Computational Physics

Data interpolation and

Data fitting

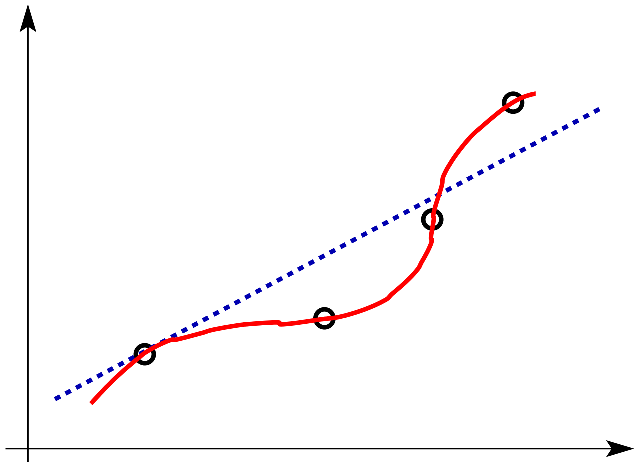
Fernando Barao, Phys Department IST (Lisbon)

Numerical methods

- ✔ System of linear equations
  - Gauss elimination
  - LU decomposition
  - Gauss-Seidel method
- ✔ Interpolation
  - Lagrange interpolation
  - Newton method
  - Neville method
  - Cubic spline
- ✔ Numerical derivatives
  - First derivative $\mathcal{O}(h^2)$, $\mathcal{O}(h^4)$
  - Second derivative $\mathcal{O}(h^2)$, $\mathcal{O}(h^4)$
  - Derivative by interpolation
- ✔ Numerical integration
  - Newton-Cotes: trapezoidal and Simpson rules
  - Gaussian quadrature
- ✔ Monte-Carlo methods

Data interpolation

- Having a set of discrete data points (x_i, y_i), **data interpolation** is the way of getting a continuous description passing through the data points



Lagrange interpolation

- Lagrange interpolation relies on the fact that in a finite interval a function $\mathbf{f}(\mathbf{x})$ can allways be represented by a polynomial $\mathbf{P}(\mathbf{x})$
- **Linear interpolation:** polynomial of **degree one** passing through data points $(\mathbf{x_1}, \mathbf{y_1})$ and $(\mathbf{x_2}, \mathbf{y_2})$

$$\boxed{P(x) = P_0 + P_1 x}$$

System to be solved:

$$\begin{cases} y_1 = & P_0 + P_1 x_1 \\ y_2 = & P_0 + P_1 x_2 \end{cases} \Rightarrow \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \end{pmatrix} \begin{pmatrix} P_0 \\ P_1 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

$$\begin{cases} P_1 = \frac{y_2 - y_1}{x_2 - x_1} \\ P_0 = y_2 - P_1 x_1 \end{cases} \qquad P(x) = P_0 + P_1 x = y_1 \frac{x - x_2}{x_1 - x_2} + y_2 \frac{x - x_1}{x_2 - x_1}$$

Lagrange interpolation (cont.)

✔ **second-degree polynomial interpolation:** polynomial of **degree two** passing through data points $(\mathbf{x_1}, \mathbf{y_1})$, $(\mathbf{x_2}, \mathbf{y_2})$ and $(\mathbf{x_3}, \mathbf{y_3})$

$$P(x) = P_0 + P_1 x + P_2 x^2$$

System to be solved:

$$\begin{cases} y_1 = & P_0 + P_1 x_1 + P_2 x_1^2 \\ y_2 = & P_0 + P_1 x_2 + P_2 x_2^2 \\ y_3 = & P_0 + P_1 x_3 + P_2 x_3^2 \end{cases} \Rightarrow \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{pmatrix} \begin{pmatrix} P_0 \\ P_1 \\ P_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

$$P(x) = y_1 \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} + y_2 \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} + y_3 \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}$$

Lagrange interpolation (cont.)

