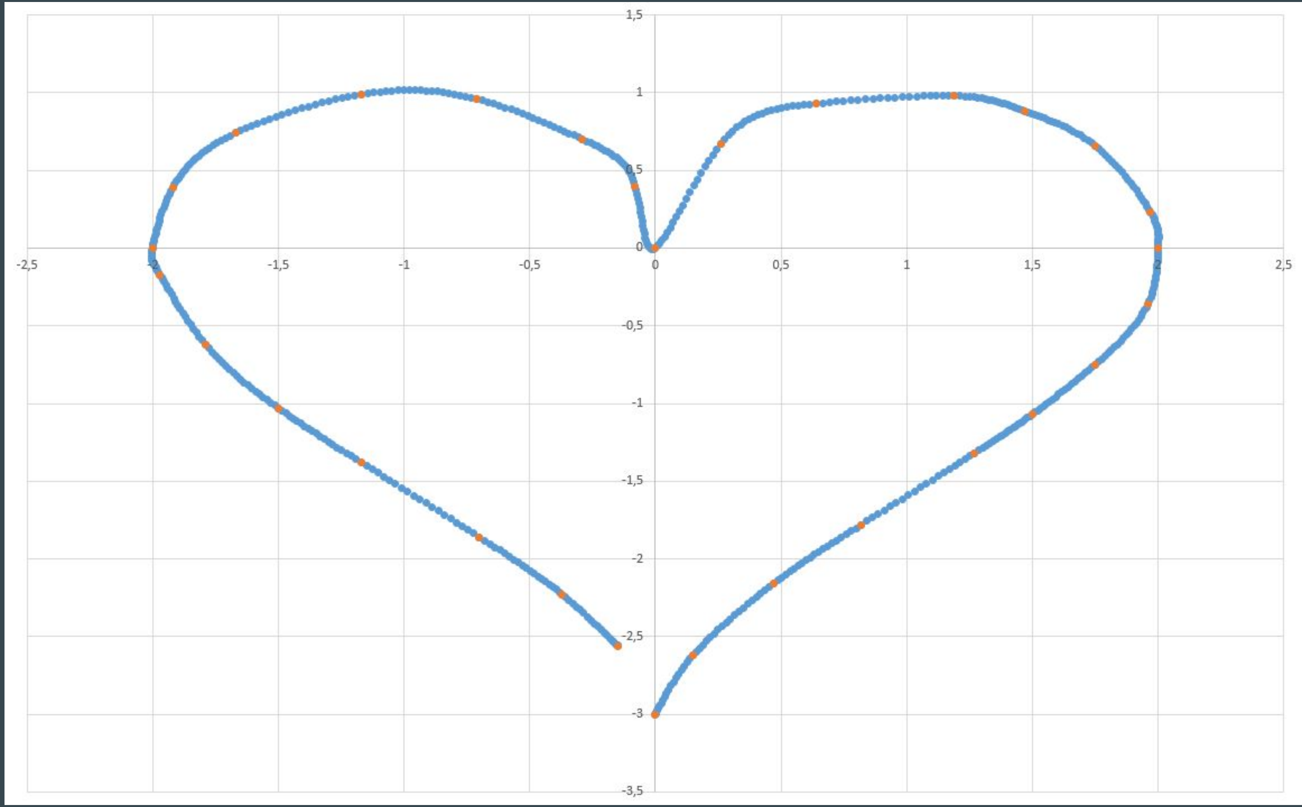
Heart - aberto e não uniforme



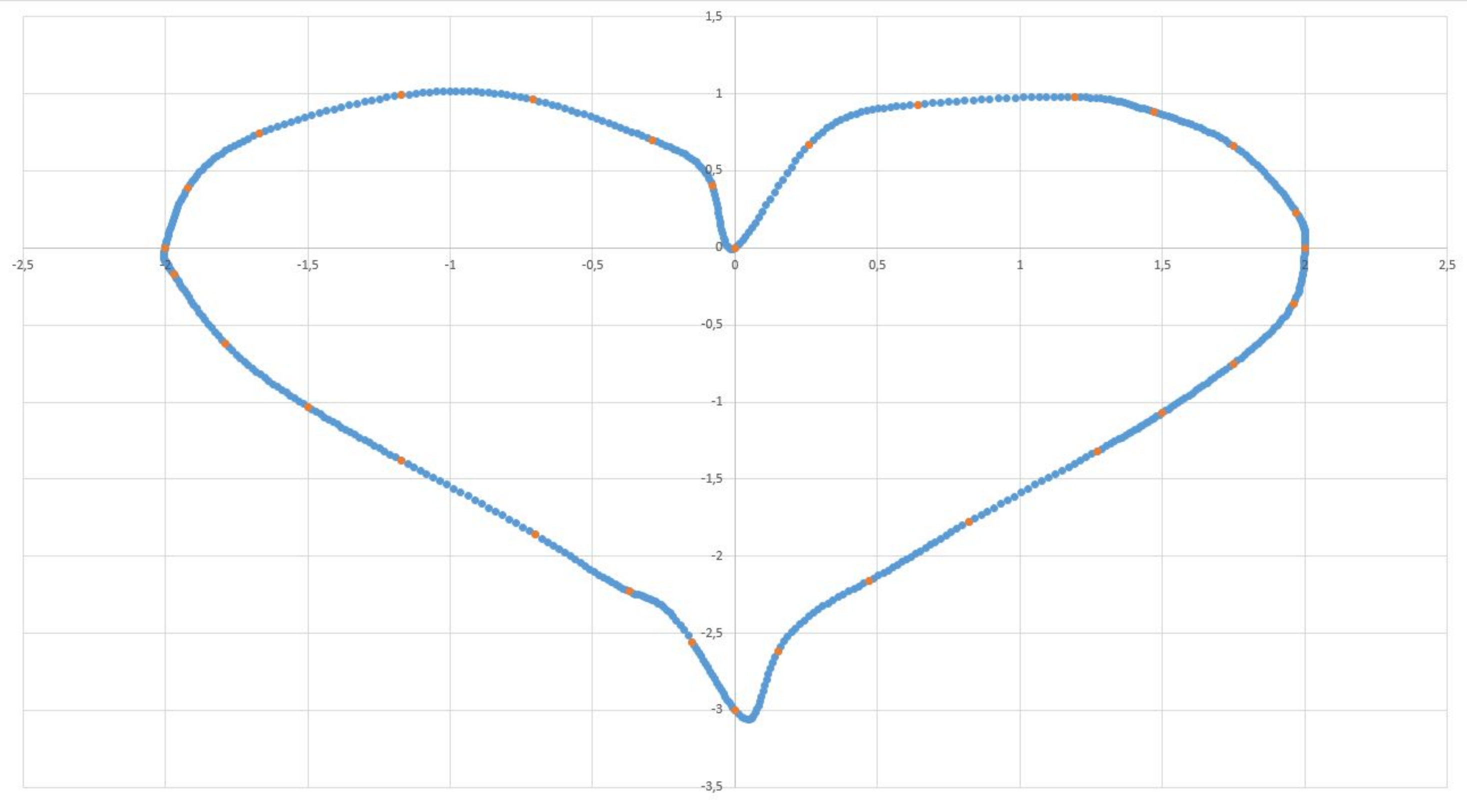
