COMPUTATIONAL PHYSICS

Numerical methods

System of linear equations

Interpolation

COMPUTATIONAL PHYSICS

Numerical methods

System of linear equations

- ✓ *Gaussian elimination*
- ✓ *LU decomposition*
- ✓ *Iterative methods*

Systems of Linear equations

- Solving systems of linear equations is ubiquitous in Science and when developing numerical algorithms it's almost inevitable !

- There are fundamentally 2 types:

  - ***Direct*** – Algorithms that, in the absence of round-off errors, find the exact solution within a finite sequence of steps
    - **Gauss Elimination** (with/without pivoting)
    - **LU decomposition** (includes Doolittle algorithm and Thomas algorithm for Banded matrices)

  - ***Iterative*** – Starting from an initial guess, the solution is sought for through a sequence of iterations (useful when using very large sparse matrices)
    - **Gauss-Seidel**

Systems of Linear equations - Basics

- Solving the system of linear equation in matrix form **A**x=b where

$$A_{11}x_1 + A_{12}x_2 + \cdots + A_{1n}x_n = b_1$$
$$A_{21}x_1 + A_{22}x_2 + \cdots + A_{2n}x_n = b_2$$
$$\cdots$$
$$A_{n1}x_1 + A_{n2}x_2 + \cdots + A_{nn}x_n = b_n$$

Real valued coefficients – A_{ij}
Real value constants – b_j
Real value unknowns – x_i

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Though apparently simple…it can become less obvious depending on the coefficient matrix (***A***) behavior...

Augmented Matrix $\left(A|b\right)$

Matrix conditioning

- Suppose we were to change slightly the constants vector ***b*** i.e. ***b+Δb***.
- How does the solution ***x*** change ? Is $\|\Delta x\| / \|\Delta b\| << 1$?

$$A = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1+10^{-c} & 1-10^{-c} \end{bmatrix} \qquad A^{-1} = \begin{bmatrix} 1-10^{c} & 10^{c} \\ 1+10^{c} & -10^{c} \end{bmatrix}$$

- If b=[1 1], solution is x=[1 1]. But if we change ***b*** to ***b+Δb***……

$$A\Delta x = \Delta b \Leftrightarrow \Delta x = A^{-1}\Delta b = \begin{bmatrix} \Delta b_1 - 10^{c}(\Delta b_1 - \Delta b_2) \\ \Delta b_1 + 10^{c}(\Delta b_1 - \Delta b_2) \end{bmatrix}$$

- As c>>1, the change in solution ***x*** becomes increasingly severe !!!

Matrix conditioning – further insight

- As it turns out, $\frac{\|\Delta x\|}{\|x\|} \leq \underbrace{\|A\|\|A^{-1}\|}_{\kappa(A)} \frac{\|\Delta b\|}{\|b\|}$

  $\kappa(A)$ - *Condition number of A*

- In practical terms, if $|A| << \|A\|$ (κ>>1) then the matrix is ill-conditioned and one in bound to have imprecision in our system solution.

- $|A| - \text{determinant}$ and possible norms $\|A\|_\infty = \max_{1\leq i\leq n}\left(\sum_{j=1}^{n}|A_{ij}|\right)$

$$\|A\|_F = \left(\sum_{i=1}^{n}\sum_{j=1}^{n}A_{ij}^2\right)^{1/2}$$

- As the determinant of the matrix goes to 0 (singular matrix), the condition number κ(A) becomes challenging high.

Direct methods to solve system of equations

- Start be recalling some useful properties of systems of linear equations and matrices (coeff. matrix ***A***):
  - ✓ When swapping the order of equations of the system, the determinant of ***A*** changes sign but the solution remains the same.
  - ✓ When multiplying one equation by a non-zero constant (c), |**A**| →c|**A**| and the solution remains the same.
  - ✓ When adding a multiple of one equation to another equation, both the solution and |**A**| remain unchanged.
  - ✓ Any square matrix admits a decomposition PA=LU where P is a permutation matrix (reordering of A-rows) and L and U are

$$L = \begin{pmatrix} L_{11} & 0 & \cdots & 0 \\ L_{21} & L_{22} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ L_{n1} & L_{n2} & \cdots & L_{nn} \end{pmatrix} \qquad U = \begin{pmatrix} U_{11} & U_{12} & \cdots & U_{1n} \\ 0 & U_{22} & \cdots & U_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & U_{nn} \end{pmatrix}$$

Gauss elimination method

- Most obvious method but has a ***big caveat***: pivoting with reordering is necessary to avoid propagation of huge number during the elimination stage….(see *later examples*)....

Elimination stage

$$Ax = b \quad \rightarrow \quad Ux = c$$

→ At each step, multiply a pivot row (the one to retain the diagonal term of U) by a constant and subtract to each one of the rows below i.e.

$$Row_j \rightarrow Row_j - \beta_{ij} Row_i$$

→ Procedure ends once A → U

Back stubstitution stage

$$Ux = c \quad \rightarrow \quad \mathrm{I}x = d$$

→ Once we get to $\left(U \middle| c\right)$ form, trivially cycle to get x_i starting from i=n up to i=1 !

Row-n	$U_{nn} x_n = c_n \Leftrightarrow x_n = c_n / U_{nn}$
Row-k k=n-1..1	$U_{kk} x_k + \ldots + U_{kn} x_n = c_k$ $\Leftrightarrow x_k = 1/U_{kk} \left(c_k - \sum_{i=k+1}^{n} x_i U_{ki} \right)$

Gauss elimination stage - detail

$$\left(\begin{array}{ccc|c} \boxed{4} & 2 & 1 & 7 \\ 2 & -1 & 3 & -3 \\ 1 & -2 & -3 & 0 \end{array}\right) \quad \begin{array}{l} row_2 - row_1 \times (2/\boxed{4}) \\ row_3 - row_1 \times 1/\boxed{4} \end{array} \quad \left(\begin{array}{ccc|c} 4 & 2 & 1 & 7 \\ 0 & -2 & 5/2 & -13/2 \\ 0 & -5/2 & -13/4 & -7/4 \end{array}\right)$$

$$\left(\begin{array}{ccc|c} 4 & 2 & 1 & 5 \\ 0 & \boxed{-2} & 5/2 & -13/2 \\ 0 & -5/2 & -13/4 & -7/4 \end{array}\right) \quad row_3 - row_2 \times 5/4 \quad \left(\begin{array}{ccc|c} 4 & 2 & 1 & 5 \\ 0 & -2 & 5/2 & -13/2 \\ 0 & 0 & -51/8 & 51/8 \end{array}\right)$$

→ At each step, as long as the pivot row has a dominant value on the column to eliminate below, we are safe i.e. ***no division by "close to zero" or zero***

→ If we ever come across such a case, ***partial pivoting*** to the rescue !

→ At each step of the elimination stage, find first the largest (***in relative magnitude in it's row***) matrix element in the column of interest and promote the associated row to pivot row.

