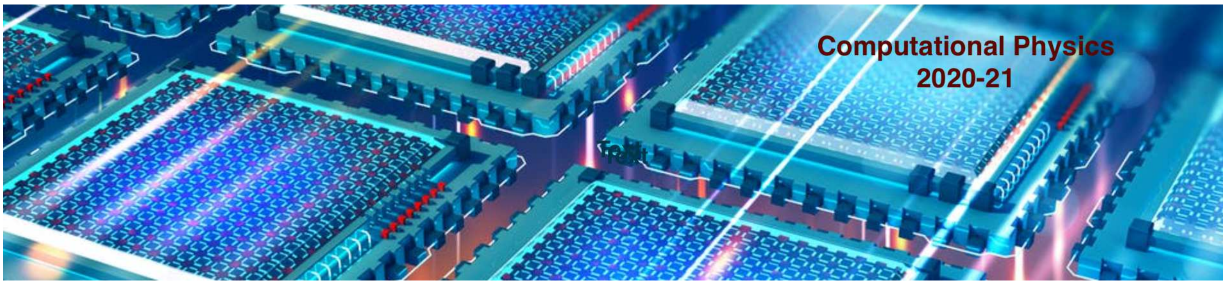
# Computational Physics

***numerical methods with C++ (and UNIX)***

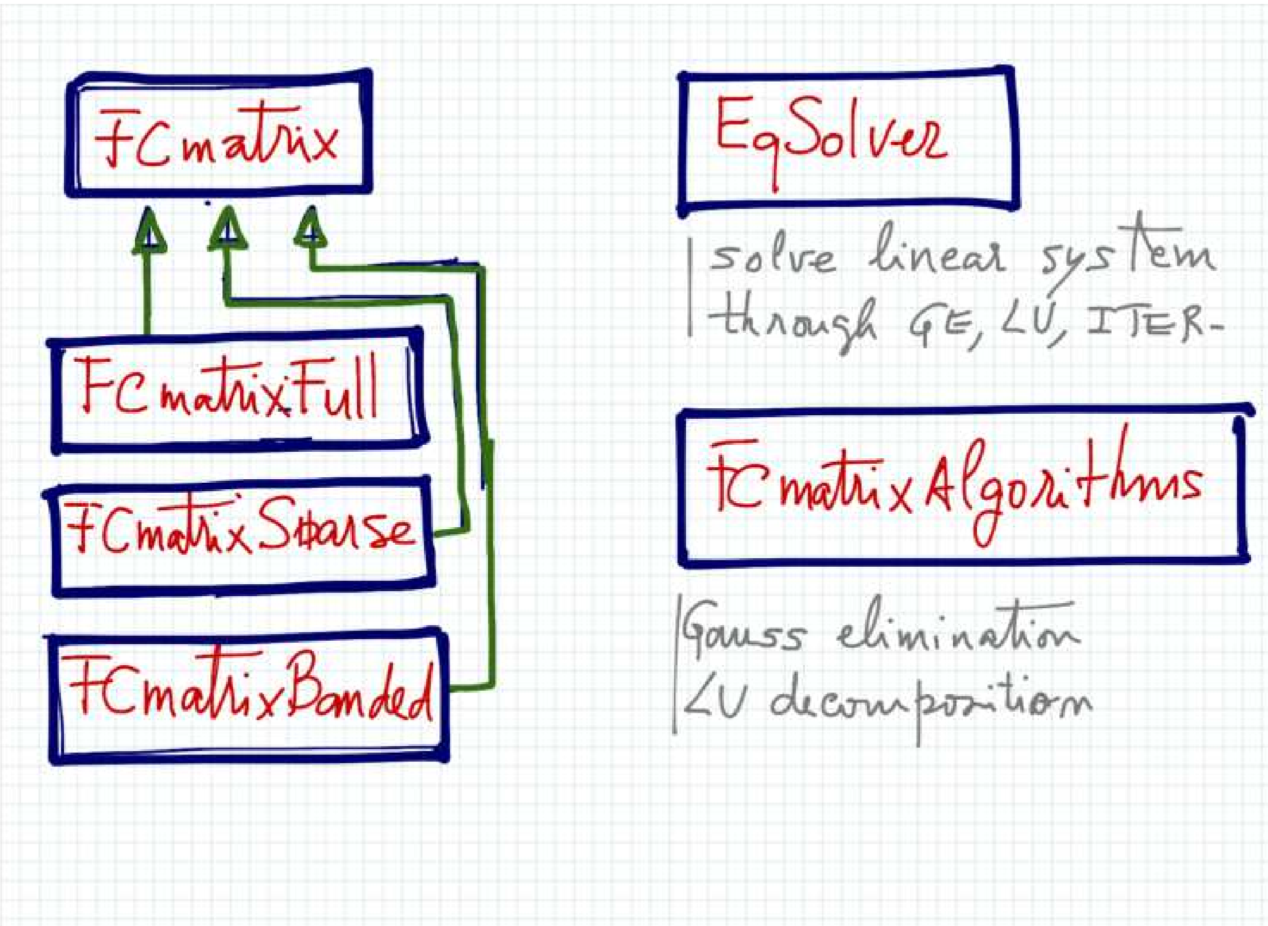
***2020-21***




Fernando Barao
Instituto Superior Tecnico, Dep. Fisica
email: fernando.barao@tecnico.ulisboa.pt