Heart - fechado e não uniforme



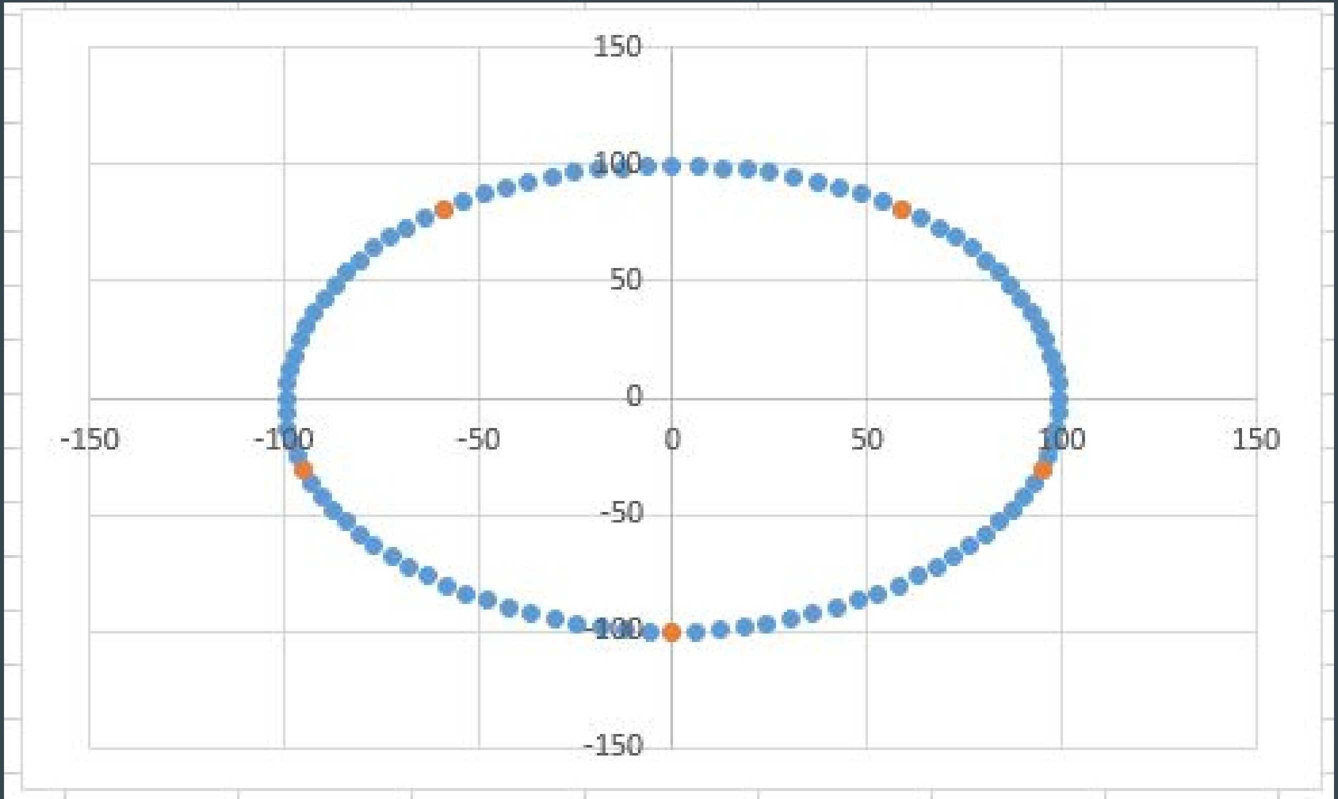
Heart - aberto e uniforme