Gauss elimination stage – partial pivoting

$$\left(\begin{array}{ccc|c} \delta & 1 & 1 & 0 \\ 1 & -1 & 1 & 1 \\ 2 & 1 & 0 & -1 \end{array}\right) \longrightarrow \left(\begin{array}{ccc|c} \delta & 1 & 1 & 0 \\ 0 & -1-1/\delta & 1-1/\delta & 1 \\ 0 & 1-2/\delta & -2/\delta & -1 \end{array}\right)$$

Contradicting result if
δ<<1

→ A quick scan over the first column prompts for swapping first and last rows

$$\left(\begin{array}{ccc|c} 2 & 1 & 0 & -1 \\ 1 & -1 & 1 & 1 \\ \delta & 1 & 1 & 0 \end{array}\right) \longrightarrow \left(\begin{array}{ccc|c} 2 & 1 & 0 & -1 \\ 0 & -3/2 & 1 & 3/2 \\ 0 & 1-\delta/2 & 1 & \delta/2 \end{array}\right)$$

→ No problem at all for **δ<<1** or **δ=0**

Gauss elimination stage – partial pivoting

- A possible algorithm for the gauss elimination stage would be:

```
for (int row=0; row < nrows-1; row++) {

  // as we cycle in row, we need to decide if A[row:nrows,row] is an adequate pivot or not...
  int drow=A.GetColMax(row);  // get index of row (drow) where A[drow,row] has the
                              // highest relative magnitude in it's row
  A.swapRows(row,drow);  //promote that row to contain the pivot
  vecb.swap(row,drow);   //likewise for vector b.

  ....check if system is not undetermined.....

// Now we add suitable multiples of the (new) "row"th row to the ones below
// to eliminate terms in that column....
  for (int i=row+1; i < nrows; i++) {
    lambda=A[i][row]/A[row][row];  //scaling factor
    A[i]=A[i]-A[row]*lambda;  //calculate the new row-i
    vecb[i]=vecb[i]-vecb[row]*lambda;  //calculate new vector v index-i
  }

}
```

Class scheme suggested - Vec

```
#ifndef H_FCVEC_H
#define H_FCVEC_H
#include <iostream>
Class Vec {
  private:
    int N; //number of elements
    double * entries; // pointer to array of doubles
public:
  Vec(int n=1, double d=0.); //constructor with num. el. and value
  Vec(int n, double * ptr); //constructor with num. el. and ptr array
  ~Vec(); //destructor
  Vec(const Vec &); //copy constructor…useful where you least expect
  void SetEntries (int n, double* ptr); //set entries
  void Print(); //print the vector content
  void swap(int n, int m); //swap elements of order n and m in vector
  int size () const; //return size of vector
  double dot (const Vec & obj); //scalar product with another vector
  Vec & operator=(const Vec & obj); //operator=
  Vec operator+(const Vec & obj); //operator+
  Vec & operator+=(const Vec & obj); //operator+=
  Vec operator-(const Vec & obj); //operator-
  Vec & operator-=(const Vec & obj); //operator-=
  double& operator[] (int x); //operator[]
  double operator[] (int x) const; //operator[] when the Vec is a const
  Vec operator*(const Vec & obj); //operator*
  Vec operator*(const double & scalar); //operator* a scalar
};
#endif
```

Notes

- *Const* functions are required when the calling object is itself a *Const*
- *Const* function cannot modify non-static data members nor call other *non const* member functions.
- *Non const* objects can call *const* member function though…

LU decomposition

- Every square (mostly invertible) matrix can be written as ***PA=LU*** where P is a permutation matrix (reordering of A-rows) → *LU decomposition*.
- Useful to solve linear systems since: ***Ax=b*** → LUx=c → **Ly=c** and **Ux=y**
- The permutation matrix **P** is easily understood when we consider how close **L** and **U** are to the gauss elimination of **A**…!

- There are several LU decompositions, depending on the particular choices for the *diagonal terms* of ***L*** or ***U*** i.e.

Doolittle: Main diagonal of ***L*** set to 1 → $L_{ii}=1$ i=1,2,…,n

Crout: Main diagonal of ***U*** set to 1 → $U_{ii}=1$ i=1,2,…,n

Choleski: $U=L^*$ (conjugate transpose), valid if A is Hermitian positive definite

LU decomposition – Doolittle algorithm

- Since L has 1 in main diagonal, one easily derives that (*no pivoting case*)

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ L_{21}U_{11} & L_{21}U_{12} + U_{22} & L_{21}U_{13} + U_{23} \\ L_{31}U_{11} & L_{31}U_{12} + L_{32}U_{22} & L_{31}U_{13} + L_{32}U_{23} + U_{33} \end{pmatrix}$$

where

$$L = \begin{pmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{pmatrix}$$

$$U = \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix}$$

✓ Thus, we easily learn that, in sequence, per each line of U we derive, one can derive a column of L !

The general algorithm is:

```
for i = 1:n
   for j = i:n
      L_ik U_kj = A_ij gives row-i of U
   end
   for j = i+1:n
      L_jk U_ki = A_ji gives column-i of L
   end
end
```

LU decomposition – Gauss el. like algorithm

- Alternatively, one can easily perform Gauss elimination on matrix A and interpret the elements of L accordingly.....

$$A = \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ L_{21}U_{11} & L_{21}U_{12} + U_{22} & L_{21}U_{13} + U_{23} \\ L_{31}U_{11} & L_{31}U_{12} + L_{32}U_{22} & L_{31}U_{13} + L_{32}U_{23} + U_{33} \end{pmatrix} \begin{matrix} \\ row_2 - row_1 \times \boxed{L_{21}} \\ row_3 - row_1 \times \boxed{L_{31}} \end{matrix}$$

$$\rightarrow \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & L_{32}U_{22} & L_{32}U_{23} + U_{33} \end{pmatrix} \begin{matrix} \\ \\ row_3 - row_2 \times \boxed{L_{32}} \end{matrix}$$

✓ Indeed the elements of L-matrix are just the multipliers used in Gauss elimination !

✓ But ***mind partial pivoting*** → elements of L-matrix ***also swap location*** !

LU decomposition – Doolittle algorithm

- A practical information: the matrices ***L*** and ***U*** can actually be stored in a single matrix e.g. ***Z*** since we know the diagonal of ***L*** has 1's !

$$Z = \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ L_{21} & U_{22} & U_{23} \\ L_{31} & L_{32} & U_{33} \end{pmatrix}$$

- Code snippet...

Ax=b

```
LU=LUdecomposition_Doolittle(A);
FCmatrixFull L(LU[0]); //use explicit matrix L
FCmatrixFull U(LU[1]); //use explicit matrix u
delete[] LU;

Vec y(b.size(),0.0); //initialize to 0
// Ly=b
for (int i = 0; i < L.Get_nRows(); i++) {
   y[i] = (b[i]-L[i].dot(y)) / 1.;  //L[i][i]=1 by definition !!!
  }

Vec x(b.size(),0.0); //initialize to 0
//backsubstitution  Ux=y
for (int i = U.Get_nRows()-1; i >= 0; i--) {
   x[i] = (y[i]-U[i].dot(x)) / U[i][i];
}
return x;
```

Class scheme suggested

Vec

- N and entries
- All methods we can do with 1D vectors…

FCmatrix

- Mother of all matrices...x

FCmatrixFull

- A full matrix with all NxN entries filled

FCmatrixBanded

- Just 3 diagonals
- “Almost” all methods of FCmatrixFull...(?)