✔ **n polynomial interpolation:** polynomial of **degree n** passing through $(\mathbf{n+1})$ data points $(\mathbf{x_0}, \mathbf{y_0}), (\mathbf{x_1}, \mathbf{y_1}), \cdots, (\mathbf{x_n}, \mathbf{y_n})$

$$P(x) = P_0 + P_1 x + P_2 x^2 + \cdots + P_n x^n$$

$$\begin{aligned} P_n(x) &= \textstyle\sum_{i=0}^{n} y_i \, \ell_i(x) \\ &= y_0 \, \ell_0(x) + y_1 \, \ell_1(x) + \\ &\quad \cdots + y_n \, \ell_n(x) \end{aligned}$$

$$\ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \qquad (i = 0, 1, 2, \cdots, n)$$

algorithm

```
// n = polynomial degree

// n+1 = nb of data points

// x,y = abcissa and values

   double x[n+1], y[n+1];

// loop on data points (0...n)

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

// we need a second loop for
// the product

      for (...) {

      }

   }
}
```

Interpolation: C++ class scheme

```
----------------------
|  class DataPoints  |
----------------------   class DataPoints {
   / \      / \            public:
    |        |                  ...
    |        |                  virtual double Interpolate(double x);
    |        |                  virtual void Draw();
    |        |                  virtual void Print();
    |        |             protected:
    |        |                  int N; //nb data points
    |        |                  double *x, *y; //x and y values
    |        |           };
    |        |
    |        |
    |        |
    |        |
    |    ----------------------------
    |    |  class LagrangeInterpol  |
    |    ----------------------------   class LagrangeInterpol : public DataPoints {
    |                                     public:
    .                                          ...
    .                                          double Interpolate(double x);
    .                                          void Draw();
 (other interpolation                          void Print();
  classes)                                public:
                                               ? //specific data to class
                                        };
```


Newton method

✔ The Newton method provides a better computational procedure to get an interpolating polynomial of degree n passing through $(n+1)$ data points

$$x_i = x_0, x_1, \cdots, x_n$$
$$y_i = y_0, y_1, \cdots, y_n$$
$$a_i = a_0, a_1, \cdots, a_n$$

$$P(x) = a_0 + a_1(x-x_0) + a_2(x-x_0)(x-x_1) + \cdots + a_n(x-x_0)(x-x_1)\cdots(x-x_{n-1})$$

✔ This polynomial can be written in an efficient computational way:

$$P(x) = a_0 + (x-x_0)\ [a_1 + (x-x_1)\ [a_2 + (x-x_2)\,[\cdots[a_{n-1} + (x-x_{n-1})a_n]\ldots]$$

✔ The coefficients are determined by imposing the polynomial to pass through the data points:

$$(x_0,y_0): \quad y_0 = a_0$$
$$(x_1,y_1): \quad y_1 = a_0 + a_1\,(x_1-x_0)$$
$$(x_2,y_2): \quad y_2 = a_0 + a_1\,(x_2-x_0) + a_2\,(x_2-x_0)(x_2-x_1)$$
$$(x_3,y_3): \quad y_3 = a_0 + a_1\,(x_3-x_0) + a_2\,(x_3-x_0)(x_3-x_1) + a_3\,(x_3-x_0)(x_3-x_1)(x_3-x_2)$$
$$\vdots$$
$$(x_n,y_n): \quad y_n = a_0 + a_1\,(x_n-x_0) + \cdots + a_n\,(x_n+x_0)(x_n-x_1)\cdots(x_n-x_{n-1})$$

Newton method

✔ **Coefficients:**

$$\begin{aligned} a_0 &= y_0 \\ a_1 &= \frac{y_1-y_0}{x_1-x_0} \equiv \nabla y_1 \\ a_2 &= \frac{1}{x_2-x_1}\left(\frac{y_2-y_0}{x_2-x_0} - \frac{y_1-y_0}{x_1-x_0}\right) \\ &= \frac{\nabla y_2 - \nabla y_1}{x_2-x_1} \equiv \nabla^2 y_2 \\ a_3 &= \nabla^3 y_3 \\ a_4 &= \nabla^4 y_4 \\ \vdots &= \vdots \\ a_n &= \nabla^n y_n \end{aligned}$$

	0th	1st	2nd	3rd	4th
x_0	y_0				
x_1	y_1	∇y_1			
x_2	y_2	∇y_2	$\nabla^2 y_2$		
x_3	y_3	∇y_3	$\nabla^2 y_3$	$\nabla^3 y_3$	
x_4	y_4	∇y_4	$\nabla^2 y_4$	$\nabla^3 y_4$	$\nabla^4 y_4$