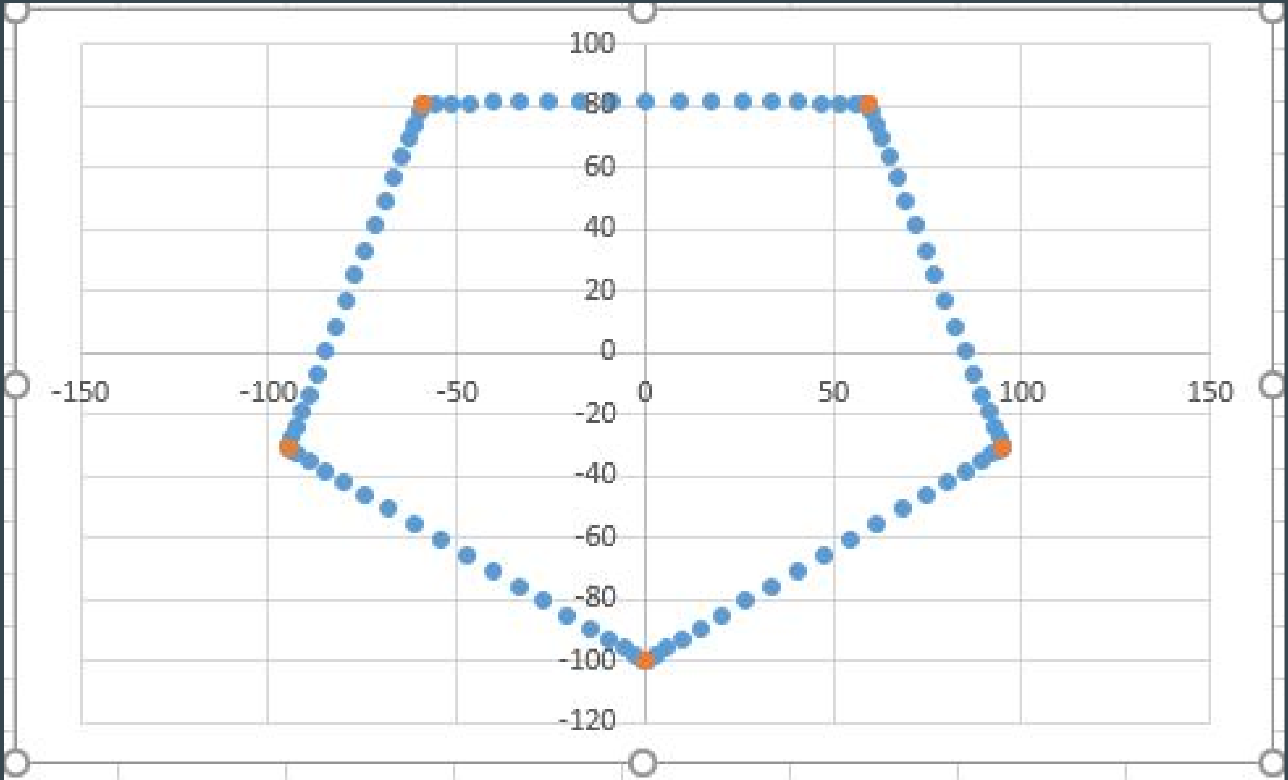
Heart - fechado e uniforme



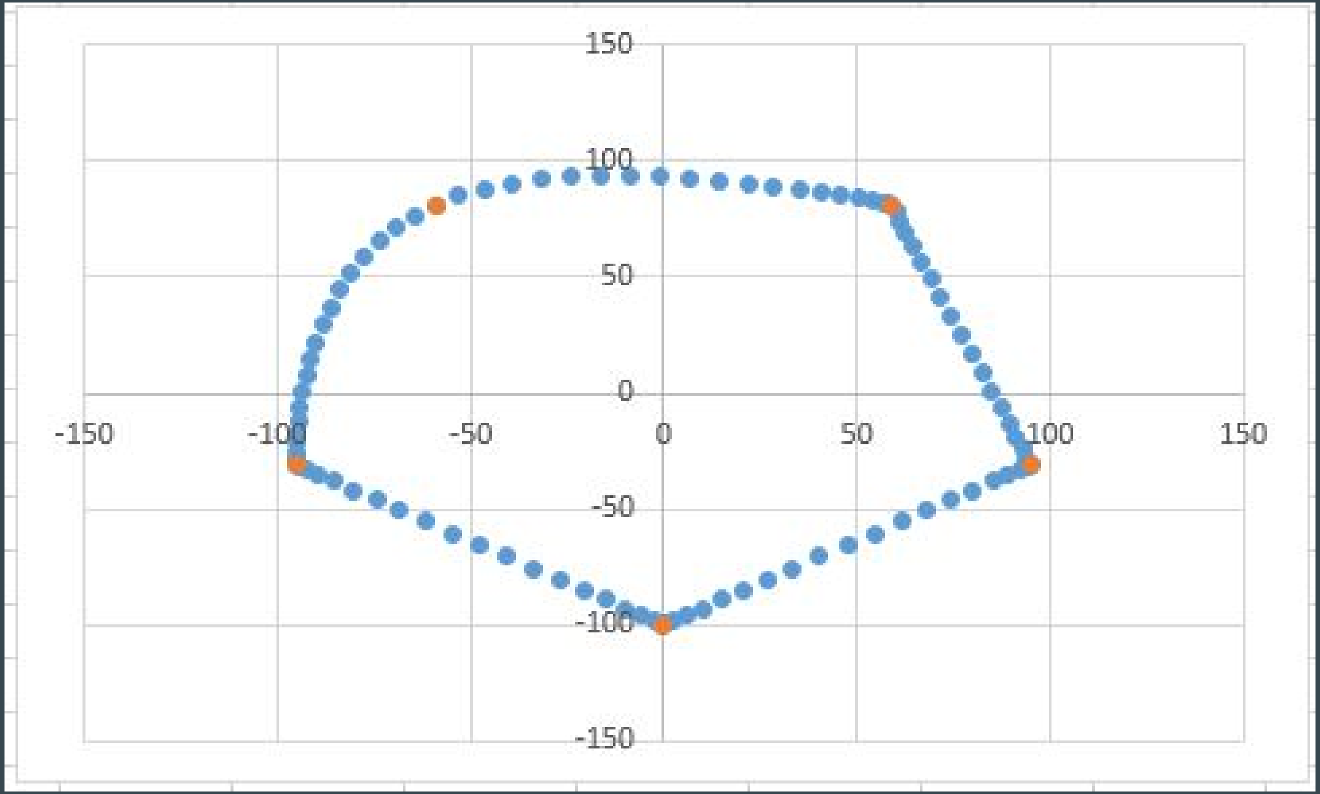
Pentágono???