EqSolver

- GaussianEliminationSolver
- LUdecompositionSolver
- TridiagonalSolver

Class scheme suggested - FCmatrix

```
#ifndef H_FCmatrix_H
#define H_FCmatrix_H
#include <vector>
#include "Vec.h”
class FCmatrix {
 public:
   //constructors
   FCmatrix();
   FCmatrix(double** fM, int fm, int fn); //matrix fm x fn
   FCmatrix(double* fM, int fm, int fn);
   FCmatrix(vector<Vec>);

   // operators
   virtual Vec& operator[] (int) = 0;
   // methods
   virtual int Get_nRows() const = 0; //number of rows of M
   virtual int Get_nCols() const = 0; //number of columns of M
   virtual double Determinant() const = 0;
   virtual Vec Get_ld() const = 0; //get the lower diagonal
   virtual Vec Get_md() const = 0; //get the main diagonal
   virtual Vec Get_ud() const = 0; //get the upper diagonal
   virtual void Print() const;
  protected:
   vector<Vec> M;
  string classname;
};
#endif
```

Notes

- FCmatrix is just a “template” → can easily be an abstract class
- There are surely methods that are less useful for some derived classes e.g. Get_Row(int) for a Banded matrix…
- Print() is defined but surely can/should be overloaded

Class scheme suggested - FCmatrixFull

```
class FCmatrixFull : public FCmatrix {
 public:
   // constructors
   FCmatrixFull();
   FCmatrixFull(double** fM, int fm, int fn); //matrix fm x fn
   FCmatrixFull(double* fM, int fm, int fn);
   FCmatrixFull(vector<Vec>);

   // copy constructor
   FCmatrixFull(const FCmatrixFull&);

   // operators
   FCmatrixFull operator=(const FCmatrix &); // equal 2 matrices of any kind
   FCmatrixFull operator+(const FCmatrix &) const; // add 2 matrices of any kind
   FCmatrixFull operator-(const FCmatrix &) const; // sub 2 matrices of any kind
   FCmatrixFull operator*(const FCmatrix &) const; // mul 2 matrices of any kind
   FCmatrixFull operator*(double lambda) const; // mul matrix of any kind by scalar
   Vec operator*(const Vec &) const; // mul matrix by Vec

   // virtual inherited
   int Get_nRows() const; //number of rows of M
…
   Vec& operator[] (int);
   int GetRowMax(int i=0) const;
   int GetColMax(int j=0) const;
…
   void swapRows(int,int);
};
#endif
```

Systems of equations – Banded matrices

- Frequently, one is faced with systems of equations where the matrix is "banded" i.e. main diagonal + some upper & lower diagonals non null.

$$\begin{pmatrix} A_{11} & A_{12} & 0 & & & \\ A_{21} & A_{22} & A_{23} & 0 & & \\ 0 & A_{32} & A_{33} & A_{34} & \ddots & \\ & \ddots & \ddots & \ddots & \ddots & 0 \\ & & 0 & A_{89} & A_{88} & A_{89} \\ & & & 0 & A_{98} & A_{99} \end{pmatrix}$$

Immediate note: We can just store the diagonals and save memory and computational time !

$A_{i+1,i}$ - [a]
$A_{i,i}$ - [b]
$A_{i,i+1}$ - [c]

Second note: The same algorithms as before e.g. Doolittle LU, can be used though a particular one emerges → ***Thomas algorithm***

Banded matrices - Thomas algorithm

- The basic idea is to transform the coefficient matrix (and constant vector) in some more *amenable* form…

$$\left(\begin{array}{cccccc|c} b_1 & c_1 & 0 & 0 & . & 0 & d_1 \\ a_2 & b_2 & c_2 & 0 & . & 0 & d_2 \\ 0 & a_3 & b_3 & c_3 & . & 0 & d_3 \\ 0 & . & . & . & . & . & . \\ . & . & . & a_{n-1} & b_{n-1} & c_{n-1} & d_{n-1} \\ 0 & 0 & 0 & . & a_n & b_n & d_n \end{array}\right)$$

$$c_i^* = \begin{cases} c_1 / b_1 & \text{i=1} \\ c_i / \left(b_i - c_{i-1}^* a_i\right) & \text{i=2,...,n-1} \end{cases}$$

$$d_i^* = \begin{cases} d_1 / b_1 & \text{i=1} \\ \left(d_i - d_{i-1}^* a_i\right) / \left(b_i - c_{i-1}^* a_i\right) & \text{i=2,...,n} \end{cases}$$

$$\left(\begin{array}{cccccc|c} 1 & c_1^* & 0 & 0 & . & 0 & d_1^* \\ 0 & 1 & c_2^* & 0 & . & 0 & d_2^* \\ 0 & 0 & 1 & c_3^* & . & 0 & d_3^* \\ 0 & . & . & . & . & . & . \\ . & . & . & 0 & 1 & c_{n-1}^* & d_{n-1}^* \\ 0 & 0 & 0 & . & 0 & 1 & d_n^* \end{array}\right)$$

$$x_n = d_n^*$$

$$x_i = d_i^* - x_{i+1} c_i^* \qquad \text{i=n-1,...,2,1}$$

Iterative methods for system of equations

- In iterative methods, rather than getting the exact solution (round-off errors aside), an approximate solution (***x****) is sought for that minimises the error (real value) $\left\| Ax^* - b \right\|$.
- This real would be exactly 0 for the exact solution. Our hope it that, by iterating (*index k*) as many times we need, a certain precision is met i.e.

$$\left\| x^{(k)} - x^{(k-1)} \right\| \leq \varepsilon$$

- However, ***convergence*** is ***only ensured*** if the matrix A is ***diagonally dominant***

Definition

A $n \times n$ matrix **A** = (*Aij*) is **strictly diagonally dominant** if

for each $1 \leq \mathrm{i} \leq \mathrm{n}, \left| A_{ii} \right| > \sum_{j \neq i} \left| A_{ij} \right|$

- Under such conditions, convergence will ***always be met*** and the number of iterations depends on "*how good*" our initial guess is...