The diagonal terms of the table are the coefficients of the polynomial

Divided diferences:

$$\begin{aligned} \nabla y_i &= \frac{y_i-y_0}{x_i-x_0} & (i = 1, 2, .., n) \\ \nabla^2 y_i &= \frac{\nabla y_i - \nabla y_1}{x_i-x_1} & (i = 2, 3, .., n) \\ \nabla^3 y_i &= \frac{\nabla^2 y_i - \nabla^2 y_2}{x_i-x_2} & (i = 3, 4, .., n) \\ \vdots & \quad \vdots \\ \nabla^n y_n &= \frac{\nabla^{n-1} y_n - \nabla^{n-1} y_{n-1}}{x_n - x_{n-1}} \end{aligned}$$

Newton method: interpolating polynomial

✔ Suppose four data points $\Rightarrow$ $n = 3$ polynomial degree
$(x_0, y_0) \cdots (x_3, y_3)$

✔ The polynomial

$$\begin{aligned} P_3(x) &= a_0 + (x-x_0)\,a_1 + (x-x_0)(x-x_1)\,a_2 + (x-x_0)(x-x_1)(x-x_2)\,a_3 \\ &= a_0 + (x-x_0)\left[a_1 + (x-x_1)\left[a_2 + (x-x_2)\,a_3\right]\right] \end{aligned}$$

Recurrence relations to evaluate polynomial:

$P_0(x) = a_3$

$P_1(x) = a_2 + (x - x_2)P_0(x)$

$P_2(x) = a_1 + (x - x_1)P_1(x)$

$P_3(x) = a_0 + (x - x_0)P_2(x)$

$$\boxed{P_k(x) = a_{n-k} + (x - x_{n-k})P_{k-1}(x) \quad (k = 1, 2, ..., n)}$$

Computing the interpolated value at x with the polynomial computed in a recursive way:

$P_0(x) = a_n$

$P_1(x) = a_{n-1} + (x - x_{n-1})P_0(x)$

$P_2(x) = a_{n-2} + (x - x_{n-2})P_1(x)$

$\vdots$

$P_k(x) = a_{n-k} + (x - x_{n-k})P_{k-1}(x) \quad (k = 1, 2, ..., n)$

Newton method: algorithm

Coefficients:

```
// degree n polynomial
// n+1 data points
//
// For computing the coefficients
// we can use a one-dimensional
// array a[n+1]
//
// X[n+1] array, contains x data values

 1) make array a[n+1];

 2) copy contents of Y[] data to array a[]

 3) compute divided differences and
    store them in the one dimensional
    array a[]

    loop on k=1; k<n+1; k++

       loop on i=k; i<n+1; i++

          a[i] = (a[i] - a[k-1]) /
                 (X[i] - X[k-1])
```

Polynomial:

```
// degree n polynomial
// n+1 data points
//
// For computing the polynomial at a point x
// we use the recurrence existing
// after factorizing the polynomial
//
// We assume having already the
// coefficients
// computed in the array a[n+1]
//
// X[n+1] array, contains x data values
//

 1) init the last polynomial P

    P = a[n];

 2) loop on k=1; k<n+1; k++

       P = a[n-k] + (x - X[n-k])*P
```


Neville method

- ✔ The Neville algorithm is still better by computing standards for finding the n degree polynomial because does not require a computation in two steps
- ✔ It uses linear interpolations between successive iterations: one point needed at $0th$ order, two points at $1st$ order, three points at $2nd$ order, ..., $n+1$ points at nth order

0th order: $P_0[x_0] = y_0, \cdots P_n[x_n] = y_n$

1st order (linear): $P_1[x_0, x_1] = C_0 + C_1 x = \frac{y_1(x-x_0)-y_0(x-x_1)}{x_1-x_0} = \frac{(x-x_0)\,P[x_1]-(x-x_1)\,P[x_0]}{x_1-x_0}$

2nd order: $P_2[x_0, x_1, x_2] = \frac{(x-x_2)\,P[x_0,x_1]-(x-x_0)\,P[x_1,x_2]}{x_0-x_2}$

3rd order: $P_3[x_0, x_1, x_2, x_3] = \frac{(x-x_3)\,P[x_0,x_1,x_2]-(x-x_0)\,P[x_1,x_2,x_3]}{x_0-x_3}$