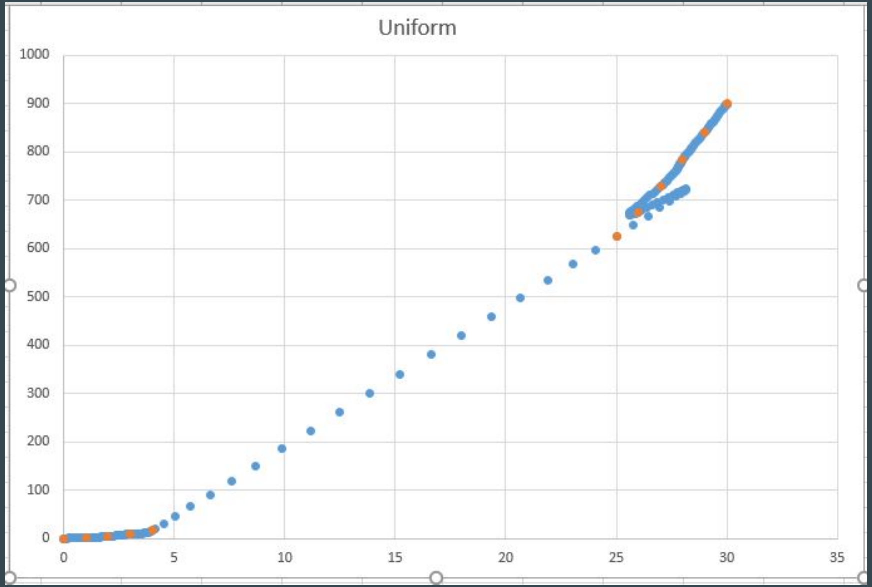
Tensão nos vértices: 200

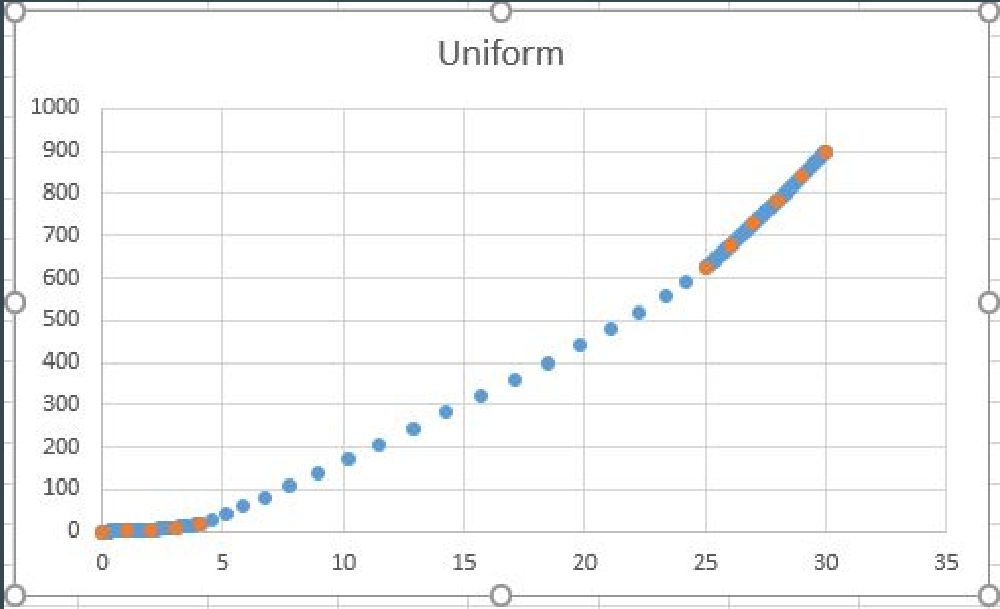


Tensão nos vértices: 150, 50, 0, 200, 100

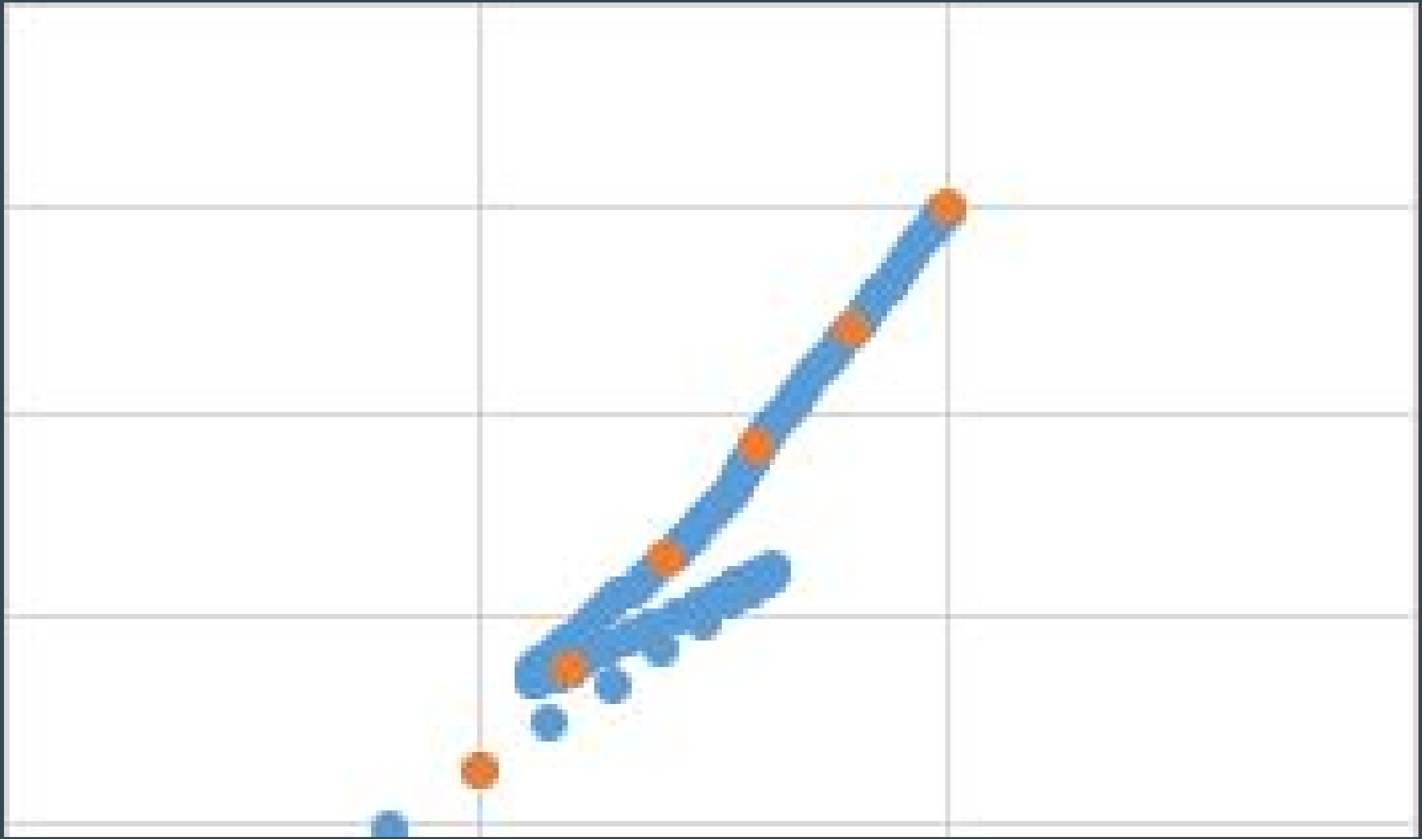


Uniforme vs. não uniforme

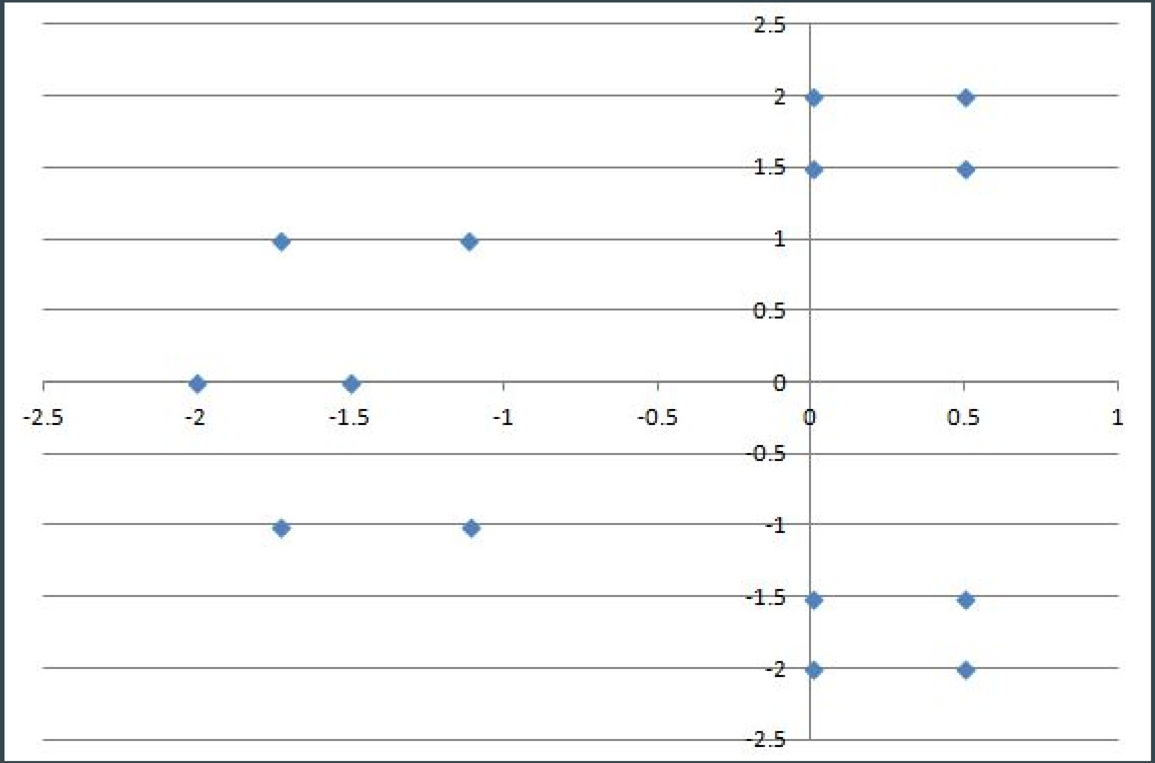




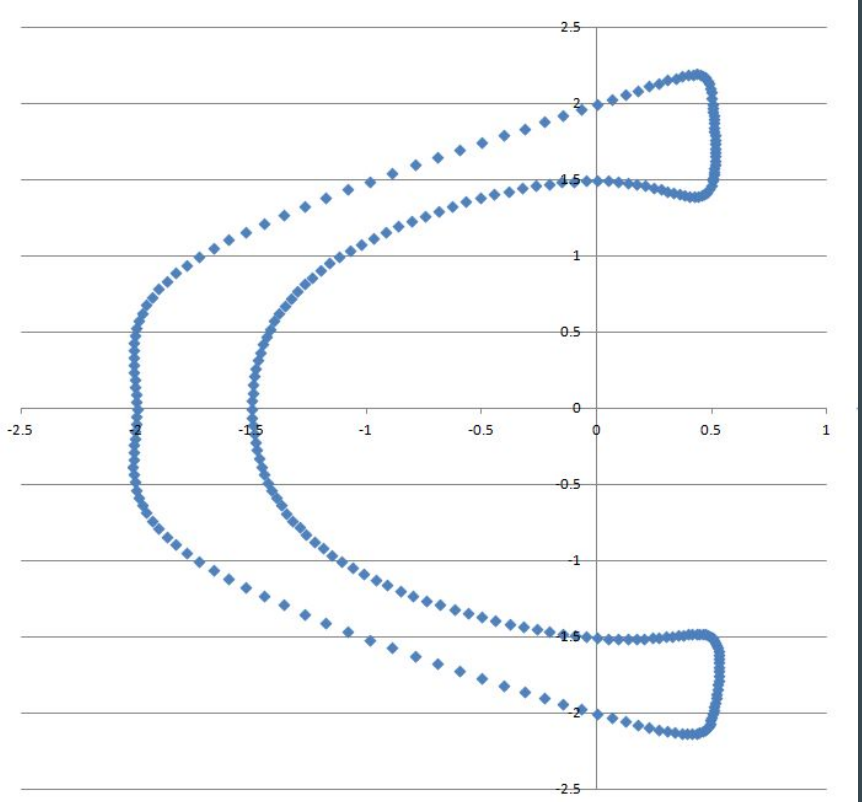
Detalhe: uniforme



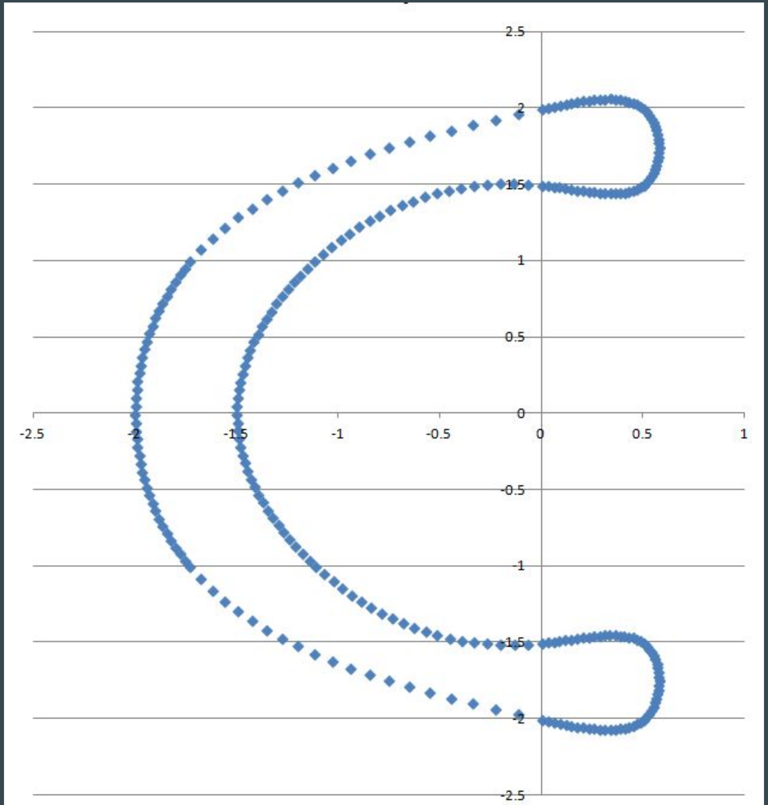
Teste em formato de C - imagem original