Iterative methods - Jacobi method

- Writing the system of equations ***A***x=b as

$$\sum_{j=1}^{n} A_{ij} x_j = b_i \quad \text{i=1,2,...,n}$$

- Isolating the diagonal term $$A_{ii} x_i = b_i - \sum_{\substack{j=1 \\ (j \neq i)}}^{n} A_{ij} x_j \Leftrightarrow x_i = \frac{1}{A_{ii}} \left(b_i - \sum_{\substack{j=1 \\ (j \neq i)}}^{n} A_{ij} x_j \right)$$

- At every iteration-k on does $$x_i^{(k)} = \frac{1}{A_{ii}} \left(b_i - \sum_{\substack{j=1 \\ (j \neq i)}}^{n} A_{ij} x_j^{(k-1)} \right)$$

To test

$$\boxed{\left\| x^{(k)} - x^{(k-1)} \right\| \leq \varepsilon}$$

- In matrix form: A=D+L+U → $$\boxed{x^{(k)} = D^{-1} \left(b - (L+U) x^{(k-1)} \right)}$$

Iterative methods – Gauss-Seidel method

- Similar to Jacobi method with slight difference: as soon as new estimates for x_i become available during the algorithm they are immediately used !

$$x_i^{(k)} = \frac{1}{A_{ii}}\left(b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} A_{ij}x_j^{(k-1)}\right)$$

- Example:

$$\begin{pmatrix} 5 & -1 & 2 \\ 2 & 7 & 1 \\ 2 & 2 & 6 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ 5 \end{pmatrix} \longrightarrow \begin{aligned} x_1^{(k)} &= \left(4 + x_2^{(k-1)} - 2x_3^{(k-1)}\right)/5 \\ x_2^{(k)} &= \left(1 - 2x_1^{(k)} - x_3^{(k-1)}\right)/7 \\ x_3^{(k)} &= \left(5 - 2x_1^{(k)} - 2x_2^{(k)}\right)/6 \end{aligned}$$

- In matrix form: A=D+L+U → $\boxed{x^{(k)} = (L+D)^{-1}\left(b - Ux^{(k-1)}\right)}$

Class scheme suggested - EqSolver

```
#include "FCmatrixFull.h"
#include "FCmatrixBanded.h"
using namespace std;
class EqSolver {

public:
  EqSolver();
  EqSolver(const FCmatrixFull&, const Vec&); // matriz M e
vector de constantes
  EqSolver(const FCmatrixBanded&, const Vec&); // matriz
tridiagonal M e vector de constantes

// set
void SetConstants(const Vec&);
void SetMatrix(const FCmatrixFull&);
void SetMatrix(const FCmatrixBanded&);

Vec GaussEliminationSolver();
Vec LUdecompositionSolver();
Vec TridiagonalSolver();
Vec JacobiSolver(double tol=1.E-6);
…
```

```
…
private:
/* return triangular matrix and changed vector of constants */
void GaussElimination(FCmatrixFull&, Vec&);
//decomposição LU com |L|=1
void LUdecomposition(FCmatrixFull&, vector<int> & index);
Vec TridiagonalThomas(FCmatrixBanded &, Vec &);

FCmatrixFull M; //matriz de coeffs
FCmatrixBanded Band; //objecto com bandas
Vec b; //vector de constantes

bool isSolved;
bool isLUSolved;

bool isMatrix; //full matrix set ?
bool isBand; //banded matrix set ?
bool isB; //vector b set ?

};
```

Class scheme suggested - JacobiSolver

```
//Jacobi iteration
Vec EqSolver::JacobiSolver() {
// linear system of m unknowns
  int m;
  m=b.size();
  Vec x(m); //full of 0
  Vec x_last(m); //stores last iteration
  bool btol = false;
  int it = 0.;
  double eps = 1.E-6; //tolerance
  while (!btol && (it++ < 1000)) {
    x_last= x;
    for (int i=0; i<m; i++) {
      x[i] = 0.;
      for (int j=0; j<m; j++)
        if (i != j) x[i] += -M[i][j]*x_last[j];
      x[i] += b[i];
      x[i] /= M[i][i];
      if (fabs(x[i]-x_last[i]) < eps) btol = true;
      else btol = false;
    }
    it++;
  }
  return x;
}
```

```
int main() {
 double matD[][3] = {{7.,-2.,4.},{-2.,5.,3.},{-1.,4.,8.}};
 vector<Vec> mat;
 Vec tmp; //hold each line of matrix...
 //copy rows as arrays into Vecs
 for (int i=0;i<3;++i) {
   tmp.SetEntries(3,matD[i]);
   mat.push_back(tmp);
 }
 cout << "Assigning MD matrix..." << endl;
 FCmatrixFull D(mat);
 Vec b(3,0.);b[0]=16.;b[1]=0.;b[2]=-1.;

 EqSolver dudu(D,b);
 Vec result;
 result=dudu.JacobiSolver();
 cout << "Solution:[" << flush;
 for (int i=0;i<result.size()-1;i++) {
   cout << result[i] << "," << flush;
 }
 cout << result[result.size()-1] << "]" << endl;
 cout<<"Exiting main..."<<endl;
 return 0;
}
```

COMPUTATIONAL PHYSICS

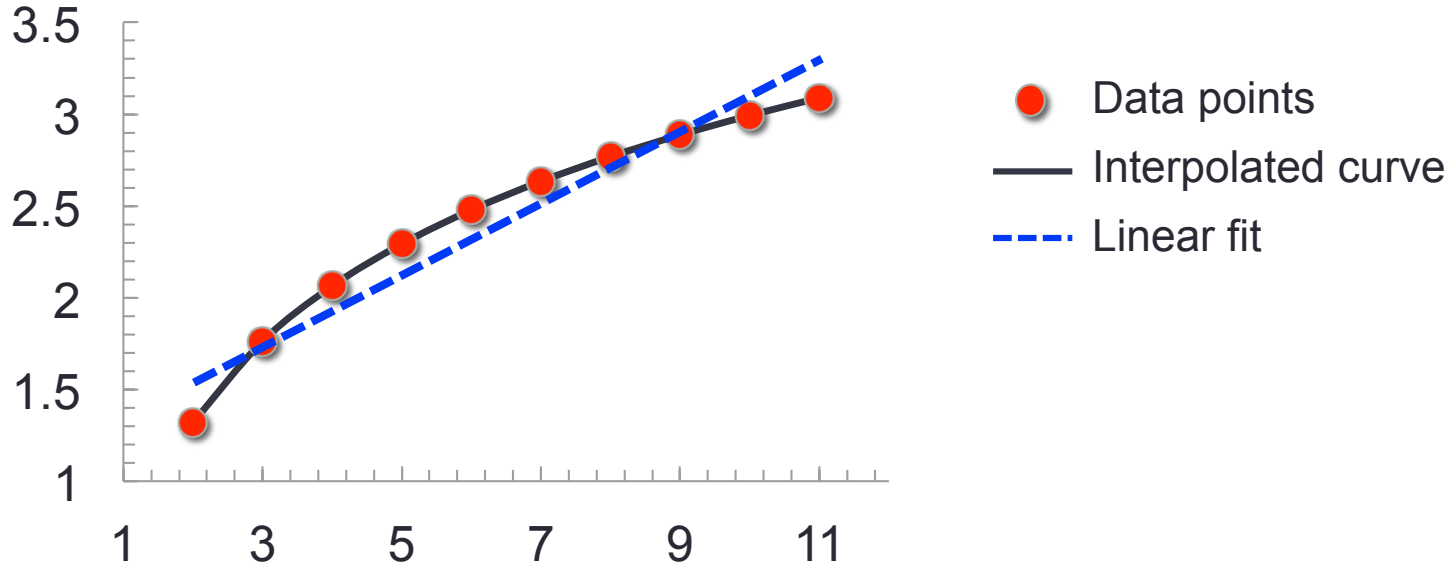
Numerical methods

Interpolation

- ✓ *Lagrange*
- ✓ *Newton method*
- ✓ *Neville method*
- ✓ *Cubic spline*

Interpolation *vs* fitting

- On **data interpolation**, the data points are assumed to be known *exactly* and the curve performing the interpolation (*of some type*) passes through each of the data points *(nodes)*.
- **Data fitting**, on the other hand, assumes data to have some error and the curve performing the fitting (of some type) passes as near as possible through the data points (e.g. in a least squared sense)



- ***Same goal though***: get a functional form to obtain $f(x_i)$ with $x_i \neq$ {data points}

Lagrange interpolation

- ***Fundamental idea***: there is an unique ***polynomial of degree-N*** that goes through a sequence of ***N+1 data points***.