... ...

x values	0th order	1st order	2nd order	3rd order	...order
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	$P_0(x_4) = y_4$	$P_1[x_3, x_4]$	$P_2[x_2, x_3, x_4]$	$P_3[x_1, x_2, x_3, x_4]$	
...	...	...	...		
x_n	$P_0(x_n) = y_n$	$P_1[x_{n-1}, x_n]$	$P_2[x_{n-2}, x_{n-1}, x_n]$	$P_3[x_{n_3}, x_{n-2}, x_{n-1}, x_n]$	

Neville method: algorithm?

1) We can work with only one array (1-dim) y[] containing the 0th order polynomials

2) loop on the order of the polynomials:

i=0, i<n+1

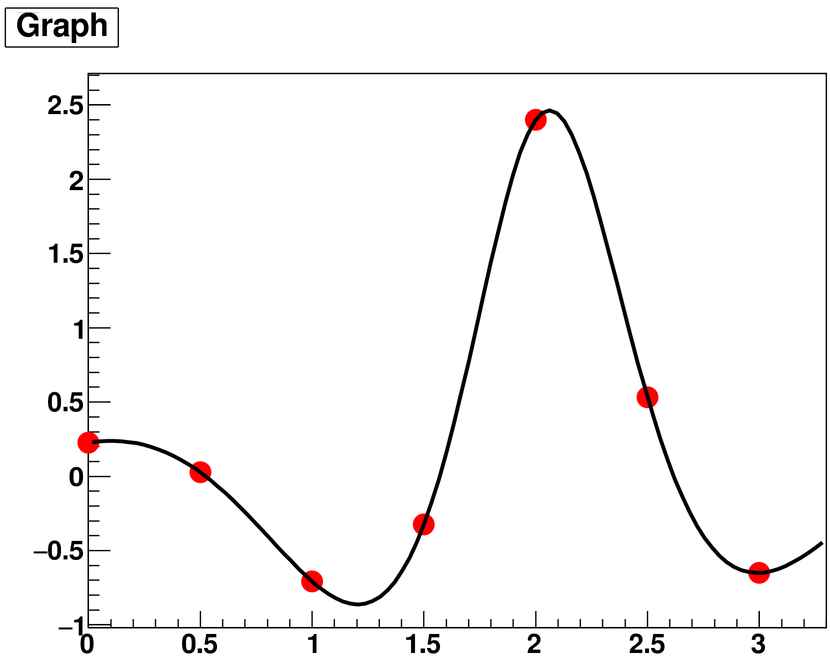
3) loop on every column to compute the different polynomials

y[] array will be rewritten with new values

4) the interpolant calculated at the coordinate x, corresponds to the last value

interpolation example

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



DataPoints base class

```
#ifndef __DataPoints__
#define __DataPoints__

#include "cFCgraphics.h"

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;
};
#endif
```


DataPoints class code

```
#include "DataPoints.h"
#include "TGraph.h"

DataPoints::DataPoints() {
  N = 0;
  x = NULL;
  y = NULL;
}

DataPoints::DataPoints(int fN, double* fx, double* fy) : N(fN) {
  x = new double[N];
  y = new double[N];
  for (int i=0; i<N; i++) {
    x[i] = fx[i];
    y[i] = fy[i];
  }
}

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

DataPoints class code (cont.)