## Linear system of equations: classes



# Linear system of equations: classes

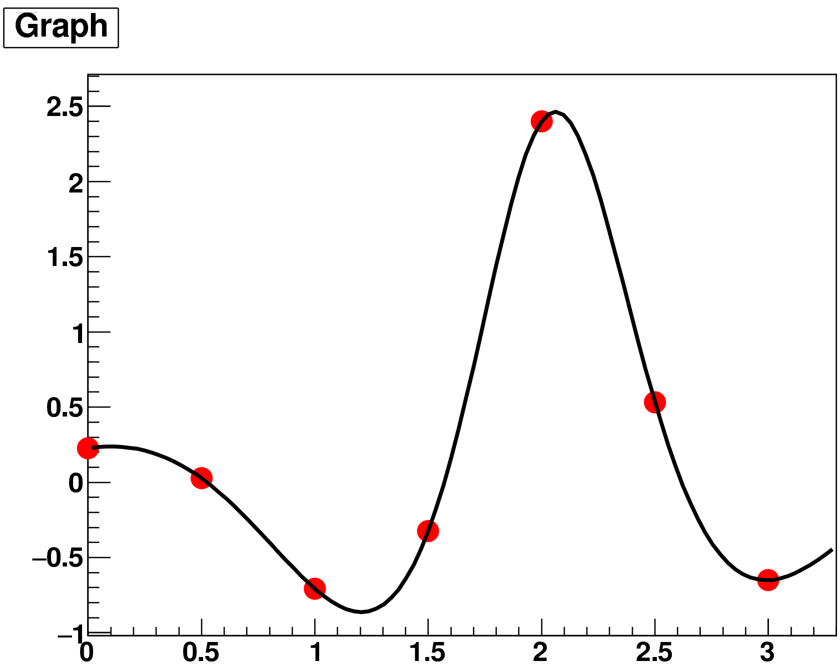
## FCmatrixAlgorithm header

```
class FCmatrixAlgorithm {
 public:
/*
Implements Gauss elimination
- It can be a simple matrix (A)
- It can be the augmented matrix (A | b) for a linear syst solution
- If returns the argument matrix (A or A|b) + extra column with row
  indices
*/
  //default flag=0, no pivoting
  static void GaussElimination(FCmatrix& , int flag=0);

/*
Implements LU decomposition (Doolitle)
- Decomposition of a simple matrix (A)
- If returns the argument matrix (A) + extra column with row indices
*/
  static void LUdecomposition(FCmatrix& , int flag=0);
};
```



# interpolation example

$$f(x) = \frac{\cos(3x)}{0.4 + (x-2)^2}$$



# DataPoints base class

```
#ifndef __DataPoints__
#define __DataPoints__

#include "cFCgraphics.h" // Note: This class is not being used on 2020-21 co

class DataPoints {
 public:
  DataPoints();
  DataPoints(int, double*, double*);
  virtual ~DataPoints();

  virtual double Interpolate(double x) {return 0.;}
  virtual void Draw();
 protected:
  int N; // number of data points
  double *x, *y; // arrays
  cFCgraphics G; // not being used on 2020-21
};
#endif
```



# DataPoints class code

```
#include "DataPoints.h"
#include "TGraph.h"
#include <algorithm>

// using initialization list of constructor
DataPoints::DataPoints() : N(0), x(nullptr), y(nullptr) { ; }

DataPoints::DataPoints(int fN, double* fx, double* fy) :
N(fN) ,
x(new double[N]),
y(new double[N]) {
  std::copy(fx, fx+N, x);
  std::copy(fy, fy+N, y);
}

DataPoints::~DataPoints() {
  delete [] x;
  delete [] y;
}
```

# DataPoints class code (cont.)

Example of drawing using *cFCgraphics* class

```
void DataPoints::Draw() {
  TGraph *g = new TGraph(N,x,y);
  g->SetMarkerStyle(20);
  g->SetMarkerColor(kRed);
  g->SetMarkerSize(2.5);
  TPad *pad1 = G.CreatePad("pad1");
  G.AddObject(g,"pad1","AP");
  G.AddObject(pad1);
  G.Draw();
}
```

Example returning *TGraph*

```
TGraph* DataPoints::Draw() {
  TGraph *g = new TGraph(N,x,y);
  g->SetMarkerStyle(20);
  g->SetMarkerColor(kRed);
  g->SetMarkerSize(2.5);
  return g;
}
```