- Example: ***linear interpolation*** → degree-1 and data points (x_1,y_1), (x_2,y_2)

$$\boxed{P(x) = a_1 + a_2 x} \longrightarrow \begin{cases} y_1 = a_1 + a_2 x_1 \\ y_2 = a_1 + a_2 x_2 \end{cases} \longrightarrow \begin{aligned} a_1 &= \left(y_1 x_2 - y_2 x_1\right) / \left(x_2 - x_1\right) \\ a_2 &= \left(y_2 - y_1\right) / \left(x_2 - x_1\right) \end{aligned}$$

→ Much more appealing form at the end:

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

Too soon to sense a pattern….let's see a degree-2 and nodes (x_1,y_1), (x_2,y_2), (x_3,y_3)…

Lagrange interpolation

$$\boxed{P(x) = a_1 + a_2 x + a_3 x^2}$$

$$\begin{cases} y_1 = a_1 + a_2 x_1 + a_3 x_1^2 \\ y_2 = a_1 + a_2 x_2 + a_3 x_2^2 \\ y_3 = a_1 + a_2 x_3 + a_3 x_3^2 \end{cases}$$

$$\begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

$$\boxed{P(x) = y_1 \underbrace{\frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)}}_{\mathcal{L}_1(x)} + y_2 \underbrace{\frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)}}_{\mathcal{L}_2(x)} + y_3 \underbrace{\frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}}_{\mathcal{L}_3(x)}}$$

- ✓ A pattern emerges. Each node y_i multiplies a polinomyal degree-N that is trivially a normalised product of $(x-x_j)$ terms with $j \neq i$ → ensures it is zero elsewhere !
- ✓ Indeed, the most trivial polymonial of degree-n only non-zero at node-i :

$$\mathcal{L}_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{n+1} \frac{(x-x_j)}{(x_i-x_j)}$$

$$\varepsilon_{\mathcal{L}_i}(x) = \frac{\left|f^{(n+1)}(\xi(x))\right|}{(n+1)!} \prod_{i=1}^{n+1} (x-x_i)$$

error

Newton divided difference interpolation

- ***Fundamental idea***: The polynomial expansion involves finite differences of increasing degree to ensure the polynomial goes through all nodes.

- Example: revisiting the ***linear interpolation*** with nodes (x_1,y_1), (x_2,y_2)

$$P(x) = a_1 + a_2(x - x_1) \longrightarrow P(x) = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

- Now for 3 nodes and 2nd degree polynomial…

$$P(x) = a_1 + a_2(x - x_1) + a_3(x - x_1)(x - x_2)$$

$$a_1 = y_1$$

$$a_2 = \frac{y_2 - y_1}{x_2 - x_1}$$

$$a_3 = \frac{y_3 - y_1 - \frac{y_2 - y_1}{x_2 - x_1}(x_3 - x_1)}{(x_3 - x_1)(x_3 - x_2)}$$

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

Less operations than Lagrange Interpolation !

Newton divided difference interpolation

- Generalizing, a pattern layout as shown below emerges:

x_1	$y_{[1]}$				
x_2	$y_{[2]}$	$y_{[1,2]}$			
x_3	$y_{[3]}$	$y_{[2,3]}$	$y_{[1,2,3]}$		
x_4	$y_{[4]}$	$y_{[3,4]}$	$y_{[2,3,4]}$	$y_{[1,2,3,4]}$	
x_5	$y_{[5]}$	$y_{[4,5]}$	$y_{[3,4,5]}$	$y_{[2,3,3,5]}$	$y_{[1,2,3,4,5]}$
$\ldots$	$\ldots$				

Recursive relation

$$y_{[1,\ldots,k]} = \frac{y_{[2,\ldots,k]} - y_{[1,\ldots,k-1]}}{\left(x_k - x_1\right)}$$

$$P(x) = y_{[1]} + y_{[1,2]}\left(x - x_1\right) + y_{[1,2,3]}\left(x - x_1\right)\left(x - x_2\right) + \ldots + y_{[1,2,3,\ldots,n]}\prod_{i=1}^{n-1}\left(x - x_i\right)$$

```
double NewtonInterpolator::DiffTable(int i, int j) {
  if (i == j)
    return y[i];
  else {
    return (DiffTable(i+1,j)-DiffTable(i,j-1))/(x[j]-x[i]);
  }
}
//This method is far from optimal, can you find why ?
```

```
double NewtonInterpolator::Interpolate(double xval) {
  double A,aux; aux = 1.0; A=y[0];
  for (int k=1; k<N; k++) {
    for (int i=0; i<k; i++)
      aux *= (xval-x[i]);
    A+=Ydiff[k]*aux;
    ........//something deliberatly missing here....
  }
  return A;}
```

Neville Interpolation

- Even slightly better than the Newton method and yet still *recursive*, there is another method know as ***Neville method***.
- As before, first with a few points, then generalised…

3-nodes example

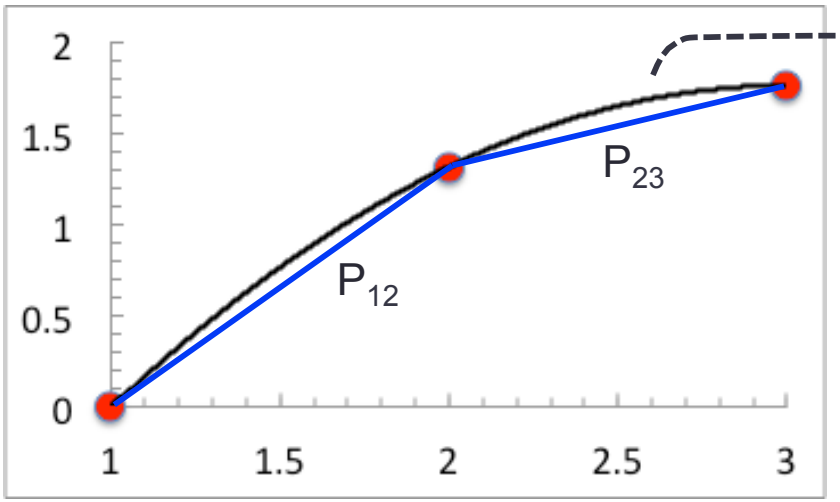
$$P_{12}(x) = \boxed{y_1}\frac{x - x_2}{x_1 - x_2} + \boxed{y_2}\frac{x - x_1}{x_2 - x_1}$$

$P_1(x)$ $P_2(x)$

$$P_{23}(x) = \boxed{y_2}\frac{x - x_3}{x_2 - x_3} + \boxed{y_3}\frac{x - x_2}{x_3 - x_2}$$