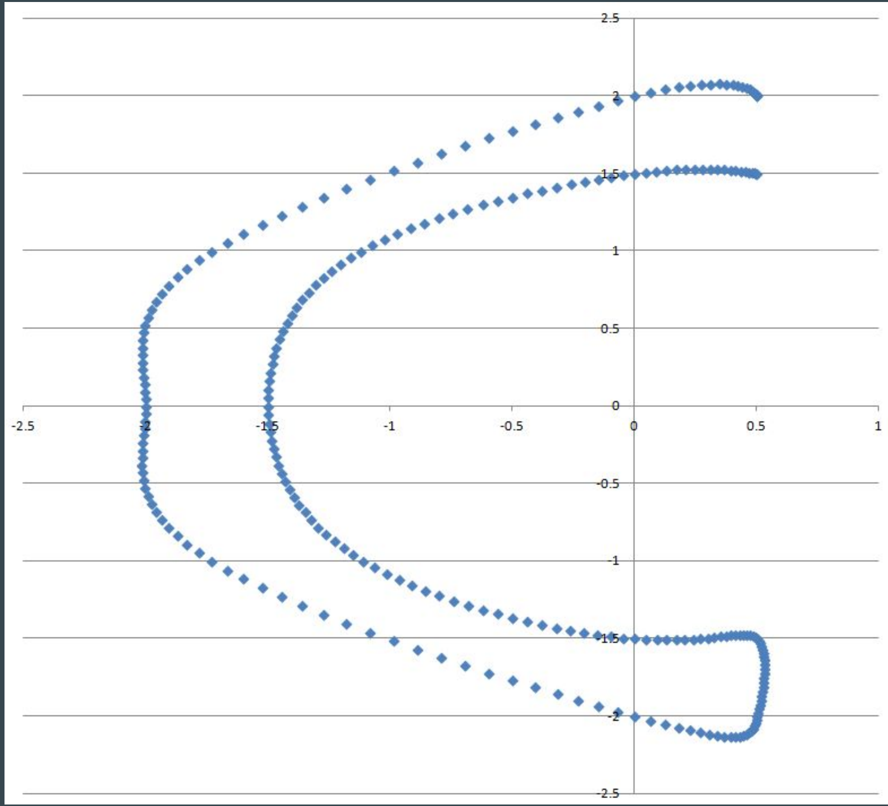
Teste em formato de C - fechada e uniforme



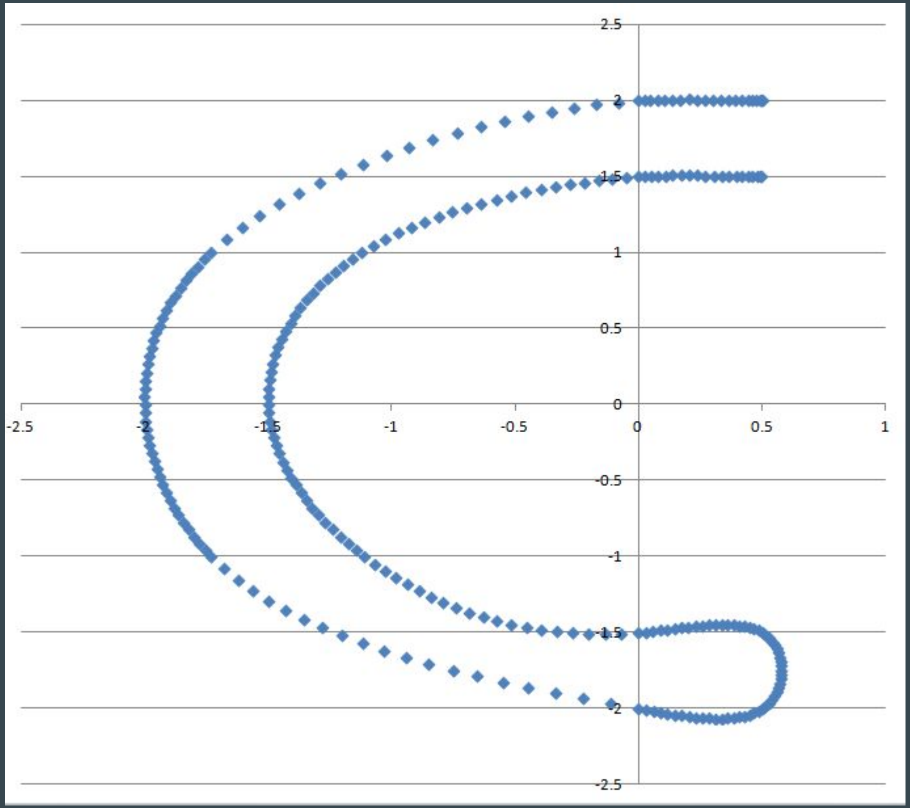
Teste em formato de C - fechada e não uniforme



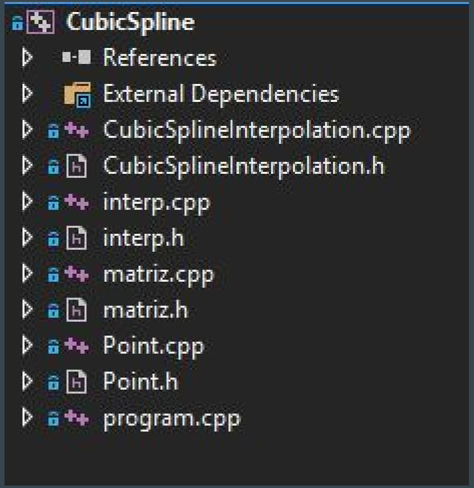
Teste em formato de C - aberta e uniforme



Teste em formato de C - aberta e não uniforme



Implementação



CubicSplineInterpolation → Cálculos (CubicSpline)

interp.cpp → Fatoração LU

matriz.cpp → Operações com matrizes

point.cpp → Ponto 2D (x,y)

program.cpp → Leitura / Escrita de arquivos csv (input/output)

CubicSplineInterpolation

```
class CubicSplineInterpolation
{
public:

    // Constructor recebe array de pontos e num de pontos = dimensão do problema
    CubicSplineInterpolation(Point *points, int n) : CubicSplineInterpolation(points, n, true, true) {};

    // Constructor recebe array de pontos e num de pontos = dimensão do problema
    CubicSplineInterpolation(Point *points, int n, bool openSpline, bool uniformSpline);

    // Realiza contas para interpolação cúbica (spline)
    void calculateSpline();

    // Retorna um ponto no tramo i, com parâmetro t
    Point calculatePoint(int i, double t);

    void setTension(int i, double tension);
```