# Neville interpolator class

```
#ifndef __NevilleInterpolator__
#define __NevilleInterpolator__

#include "DataPoints.h"
class NevilleInterpolator : public DataPoints {

 public:
  NevilleInterpolator(int N=0, double *x=NULL, double *y=NULL);
  ~NevilleInterpolator() {;}

  // redefine Interpolate method
  double Interpolate(double x);
  void Draw();

  void SetFunction(TF1* f=nullptr) { // underlying function
    if (f) F0=f;
  }

 private:
  double fInterpolator(double *fx, double *par) {
    return Interpolate(fx[0]);
  }
  TF1* FInterpolator;
  TF1* F0; // underlying function from where points
           // were extracted
};
#endif
```

# Neville interpolator algorithm: reminder

| x | 0th order | 1st order | 2nd order | 3rd order | korder |
|---|---|---|---|---|---|
| $x_0$ | $P_0(x_0) = y_0$ | | | | |
| $x_1$ | $P_0(x_1) = y_1$ | $P_1[x_0, x_1]$ | | | |
| $x_2$ | $P_0(x_2) = y_2$ | $P_1[x_1, x_2]$ | $P_2[x_0, x_1, x_2]$ | | |
| $x_3$ | $P_0(x_3) = y_3$ | $P_1[x_2, x_3]$ | $P_2[x_1, x_2, x_3]$ | $P_3[x_0, x_1, x_2, x_3]$ | |
| $x_4$ | $P_0(x_4) = y_4$ | $P_1[x_3, x_4]$ | $P_2[x_2, x_3, x_4]$ | $P_3[x_1, x_2, x_3, x_4]$ | |
| $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | | |
| $x_n$ | $P_0(x_n) = y_n$ | $P_1[x_{n-1}, x_n]$ | $P_2[x_{n-2}, x_{n-1}, x_n]$ | $P_3[x_{n_3}, x_{n-2}, x_{n-1}, x_n]$ | |

n+1 points: $i = 0, \cdots, n$

order: $k = 1, \cdots, n$

Algorithm: k order $= 1, \cdots, n \quad i = k, \cdots, n$

$$P_{k,i} = \frac{(x - x_i)\; P_{i-k,i-1} - (x - x_{i-k})\; P_{k-1,i}}{x_{i-k} - x_i} \qquad P_{k,i} \equiv P[x_{i-k}, \cdots, x_i]$$



# Neville interpolator class

```
NevilleInterpolator::NevilleInterpolator(int fN, double *fx, double *fy) : DataPoints(fN, fx, fy) {
  FInterpolator = new TF1("FInterpolator", this, &NevilleInterpolator::fInterpolator,
                          -0.1 ,3.1, 0, "NevilleInterpolator", "fInterpolator");
  DataPoints::Print();
  F0=NULL;
}

double NevilleInterpolator::Interpolate(double xval) {
  // Neville algorithm

  double* yp = new double[N];
  for (int i=0; i<N; i++) {
    yp[i] = y[i]; // auxiliar vector
  }

  for (int k=1; k<N; k++) { // use extreme x-values
    for (int i=0; i<N-k; i++) {
      yp[i] = (
               (xval-x[i+k])*yp[i] -
               (xval-x[i])*yp[i+1]) / (x[i]-x[i+k]);
    }
  }
  double A = yp[0];
  delete [] yp;
  return A;
}
```

Suppose 3 points ($N = 3$)

| $k$ | $i$ | |
|---|---|---|
| 1 | 0 | $(x_0 - x_1)^{-1}\ [(x - x_1)y_0 - (x - x_0)y_1]$ |
| | 1 | $(x_1 - x_2)^{-1}\ [(x - x_2)y_1 - (x - x_1)y_2]$ |
| 2 | 0 | $(x_0 - x_2)^{-1}\ [(x - x_2)y_0 - (x - x_0)y_1]$ |

the interpolated value at $x$ is the last computed value and is stored on $y[0]$

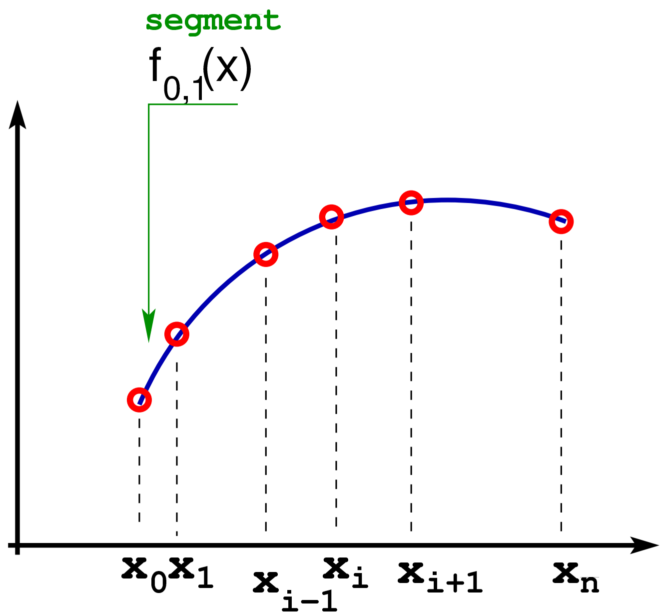
# Limitations of polynomial interpolation

- ✔ The need of knowing with a better precision an interpolation carries the solution of adding more and more points to our interpolation
  - ☞ a polynomial interpolation passing through a large number of points (degree higher than $\sim 5, 6$) can give a wrong interpolation in some segments due to *wild* oscillations
  - ☞ if the number of points (knots) is large, an eventual linear interpolation by segments is enough!
  - ☞ otherwise a degree 3 to 6 polynomial interpolation by segment
- ✔ polynomial extrapolation (interpolating outside the range of data points) is dangerous!