$P_2(x)$ $P_3(x)$

Lagrange linear int.



$$P_{123}(x) = \frac{(x - x_3)P_{12}(x) - (x - x_1)P_{23}(x)}{x_1 - x_3}$$

For a N-point set…

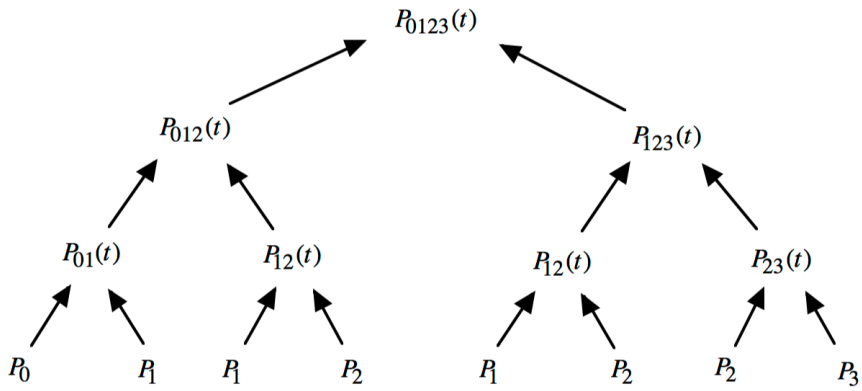
$$P_{123..N}(x) = \frac{(x - x_N)P_{12...N-1}(x) - (x - x_1)P_{23...N}(x)}{x_1 - x_N}$$

Neville Interpolation

- Algorithm implementation either via recursive function call (less efficient, top→bottom) or by appropriate looping (more efficient, bottom→top)

For a N-point set…(indexing from 0)

$$P_{012..N-1}(x) = \frac{(x - x_{N-1})P_{01...N-2}(x) - (x - x_0)P_{12...N-1}(x)}{x_0 - x_{N-1}}$$




```
double NevilleInterpolator::Interpolate(double xval) {
  double* yaux = new double[N];
  for (int i=0; i<N; i++) {
    yaux[i] = y[i]; // auxiliar vector
  }
  for (int k=1; k<N; k++) {
    for (int i=0; i<N-k; i++) {
      yaux[i] = ( (xval-x[i+k])*yaux[i] - (xval-x[i])*yaux[i+1]) / (x[i]-x[i+k]);
    }
  }
//Last value calculated is yp[0] when k=N-1 and i=0.....
  double A = yp[0];
  delete [] yp;
  return A;
}
```

Cubic spline Interpolation

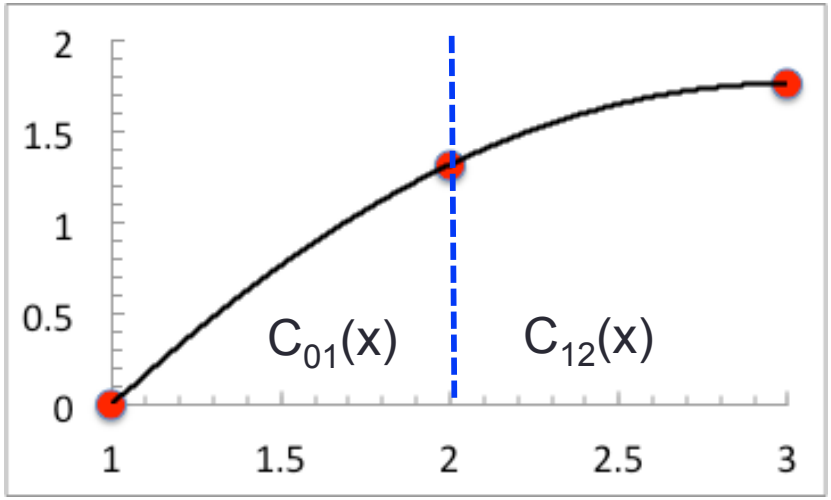
- Lagrange interpolators are useful but of limited practical use for large datasets
  - Can a 11th degree polynomial really be ***that*** good ? (*forget about Taylor series…*)
  - Isn't it plausible that as the degree increases, higher oscillation are observed ?
  - Can't we do any better e.g. imposing *lower degree* + "*continuity*" in the interpolation ?
- *Yes we can* ... cubic splines !

➢ ***Basic idea***:

- ➢ In between two consecutives data points (*nodes*), we shall define a cubic polynomial.
- ➢ In between two consecutive segments $[x_{i-1},x_i]$ and $[x_i,x_{i+1}]$, the two cubic polynomials shall have continuity in: **value**, **first order derivative**, **second order derivative**.
- ➢ At the extrema of the data points set (x_0 and x_{N-1}), some condition must be given to the second order derivative:
  - ➢ Natural boundary condition: $f''(x_0) = 0$, $f''(x_{N-1}) = 0$
  - ➢ Clamped boundary condition: $f''(x_0) = \alpha$, $f''(x_{N-1}) = \beta$ (α, β ***assigned***)

Cubic spline interpolation

- As before, first with a few points, to generalise later.
- *Outlook*: how many unknows and equations to solve we have ?



$$C(x) = a + bx + cx^2 + dx^3$$

Checklist

- ✓ $C_{01}(x_0)=y_0$, $C_{01}(x_1)=y_1$
- ✓ $C_{12}(x_1)=y_1$, $C_{12}(x_2)=y_2$
- ✓ ~~$C_{01}(x_1)=C_{12}(x_1)$~~
- ✓ $C'_{01}(x_1)=C'_{12}(x_1)$
- ✓ $C''_{01}(x_1)=C''_{12}(x_1)$
- ✓ $C''_{01}(x_0)=\alpha$
- ✓ $C''_{12}(x_2)=\beta$

2x 4 coefficients = 8
for
8 conditions…find the solution !

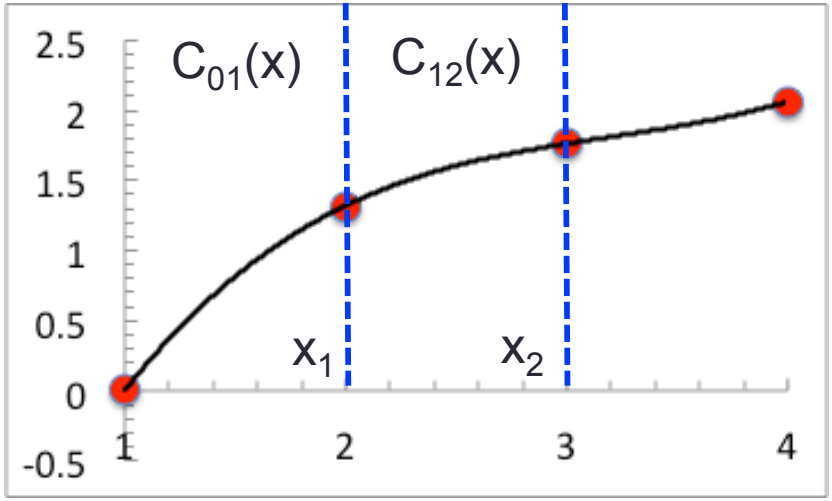
Cubic spline interpolation *(cont.)*

- ***Quick strategy***: start with continuity of second order derivative (*K*) and integrate ! Let's make it with 4 rather than just 3 points...