```
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();
}
```


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() {;}

  double Interpolate(double x);
  void Draw();

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

 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 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	$(x_0 - x_2)^{-1}\ [(x - x_2)y_0 - (x - x_0)y_1]$

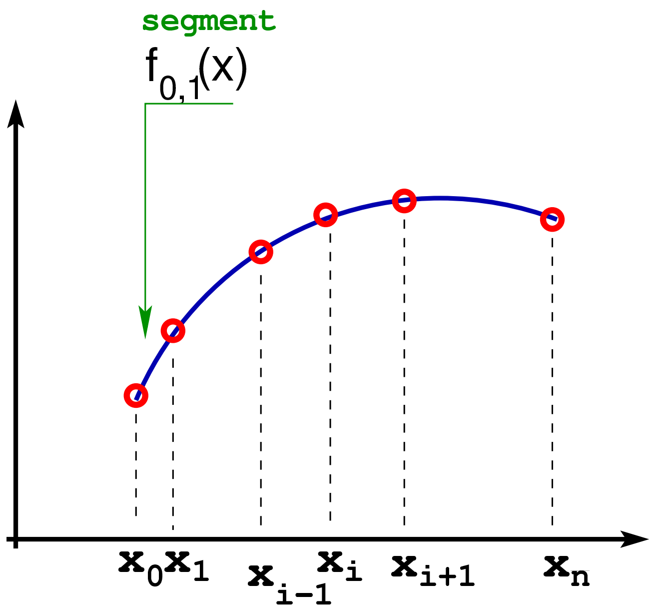
the interpolated value at x is the last computed value and is stored on $y[0]$

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:

$[x_0, x_1]$

$$\begin{aligned} f_0(x) &= a_0 + b_0(x - x_0) + c_0(x - x_0)^2 + b_0(x - x_0)^3 \\ f_0(x_0) &= y_0 \\ f_0(x_1) &= y_1 \\ f'_0(x_0) &= y'_0 \quad \text{numerically} \\ f''_0(x_0) &= 0 \end{aligned}$$

$[x_1, x_2]$

$$\begin{aligned} f_1(x) &= a_1 + b_1(x - x_1) + c_1(x - x_1)^2 + b_1(x - x_1)^3 \\ f_1(x_1) &= y_1 \\ f_1(x_2) &= y_2 \\ f'_1(x_1) &= f'_0(x_1) \\ f''_1(x_1) &= f''_0(x_1) \end{aligned}$$

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:

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

Cubic spline method (cont.)

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

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

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

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

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

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

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.)

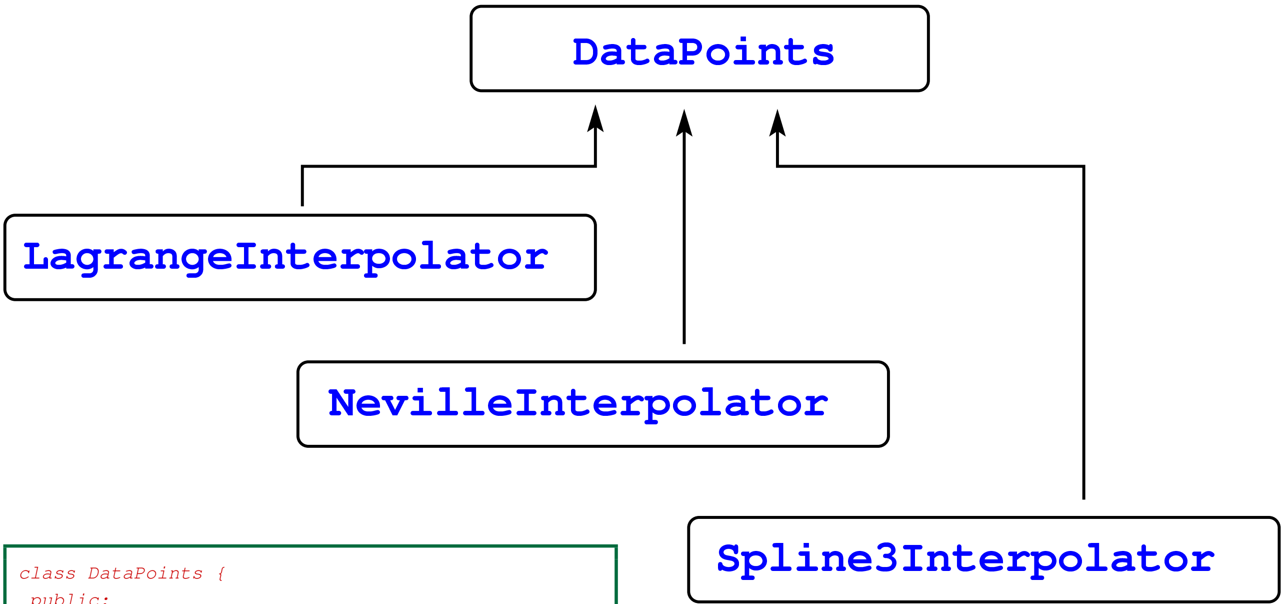
The set of equations to solve:

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

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