# Cubic spline method

- ✔ The interpolation can be performed in a given segment $[x_i, x_{i+1}]$ using a **cubic polynomial** (4 parameters to find)
- ✔ Apart from the two points data associated to the segment we ask for continuity of the 1st and 2nd derivatives at the knot $x_{i+1}$, i.e., the intersection of two segments
  - ☞ no bending at the end points ($x_0$ and $x_n$) $\Rightarrow$ 2nd derivative=0




- ✔ **The spline will be a piecewise cubic curve, put together from the $n$ cubic polynomials:** $f_{0,1}(x), f_{1,2}(x), \cdots, f_{n-1,n}(x)$

# Cubic spline method (cont.)

- Suppose we have $N = n + 1 = 6$ data knots with abcissas
  $x_0, x_1, x_2, x_3, x_4, x_5 \quad (i = 0, \cdots, n)$
- The number of intervals will be $N - 1 = n = 5$
- On every interval $[x_i, x_{i+1}]$ there will be an interpolating function $f_{x_i,x_{i+1}}$ defined by a cubic polynomial
  $f_{x_i,x_{i+1}}(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \quad (i = 0, \cdots, n - 1)$
- The set of four parameters $(a_i, b_i, c_i, d_i)$ for the interpolating cubic spline $f_j(x) \quad (j = 0, \cdots, n - 1)$ will be derived from the following conditions:

$\boxed{[x_0, x_1]}$

$$\begin{aligned} f_0(x) &= a_0 + b_0(x - x_0) + c_0(x - x_0)^2 + b_0(x - x_0)^3 \\ f_0(x_0) &= y_0 \\ f_0(x_1) &= y_1 \\ f_0'(x_0) &= y_0' \quad \text{numerically} \\ f_0''(x_0) &= 0 \end{aligned}$$

$\boxed{[x_1, x_2]}$

$$\begin{aligned} f_1(x) &= a_1 + b_1(x - x_1) + c_1(x - x_1)^2 + b_1(x - x_1)^3 \\ f_1(x_1) &= y_1 \\ f_1(x_2) &= y_2 \\ f_1'(x_1) &= f_0'(x_1) \\ f_1''(x_1) &= f_0''(x_1) \end{aligned}$$



# Cubic spline method (cont.)

- the continuity of the 2nd derivative of the spline at knot $i$ gives:
  $f''_{i-1,i}(x_i) = f''_{i,i+1}(x_i) = K_i \qquad (i = 1, \cdots, n - 1)$
  the 2nd derivative at the extremes:
  $f''(x_0) \equiv K_0 = f''(x_n) \equiv K_n = 0$
- the second derivative expression for any segment $[x_i, x_{i+1}]$, is a linear polynomial

  Using the Lagrange polynomial linear interpolator,
  $f''_{i,i+1}(x) = f''(x_i)\ell_i(x) + f''(x_{i+1})\ell_{i+1}(x)$

  whith the cardinal functions given by:
  $\ell_i(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}}$
  $\ell_{i+1}(x) = \frac{x - x_i}{x_{i+1} - x_i}$

$$f''_{i,i+1}(x) = \frac{K_i(x - x_{i+1}) - K_{i+1}(x - x_i)}{x_i - x_{i+1}}$$

## Cubic spline method (cont.)

✔ Integrating now twice:

$$f'_{i,i+1}(x) = \frac{1}{x_i - x_{i+1}} \left[ \frac{K_i}{2}(x - x_{i+1})^2 - \frac{K_{i+1}}{2}(x - x_i)^2 \right] + A$$

$$f_{i,i+1}(x) = \frac{1}{x_i - x_{i+1}} \left[ \frac{K_i}{6}(x - x_{i+1})^3 - \frac{K_{i+1}}{6}(x - x_i)^3 \right] + Ax + B$$

And redefining the constants $A$ and $B$ we can write the cubic spline for the segment:

$$\boxed{f_{i,i+1}(x) = \frac{1}{x_i - x_{i+1}} \left[ \frac{K_i}{6}(x - x_{i+1})^3 - \frac{K_{i+1}}{6}(x - x_i)^3 \right] + A(x - x_{i+1}) + B(x - x_i)}$$