$$C''_{01}(x_1) = C''_{12}(x_1) = K_1 \qquad C''_{12}(x_2) = C''_{23}(x_2) = K_2$$

$$C''_{12}(x) = K_1 \frac{x - x_2}{x_1 - x_2} + K_2 \frac{x - x_1}{x_2 - x_1}$$

$$C_{12}(x) = \frac{K_1}{6} \frac{(x - x_2)^3}{x_1 - x_2} + \frac{K_2}{6} \frac{(x - x_1)^3}{x_2 - x_1} + Ax + B$$



Matching at endpoints x_1 and x_2

$$C_{12}(x_1) = \frac{K_1}{2}(x_1 - x_2) + Ax_1 + B = y_1$$

$$C_{12}(x_2) = \frac{K_2}{2}(x_2 - x_1) + Ax_2 + B = y_2$$

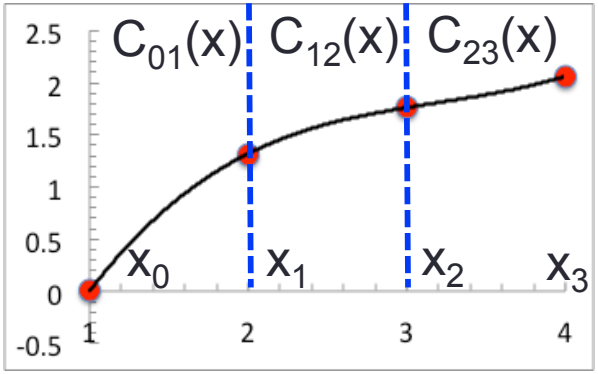
$$C_{12}(x) = \frac{K_1}{6}\left[\frac{(x - x_2)^3}{x_1 - x_2} - (x - x_2)(x_1 - x_2)\right] - \frac{K_2}{6}\left[\frac{(x - x_1)^3}{x_1 - x_2} - (x - x_1)(x_1 - x_2)\right] + \frac{y_1(x - x_2) - y_2(x - x_1)}{x_1 - x_2}$$

Cubic spline interpolation *(cont.)*

$$C_{12}(x)=\frac{K_1}{6}\left[\frac{(x-x_2)^3}{x_1-x_2}-(x-x_2)(x_1-x_2)\right]-$$
$$-\frac{K_2}{6}\left[\frac{(x-x_1)^3}{x_1-x_2}-(x-x_1)(x_1-x_2)\right]+\frac{y_1(x-x_2)-y_2(x-x_1)}{x_1-x_2}$$

Checklist

- ✓ $C_{12}(x_1)=y_1$, $C_{12}(x_2)=y_2$
- ✓ $C''_{12}(x_1)=K_1$
- ✓ $C''_{12}(x_2)=K_2$



Impose now continuity in first order derivate…

$C'_{01}(x_1)=C'_{12}(x_1)$ and $C'_{12}(x_2)=C'_{23}(x_2)$

$$K_0(x_0-x_1)+2K_1(x_0-x_2)+K_2(x_1-x_2)=6\left(\frac{y_0-y_1}{x_0-x_1}-\frac{y_1-y_2}{x_1-x_2}\right)$$

$$K_1(x_1-x_2)+2K_2(x_1-x_3)+K_3(x_2-x_3)=6\left(\frac{y_1-y_2}{x_1-x_2}-\frac{y_2-y_3}{x_2-x_3}\right)$$

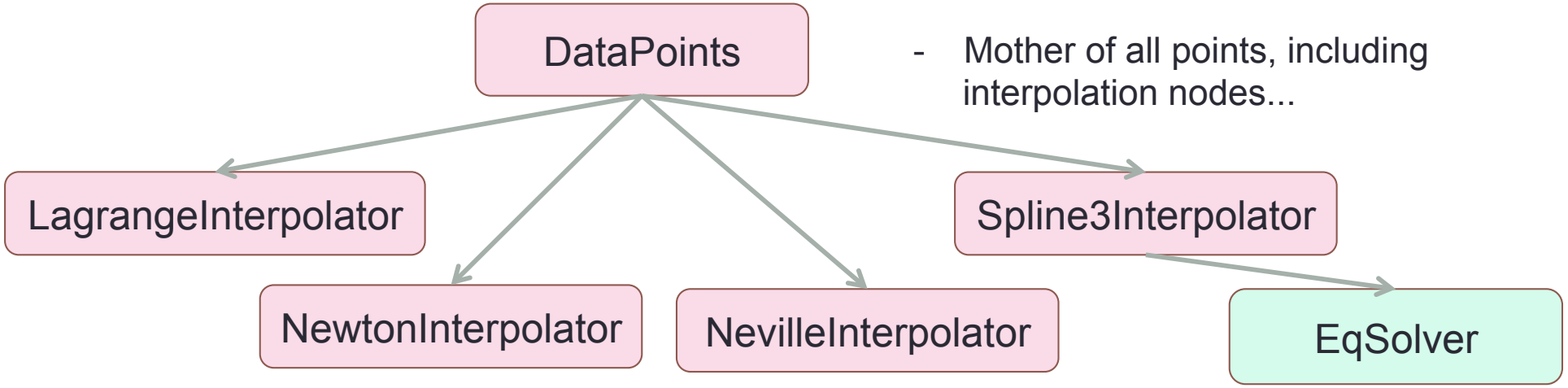
and K_0 and K_3 are known → Two equations for 2 unknowns ! → ***Time to generalize !***

Cubic spline interpolation *(cont.)*

$$\begin{pmatrix} 2(x_0-x_2) & (x_1-x_2) & & & & \\ (x_1-x_2) & 2(x_1-x_3) & (x_2-x_3) & & & \\ & (x_2-x_3) & 2(x_2-x_4) & (x_3-x_4) & & \\ & & \dots & \dots & \dots & \\ & & & (x_{N-3}-x_{N-2}) & 2(x_{N-3}-x_{N-1}) & (x_{N-2}-x_{N-1}) \\ & & & & (x_{N-2}-x_{N-1}) & 2(x_{N-2}-x_N) \end{pmatrix} \begin{pmatrix} K_1 \\ K_2 \\ K_3 \\ \dots \\ K_{N-2} \\ K_{N-1} \end{pmatrix} = \dots$$

- ✓ The r.h.s. contains both information on the nodes coordinates ***and*** boundary conditions.
- ✓ The linear system of equations is suitably solved using the ***EqSolver*** class with the ***FCBanded*** matrix objects !
- ✓ Let's see now what a possible representation for our *final class hierarchy* should look like

Class scheme suggested



- Mother of all points, including interpolation nodes...