Cálculo de R e L

CubicSplineInterpolation.cpp*

CubicSpline

```cpp

// Calcula R e L
if (_openSpline)
{
    _R = new Point[_n - 1];
    _L = new Point[_n - 1];
}
else
{
    _R = new Point[_n];
    _L = new Point[_n];
}
double Rx, Ry, Lx, Ly;
for (int i = 0; i < _n - 1; i++)
{
    Rx = (1 - _mi[i]) * tempDX[i] + _mi[i] * tempDX[i + 1];
    Ry = (1 - _mi[i]) * tempDY[i] + _mi[i] * tempDY[i + 1];

    Lx = (1 - _lambda[i + 1]) * tempDX[i] + _lambda[i + 1] * tempDX[i + 1];
    Ly = (1 - _lambda[i + 1]) * tempDY[i] + _lambda[i + 1] * tempDY[i + 1];

    _R[i] = Point(Rx, Ry);
    _L[i] = Point(Lx, Ly);
}

//calcula último tramo entre último ponto e primeiro ponto
if (!_openSpline)
{
    Rx = (1 - _mi[_n - 1]) * tempDX[_n - 1] + _mi[_n - 1] * tempDX[0];
    Ry = (1 - _mi[_n - 1]) * tempDY[_n - 1] + _mi[_n - 1] * tempDY[0];

    Lx = (1 - _lambda[0]) * tempDX[_n - 1] + _lambda[0] * tempDX[0];
    Ly = (1 - _lambda[0]) * tempDY[_n - 1] + _lambda[0] * tempDY[0];

    _R[_n - 1] = Point(Rx, Ry);
    _L[_n - 1] = Point(Lx, Ly);
}

```

LU vs. Thomas

CubicSplineInterpolation.cpp

CubicSpline

```cpp
if (_openSpline)
{
    Thomas(_n, a, b, c, tempPX, tempPY, tempDX, tempDY);
}
else
{
    //fatoração LU:
    double **P = fatoracao(_n, _A);
    //printMatriz(_n, _n, _A);
    //printMatriz(_n, _n, P);
    tempDX = substituicao(_n, _A, P, tempPX);
    tempDY = substituicao(_n, _A, P, tempPY);
}
```

Casteljau - iterativo

CubicSplineInterpolation.cpp

CubicSpline

CubicSplineInte

```
//calcula castejau de b(3)(i)(t) (usando k = 3), não recursivamente
double CubicSplineInterpolation::casteljauB(double b0, double b1, double b2, double b3, double t)
{
    double c1 = (1 - t)*((1 - t)*((1 - t)*b0 + t*b1) + t*((1 - t)*b1 + t*b2));
    double c2 = t*((1 - t)*((1 - t)*b1 + t*b2) + t*((1 - t)*b2 + t*b3));
    return c1 + c2;
}
```

Método de Thomas

Método

Uma matriz tridiagonal pode ser escrita na forma:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i,$$

onde $a_1 = 0$ e $c_n = 0$.

e cuja representação na forma matricial se dá por:

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}.$$

O primeiro passo consiste em alterar os coeficientes da seguinte forma:

$$c'_i = \begin{cases} \dfrac{c_i}{b_i} & ; \quad i = 1 \\ \dfrac{c_i}{b_i - c'_{i-1} a_i} & ; \quad i = 2, 3, \dots, n-1 \end{cases}$$

O primeiro passo consiste em alterar os coeficientes da seguinte forma:

$$c'_i = \begin{cases} \dfrac{c_i}{b_i} & ; \quad i = 1 \\ \dfrac{c_i}{b_i - c'_{i-1} a_i} & ; \quad i = 2, 3, \dots, n-1 \end{cases}$$

Marcando com o superíndice ' os coeficientes recém alterados.

Da mesma forma faz-se:

$$d'_i = \begin{cases} \dfrac{d_i}{b_i} & ; \quad i = 1 \\ \dfrac{d_i - d'_{i-1} a_i}{b_i - c'_{i-1} a_i} & ; \quad i = 2, 3, \dots, n. \end{cases}$$

A solução é então obtida pela substituição de volta:

$$x_n = d'_n$$
$$x_i = d'_i - c'_i x_{i+1} \qquad ; \ i = n-1, n-2, \dots, 1.$$

Método de Thomas

```
void CubicSplineInterpolation::Thomas(int n, double * a, double * b, double * c, double * dX, double *dY, double * x, double *y)
{
    double *cl = (double*)malloc(n * sizeof(double));
    double *dlX = (double*)malloc(n * sizeof(double));
    double *dlY = (double*)malloc(n * sizeof(double));


    if (cl == nullptr || dlX == nullptr || dlY == nullptr)
    {
        std::cout << "Unable to allocate memory." << std::endl;
        exit(1);
    }

    cl[0] = c[0] / b[0];
    dlX[0] = dX[0] / b[0];
    dlY[0] = dY[0] / b[0];

    for (int i = 1; i < n; i++)
    {
        double div = (b[i] - a[i] * cl[i - 1]);
        cl[i] = c[i] / div;

        dlX[i] = (dX[i] - a[i] * dlX[i - 1]) / div;

        dlY[i] = (dY[i] - a[i] * dlY[i - 1]) / div;
    }

    x[n - 1] = dlX[n - 1];
    y[n - 1] = dlY[n - 1];

    for (int i = n-2; i >= 0 ; i--)
    {
        x[i] = dlX[i] - cl[i] * x[i + 1];
        y[i] = dlY[i] - cl[i] * y[i + 1];
    }
}
```