## Cubic spline method (cont.)

✔ The extreme values of the function on the segment provide A and B:

$$f_{i,i+1}(x_i) = y_i \quad \Rightarrow \quad \frac{1}{x_i - x_{i+1}} \left[ \frac{K_i}{6}(x_i - x_{i+1})^3 \right] + A(x_i - x_{i+1}) = y_i$$

$$\Rightarrow \quad A = \frac{y_i}{x_i - x_{i+1}} - \frac{K_i}{6}(x_i - x_{i+1})$$

$$f_{i,i+1}(x_{i+1}) = y_{i+1} \quad \Rightarrow \quad \frac{1}{x_i - x_{i+1}} \left[ -\frac{K_{i+1}}{6}(x_{i+1} - x_i)^3 \right] + B(x_{i+1} - x_i) = y_{i+1}$$

$$\Rightarrow \quad B = \frac{y_{i+1}}{x_i - x_{i+1}} - \frac{K_{i+1}}{6}(x_i - x_{i+1})$$

$$\boxed{\begin{aligned} f_{i,i+1}(x) = \; & \frac{K_i}{6} \left[ \frac{(x - x_{i+1})^3}{x_i - x_{i+1}} - (x - x_{i+1})(x_i - x_{i+1}) \right] \\ & - \frac{K_{i+1}}{6} \left[ \frac{(x - x_i)^3}{x_i - x_{i+1}} - (x - x_i)(x_i - x_{i+1}) \right] + \frac{y_i(x - x_{i+1}) - y_{i+1}(x - x_i)}{x_i - x_{i+1}} \end{aligned}}$$

# Cubic spline method (cont.)

✔ The 1st derivative for the segment $[x_i, x_{i+1}]$ is given by:

$$f'_{i,i+1}(x) = \frac{K_i}{2}\left[\frac{(x-x_{i+1})^2}{x_i-x_{i+1}} - \frac{x_i-x_{i+1}}{3}\right] - \frac{K_{i+1}}{2}\left[\frac{(x-x_i)^2}{x_i-x_{i+1}} - \frac{x_i-x_{i+1}}{3}\right] + \frac{y_i-y_{i+1}}{x_i-x_{i+1}}$$

✔ The second derivatives values $(K_i)$ of the spline in the interior knots, are obtained from the first derivative condition:

$$f'_{i-1,i}(x_i) = f'_{i,i+1}(x_i) \qquad (i = 1, 2, \cdots, n-1)$$

$$K_{i-1}(x_{i-1}-x_i)+2K_i(x_{i-1}-x_{i+1})+K_{i+1}(x_i-x_{i+1}) = 6\left(\frac{y_{i-1}-y_i}{x_{i-1}-x_i} - \frac{y_i-y_{i+1}}{x_i-x_{i+1}}\right)$$

(trilinear matrix to solve!)



# Cubic spline method (cont.)

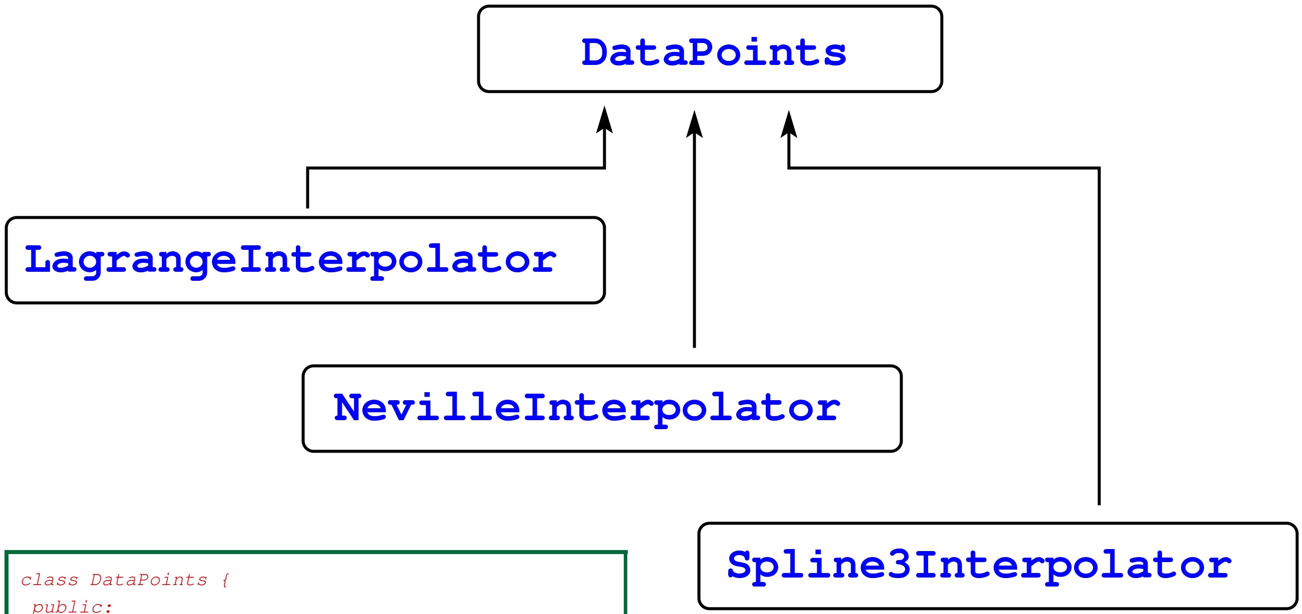
The set of equations to solve:

$$\begin{aligned}
2K_1(x_0-x_2) + K_2(x_1-x_2) &= \cdots \quad (i=1)\\
K_1(x_1-x_2) + 2K_2(x_1-x_3) + K_3(x_2-x_3) &= \cdots \quad (i=2)\\
K_2(x_2-x_3) + 2K_3(x_2-x_4) + K_4(x_3-x_4) &= \cdots \quad (i=3)\\
K_3(x_3-x_4) + 2K_4(x_3-x_5) + K_5(x_4-x_5) &= \cdots \quad (i=4)\\
\cdots &= \cdots \quad (i=n-1)
\end{aligned}$$

which corrresponds to a tri-diagonal matrix:

$$\begin{pmatrix}
2(x_0-x_2) & (x_1-x_2) & 0 & 0 & 0 & \cdots\\
(x_1-x_2) & 2(x_1-x_3) & (x_2-x_3) & 0 & 0 & \cdots\\
0 & (x_2-x_3) & 2(x_2-x_4) & (x_3-x_4) & 0 & \cdots\\
0 & 0 & (x_3-x_4) & 2(x_3-x_5) & (x_4-x_5) & \cdots\\
0 & 0 & 0 & \cdots & \cdots & \cdots
\end{pmatrix}
\begin{pmatrix} K_1\\ K_2\\ K_3\\ K_4\\ \cdots \end{pmatrix}
=
\begin{pmatrix} \cdots\\ \cdots\\ \cdots\\ \cdots\\ \cdots \end{pmatrix}$$

# classes scheme




```
class DataPoints {
 public:
  DataPoints();
  DataPoints(int, double*, double*);
  virtual ~DataPoints();
  virtual double Interpolate(double x) {return 0.;}
  virtual void Draw();
 protected:
  int N; // number of data points
  double *x, *y; // arrays
  cFCgraphics G;
};
```

```
class Spline3Interpolator : public DataPoints {
 public:
  Spline3Interpolator(int N=0, double *x=NULL, double *y=NULL);
  double Interpolate(double x);
  TF1* GetFInterpolator() {return FInterpolator;}
 private:
  void SetCurvatureLines();
  double fInterpolator(double *fx, double *par) {
    return Interpolate(fx[0]);
  }
  TF1* FInterpolator;
  double* K; //2nd derivatives
};
```



# Cubic spline: class algorithm

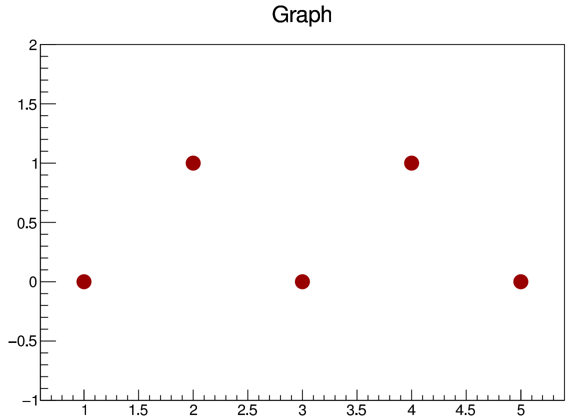
```
Spline3Interpolator::Spline3Interpolator(int fN, double *fx, double *fy) : DataPoints(fN,fx,fy) {
  DataPoints::Print();
  F0=NULL;
  FInterpolator = new TF1("FInterpolator", this, &Spline3Interpolator::fInterpolator,
                 x[0]-0.1 ,x[N-1]+0.1, 0)
  K = new double[N];
  SetCurvatureLines(); //define segment interpolators
}

void Spline3Interpolator::SetCurvatureLines() {
  // define tri-diagonal matrix and array of constants
  ...

  // solve system and get the 2nd derivative coefficients
  // store coeffs on internal array K
  ...
}

double Spline3Interpolator::Interpolate(double fx) {
// detect in wich segment is x
for (int i=0; i<N; i++) {
  if ((fx-x[i])<0.) break;
} //upper bound returned
if (i==0 || i==N-1) // out of range
 return 0.;

//retrieve segment interpolator and return function value
}
```

Code
to be
DEVELOPPED!