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

  virtual double Interpolate(double x) {return 0.;}
  virtual void Draw();
  virtual void Print(string FILE="");
 protected:
  int N; // number of data points
  double *x, *y; // arrays
  static int Nplots;
};
```

```
void Spline3Interpolator::SetCurvatureLines() {
…
   FCmatrixBanded Tri_mat(ld,md,ud);
   EqSolver banded(Tri_mat,tri_b);
   //Solve the system....
   Vec result;
   result=banded.TridiagonalSolver();
   // Assign the private member K[] array pointer…
}
double Spline3Interpolator::Interpolate(double fx) {
  double A;
  // detect in which segment is x, if outside flag as extrapolation !
   // it is pointless to use all the spline functions...we need just one !
   …
   return A;
}
```

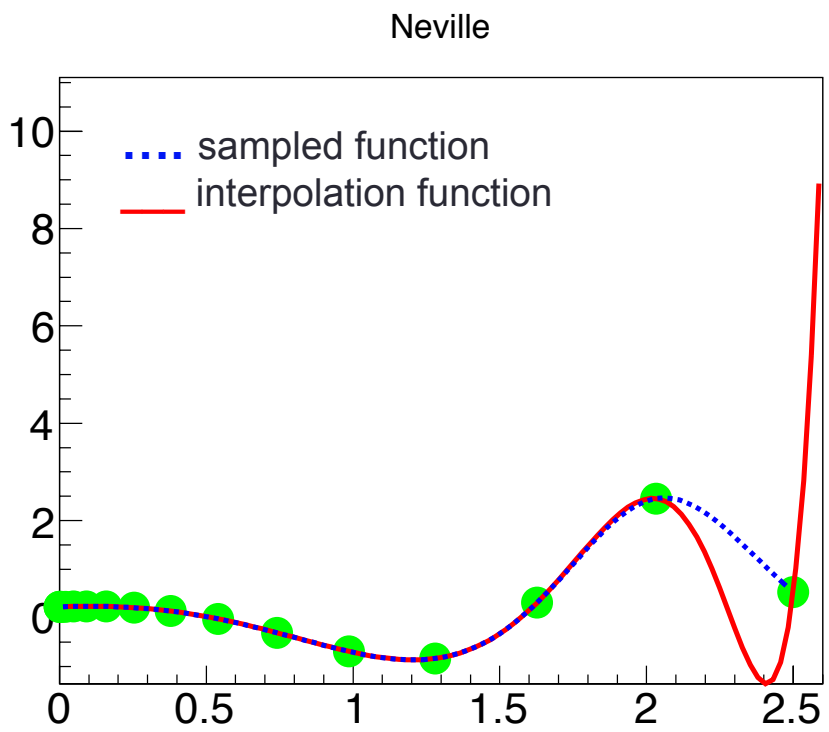
Class scheme suggested – Data Points

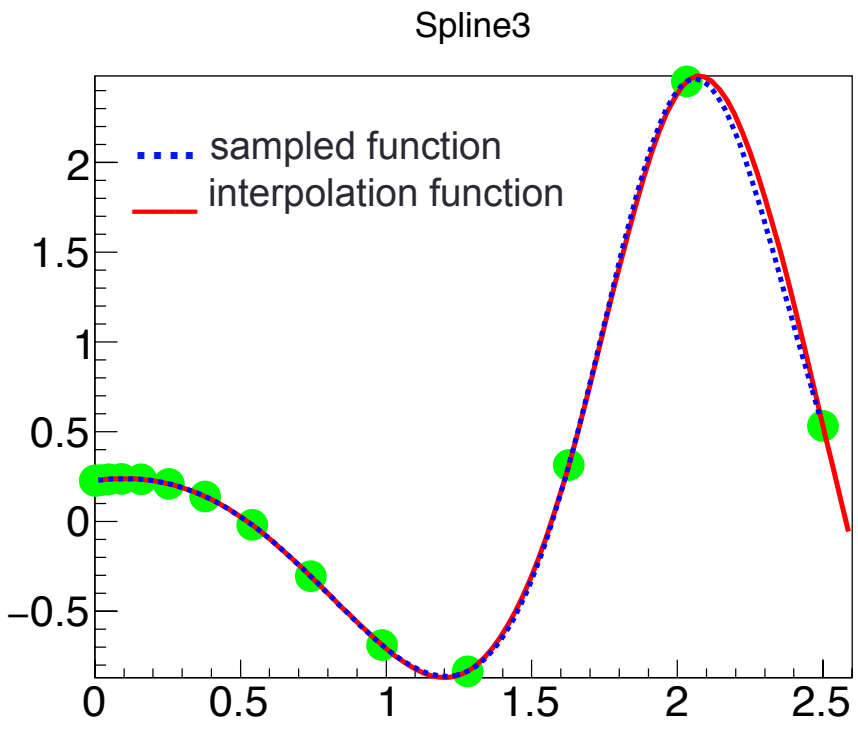
```
#include "DataPoints.h"
#include "TGraph.h"
#include "TApplication.h"
#include "TCanvas.h"
#include <cstdio>

int DataPoints::Nplots=0; //Tapplication only for first plot !
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];
    printf("[Datapoints] x=%f, y=%f \n", x[i], y[i]);
  }
}

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

```
void DataPoints::Draw() {
  TGraph *g = new TGraph(N,x,y);
  g->SetMarkerStyle(20);
  g->SetMarkerColor(kRed);
  g->SetMarkerSize(2.5);
  if (Nplots == 0) {
    //create application
    TApplication * MyRootApp;
    MyRootApp = new TApplication(”click twice", NULL, NULL);
    MyRootApp->SetReturnFromRun(true);
  }
  TCanvas *c0 = new TCanvas("c0","c0",600,500);
  g->Draw("PA");
  c0->Update();
  gPad->WaitPrimitive();
  delete g;
  Nplots++;
}
```

- ✓ In every derived class object, a new canvas on same instance of Tapplication to draw points+interpolating function+…
- ✓ *Interpolating function ?...(Practical session)*

Interpolation - benchmark





- ✓ Lagrange interpolator is fine for *short number of nodes* and preferably *equally spaced*
- ✘ Typically fails miserably to extrapolate even at close range and when node spacing is sparse...

- ✓ Cubic spline outsmarts Lagrange interpolation for larger sets → worth the penalty of system solving...
- ✓ Decent at extrapolating but can still show some wiggles at the extrema nodes → ***Runge phenomena***

Interpolation - final remarks

- ✓ The cubic spline *outperforms* Lagragian interpolators....period.
- ✓ The penalty for solving a linear system is worth it (*tridiagonal system* is quick !)
- ✓ Lagrange interpolators of increasing order requires *N^{th} order* polynomial → *large oscillations* **plague** the interpolation and *extrapolation* is **off-limits**
  - ✓ For large datasets either use a *linear interpolant* between consecutive nodes or…
  - ✓ Break the dataset in *segments* of 3 to 6 points each → Neville on each segment !
- ✓ Cubic splines, though of lower degree (3), ensure *continuity* of interpolating function of 0th, 1th and 2nd order derivatives → a must have in practical applications e.g. *estimating acceleration from interpolated displacement.*
- ✓ About cubic spline interpolation error:

$$\varepsilon_{CubicSp}(\mathrm{x}) \leq \frac{h^4}{(n+1)!}\left|\max_{x\in[x_0,x_N]} \mathrm{f}^{(4)}(x)\right| \quad \text{with} \quad h = \max\left|x_i - x_{i-1}\right|$$