# Cubic spline: Problem

Utilizar o método do "cubic spline"para determinar o valor de $y(1.5)$, dados os seguintes valores:

| x | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| y | 0 | 1 | 0 | 1 | 0 |






# Cubic spline: solution

Determination of K's (second derivatives):

$i=1,\cdots,m-1$ $\quad$ ($m+1$ points) $\quad$ $m+1=5$, $m=4$

$$i=1: \quad K_0\underbrace{(x_0-x_1)}_{-1}+2K_1\underbrace{(x_0-x_2)}_{-2}+K_2\underbrace{(x_1-x_2)}_{-1}=$$

$$=6\left(\underbrace{\frac{y_0-y_1}{x_0-x_1}}_{-1/-1}-\underbrace{\frac{y_1-y_2}{x_1-x_2}}_{1/-1}\right)$$

$$\boxed{4K_1+K_2=-12}$$

[$K_0=0$, natural spline 3]

$$i=2: \quad K_1+4K_2+K_3=+12$$

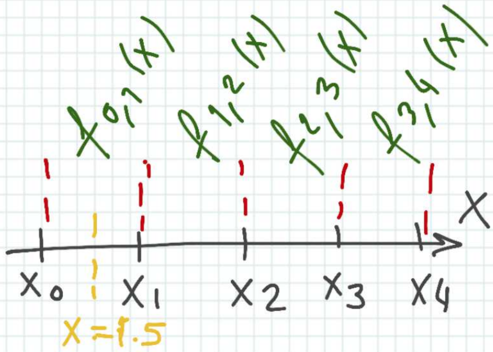
$$i=3: \quad K_2+4K_3+\underbrace{K_4}_{0}=-12$$

# Cubic spline: solution

sistem of equations

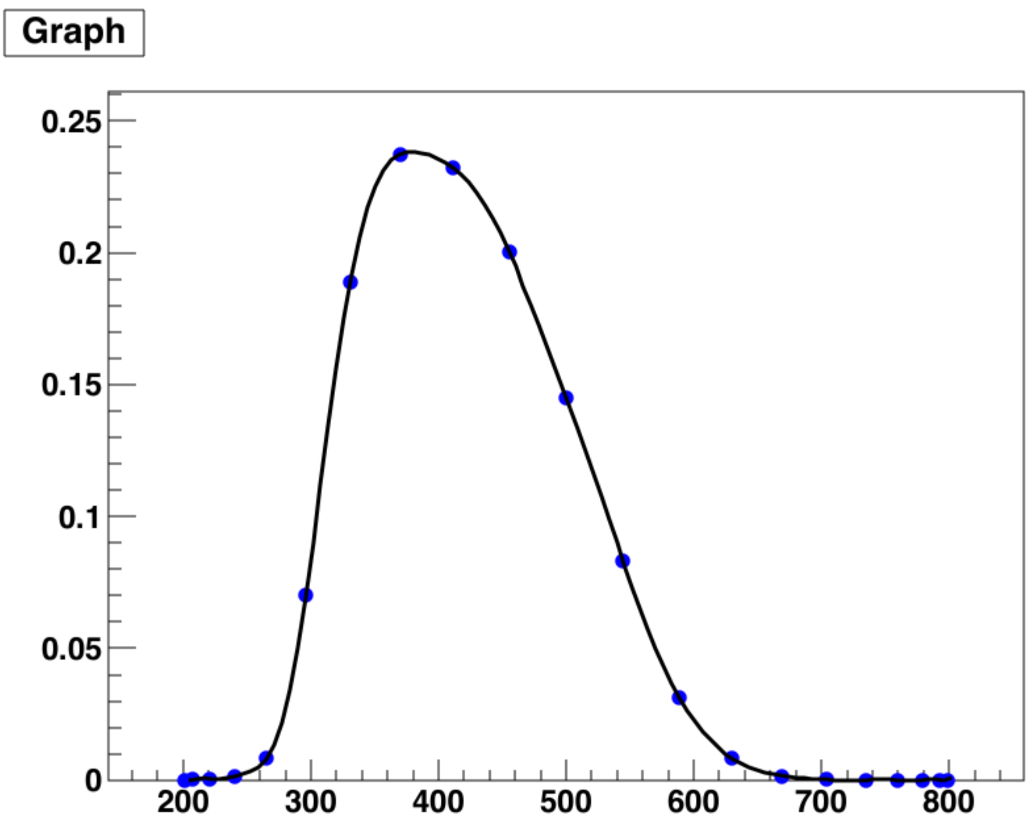
$$\begin{cases} 4K_1 + K_2 = -12 \\ K_1 + 4K_2 + K_3 = 12 \\ K_2 + 4K_3 = -12 \end{cases} \qquad \left[\begin{array}{ccc|c} 4 & 1 & 0 & -12 \\ 1 & 4 & 1 & 12 \\ 0 & 1 & 4 & -12 \end{array}\right]$$

- We need to determine $K_1, K_2, K_3$ (Gauss elimination, LU decomposition)
- Build set of spline 3 segments
- Identify and use the segment corresponding to $\underline{x} \to y(x)$

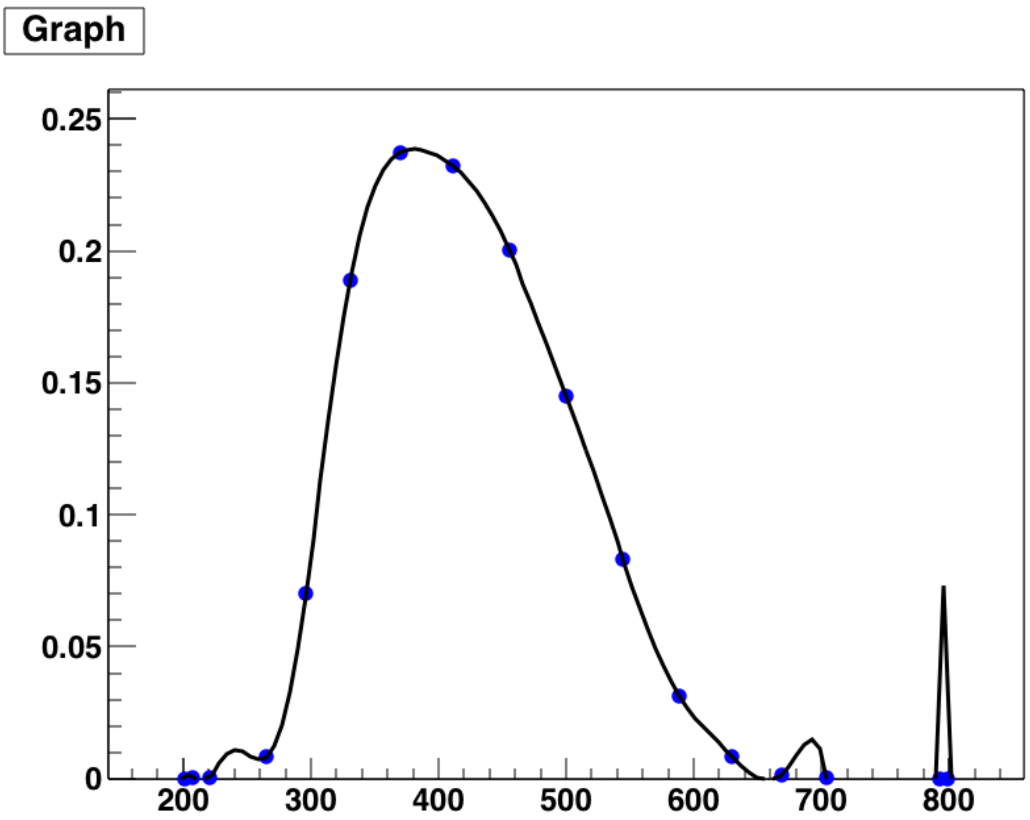





# polynomial

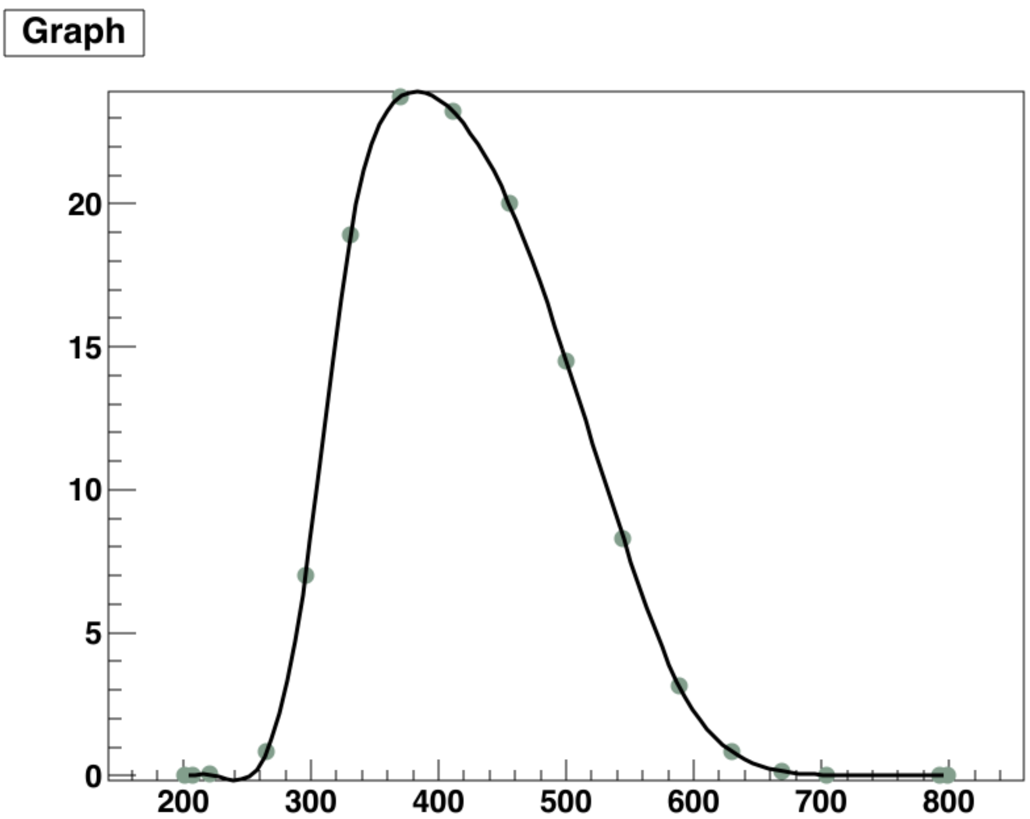


## *polynomial with too many points*






## *the same polynomial with spline3*



61-1

61-2